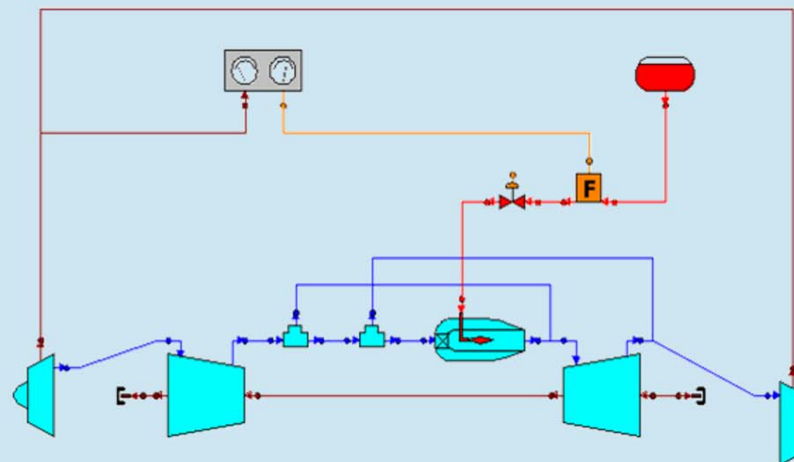
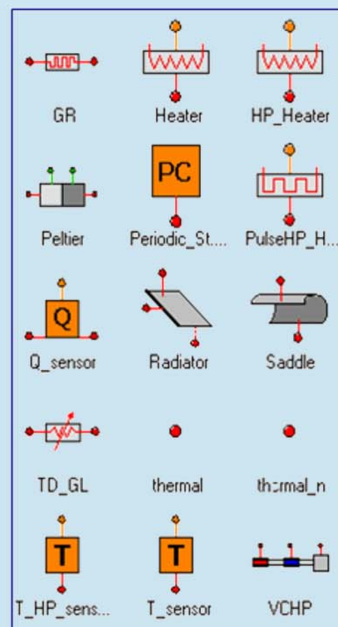
EcosimPro

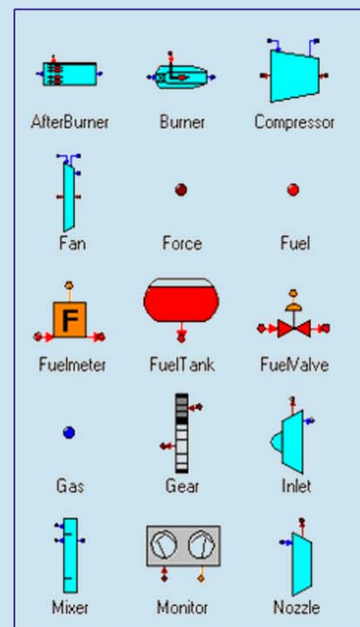
Modelling and Simulation Software

EL Modelling Language

Version 4.4







EA Internacional

© 1992-2008 EA Internacional. All rights reserved.

CONTENTS

1. INTRODUCTION 1

1.1. EXECUTIVE SUMMARY 1
1.2. OBJECT-ORIENTED MODELLING 2
1.2.1. Encapsulation 2
1.2.2. Inheritance 3
1.2.3. Aggregation 3
1.2.4. Data Abstraction 4
1.3. ORGANIZATION IN LIBRARIES 4
1.4. CONTINUOUS AND DISCRETE MODELLING 5
1.5. HIDDEN POWER 5
1.6. ECOSIMPRO MODELLING FILES 6
1.7. HOW TO USE THIS MANUAL 7
1.8. DEFINITIONS, ACRONYMS AND ABBREVIATIONS 8

2. LANGUAGE ELEMENTS 9

2.1. CHARACTERS 9
2.2. SEPARATORS 9
2.3. COMMENTS 10
2.4. NAMES 12
2.5. RESERVED WORDS 12
2.6. LITERALS 13
2.6.1. Integer Literals 13
2.6.2. Floating Point Literals 13
2.6.3. Boolean Literals 13
2.6.4. String Literals 13
2.6.5. Aggregate Literals 14
2.7. EXPRESSIONS 14
2.7.1. Arithmetic Expressions 14
2.7.2. Relational Expressions 16
2.7.3. Logical Expressions 17

3. BASIC DATA TYPES 19

3.1. CONCEPTS 19
3.2. FUNDAMENTAL DATA TYPES 20
3.2.1. REAL 21
3.2.2. INTEGER 22
3.2.3. BOOLEAN 22
3.2.4. STRING 23
3.2.5. FILEPATH 23
3.2.6. TABLES 25
3.2.7. FUNC_PTR 26
3.2.8. OBJECT_PTR 27
3.3. ENUMERATION TYPES 27
3.4. DERIVED TYPES 28
3.4.1. Arrays 28
3.4.2. SET_OF Dynamic Enumeration Types 30
3.4.3. SET_OF Default Values 32
3.5. CHARACTERISTICS OF VARIABLES 33
3.5.1. Constants 33

3.5.2. Units ... *33*
3.5.3. Description of Variables ... *34*
3.5.4. Variable Range ... *34*
3.5.5. Scope of Variables ... *34*
3.5.6. Global Variables and Types ... *37*
3.6. TIME, TSTOP AND CINT GLOBAL VARIABLES ... 37

4. LIBRARIES ... 39

4.1. WHAT IS AN EL LIBRARY? ... 39
4.2. LIBRARY VERSIONS ... 40
4.3. USING LIBRARIES IN EL ... 41
4.4. FORCING COMPILATION INTO A LIBRARY ... 42
4.5. MANIPULATION OF LIBRARIES ... 42
4.6. SYSTEM LIBRARIES ... 42
4.7. AUTOMATIC DOCUMENTATION ... 43

5. STATEMENTS ... 47

5.1. SEQUENTIAL STATEMENTS ... 47
5.1.1. Assignments ... *48*
5.1.2. Function Calls ... *49*
5.1.3. Special Functions for Printing Messages ... *49*
5.1.4. IF-THEN-ELSE Statement ... *55*
5.1.5. WHILE Statement ... *56*
5.1.6. FOR Statement ... *56*
5.1.7. ASSERT Statements ... *57*
5.1.8. RETURN Statement ... *58*
5.1.9. STOP Statement ... *58*
5.2. CONTINUOUS STATEMENTS ... 59
5.2.1. Mathematical Equations ... *60*
5.2.2. Continuous EXPAND Statement ... *61*
5.2.3. Conditional Insertion of Equations ... *62*
5.2.4. Use of Conditional Equations with Detection of Events ... *65*
5.2.5. Use of Conditional Equations without Detection of Events ... *66*
5.2.6. Sequential Blocks of Equations ... *68*
5.3. DISCRETE STATEMENTS ... 70
5.3.1. WHEN Statements ... *70*
5.3.2. ASSERT Statement in DISCRETE Block ... *73*
5.3.3. Discrete EXPAND Statement ... *73*

6. PORTS ... 77

6.1. PORT DEFINITION SYNTAX ... 78
6.2. INHERITANCE ... 79
6.3. CONTINUOUS BLOCK ... 80
6.4. PORT CONNECTING EQUATIONS ... 80
6.5. PORT CONNECTING RESTRICTIONS ... 82
6.6. BOUND AND EXPL OPERATORS ... 83
6.7. EXAMPLES OF PORT TYPES ... 83

7. COMPONENTS ... 85

7.1. INTRODUCTION ... 85
7.2. SYNTAX ... 85
7.3. ABSTRACT COMPONENTS ... 87
7.4. INHERITANCE ... 88
7.5. PORTS BLOCK ... 90

7.6. CATEGORIES OF VARIABLES 91
7.6.1. Construction Parameters 91
7.6.2. DATA Block 93
7.6.3. DECLS Block 93
7.7. OBJECTS BLOCK 98
7.8. TOPOLOGY BLOCK 99
7.8.1. Use of Aggregated Components 99
7.8.2. Using EXPAND for Multiple Connections 100
7.8.3. PATH Statement 101
7.9. INIT BLOCK 105
7.9.1. Priorities for INIT Blocks 105
7.10. DISCRETE BLOCK 107
7.11. CONTINUOUS BLOCK 110
7.11.1. Virtual Equations 113
7.12. CONTINUOUS AND DISCRETE MODELLING 114
7.13. ADVANCED MODELLING 115
7.13.1. INVERSE Statement 115
7.13.2. IMPL Operator 117

8. FUNCTIONS 119

8.1. INTRODUCTION 119
8.2. FUNCTION SYNTAX 120
8.2.1. Argument List 120
8.2.2. DECLS Block 123
8.2.3. OBJECTS Block 123
8.2.4. BODY Block 124
8.3. USING EXTERNAL C, C++ AND FORTRAN FUNCTIONS 124
8.4. SPECIAL FUNCTIONS 126
8.4.1. The delay() Function 126
8.4.2. Reading Tables from a File 127
8.4.3. Interpolation in tables 131
8.4.4. Waveform Functions 140
8.4.5. LOG_STATUS() Function 142
8.4.6. Obtaining Array Dimensions during Runtime 142
8.4.7. NEW_BRANCH() Function 145
8.4.8. Function to obtain Units of Variables 149
8.4.9. Function to obtain the Release Report 149
8.4.10. Function to obtain Internal Residues 150
8.4.11. Deactivate the solving of internal algebraic boxes. 151
8.4.12. Function to issue errors from EL 151

9. PARTITIONS 153

9.1. REMOVE DERIVATIVES 154
9.2. CHANGE DATA TO UNKNOWN VARIABLES 154
9.3. BOUNDARY CONDITIONS 155
9.4. TEARING ALGEBRAIC LOOPS 155
9.5. HIGH INDEX PROBLEMS 156

10. EXPERIMENTS 157

10.1. DECLS BLOCK 158
10.2. OBJECTS BLOCK 158
10.3. INIT BLOCK 159
10.3.1. Self-Initializing Components 160
10.4. BOUNDS BLOCK 161

10.5. BODY BLOCK 162
10.5.1. Experiment Functions and Variables 162
10.5.2. Steady States 167
10.5.3. Integration Methods 167
10.5.4. Integration at Time Zero 168
10.5.5. Detecting Type of Ongoing Calculation 169
10.5.6. Report Generation 169
10.6. FUNCTIONS IN EXPERIMENTS 172
10.7. INITIALIZATION ISSUES 173
10.7.1. EXEC_INIT() Function 174
10.7.2. SET_INIT_ACTIVE() Function 175
10.7.3. EXEC_EVENTS() Function 177
10.8. CHANGING THE PRECISION OF THE OUTPUT DATA 177
10.9. CHANGING THE MOMENT TO REFRESH DATA 178
10.10. NSTEPS: A WAY TO REDUCE THE INTEGRATION STEP 180
10.11. SAVE AND RESTORE SNAPSHOT OF SIMULATIONS 181
10.12. SYNCHRONISING TIME WITH REAL TIME 183
10.13. DISABLING THE ROOT FINDER MECHANISM 184
10.14. GETTING STATISTICS OF JACOBIAN EVALUATIONS AND RESIDUES FUNCTIONS 185
10.15. CHANGE THE LEVEL OF DEBUGGING 186
10.16. DEBUGGING MATHEMATICAL MODELS 187
10.16.1. Check Integration Step Reduction 188
10.16.2. Check Residues Function Repeatability 189
10.16.3. Check the Nonlinearity of the Jacobian 190
10.17. HELPING DASSL_SPARSE METHOD 193
10.18. RESIDUES FUNCTION CALLS 193

11. CLASSES 195

11.1. INTRODUCTION 195
11.2. INHERITANCE 196
11.3. DECLS BLOCK 198
11.4. OBJECTS BLOCK 199
11.5. METHODS BLOCK 200
11.6. USING CLASSES 201
11.7. CLASS ASSOCIATED WITH A PARTITION 202
11.7.1. Access to Variables during Simulation 203
11.7.2. Operations Allowed with Classes of Models 207
11.7.3. Activation of Flags 208
11.7.4. Creation of Classes based on Partition Classes 209

12. USING C++ CLASSES FROM EL 211

12.1. INTRODUCTION 211
12.2. VALID DATA TYPES 213
12.3. USE OF BOOLEAN VARIABLES 214
12.4. USE OF ENUMERATIVE VARIABLES 214
12.5. USE OF STRING VARIABLES 215
12.6. USE OF OBJECT ARRAYS 216
12.7. INCLUDE FILES 216
12.8. OBJECT LIBRARIES 216

13. STEADY STATE CALCULATIONS 219

13.2. CHANGING PARAMETERS FOR THE STEADY SOLVER 225
13.3. FREEZING VARIABLES 227
13.4. STEADY SOLVER DEBUGGING INFORMATION 228

13.5. STEADY STATE CALCULATION EXAMPLE 231
13.6. EXTERNAL NEWTON-RAPHSON FUNCTIONS 232

14. LIBRARY OF PREDEFINED CLASSES 237

14.1. TABLE CLASS 237
14.1.1. Table reading 239
14.1.2. Reset table 240
14.1.3. Obtaining the dimensions 240
14.1.4. Interpolation in TABLE class 240
14.1.5. Print 244
14.2. SYSTEM_IOREADER CLASS 244
14.2.1. SYSTEM_IOREADER reading 245
14.2.2. Functions for extracting variables 245
14.2.3. Functions for extracting report related information 246
14.2.4. Functions for extracting generic information 246

15. APPENDIX A: REGULAR EXPRESSIONS 247

16. APPENDIX B: THE SYSTEM LIBRARIES 249

17. APPENDIX C: SYNTAX OF EL 257

18. APPENDIX D: VALIDITY OF STATEMENT INSTRUCTIONS 265

19. APPENDIX E: LIST OF ERRORS 267

20. APPENDIX F: VALID FORMATS FOR WRITE() FAMILY OF FUNCTIONS 275

20.1. FLAG DIRECTIVES 276
20.2. WIDTH SPECIFICATION 276
20.3. PRECISION SPECIFICATION 277
20.4. WRITE TYPE FIELD CHARACTERS 278

21. APPENDIX G: XML FORMATS 281

21.1. TABLES IN XML 281
21.2. SYSTEM_IOREADER IN XML 284

22. INDEX BY WORDS 289

1. Introduction

1.1. Executive Summary

This document describes the language used in EcosimPro simulation tool for modelling dynamic systems based on equations and discrete events. The language is named EL.

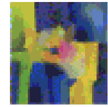
EL has been developed for use in modelling combined continuous-discrete physical systems. It allows mathematical modelling of complex components represented by differential-algebraic equations. One of the design goals for EL was to make it clear and easy to use for engineers doing simulations of this kind. As this Manual shows, models in EL are represented in a natural and intuitive way.

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.

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.

Once a model has been generated, experiments can be run on it. EL has an experiment language similar to the modelling language which is used to integrate the model, calculate steady states, optimize parameters, etc.



EL's design is based on continuous modelling concepts developed in the 1970s, new ideas of the '80s and '90s, and modern object-oriented techniques applied successfully in other fields. It also draws on 35 years' experience of a large number of engineers working in the field of continuous simulation of industrial processes and aerospace applications.

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.

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.

1.2. Object-Oriented Modelling

Object-oriented modelling 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.

1.2.1. Encapsulation

One of the aims of object-oriented modelling 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 modelling 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 modelling, 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.

1.2.2. Inheritance

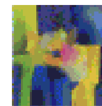
Inheritance is what gives EL its tremendous power for sharing interfaces and behaviour. In all fields of simulation, when many components are developed it becomes apparent that they share much of the same behaviour. EL can bring together common data and equations in parent components, to be inherited 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.

1.2.3. Aggregation

In EL, all modelling 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.

1.2.4. 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 modelling).

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

1.3. Organization in Libraries

Libraries are the basic elements around which EL organizes the information it manipulates. A library's basic purpose is to store a set of related items.

As well as keeping work organized, working with libraries allows the modelling environment to provide other services, such as:

- Checking for obsolete items (components, ports, etc) which should be recompiled
- Encapsulation of items, allowing different libraries to contain items with the same name

An EL library can contain the following types of items:

- Components
- Connection ports
- EL functions
- Classes
- Global variables and constants
- Global enumeration types
- LIBRARY and USE directives for using other libraries

1.4. Continuous and Discrete Modelling

EL provides features for continuous modelling. With continuous modelling, the user expresses the physical model with mathematical equations particular to the model. These equations may be algebraic or derived with respect to time. They may then be processed and sorted so that the final model optimally calculates the unknown variables in the system. To do this, EL has powerful equation processing algorithms which include checking any singularities of the system, choosing the most likely boundary variables and detecting whether the problem being solved is high index. As well as diagnosing problems, EL provides help and proposes solutions.

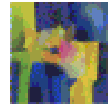
EL not only provides continuous modelling. Any physical model can be constructed with continuous equations, but they normally behave discretely. For example, a ball bouncing on the ground can be simulated with kinematics equations, but it is important to take into account that when it hits the ground, a bounce occurs, and this event has to be treated. EL provides powerful features for handling this type of event. Each type of continuous behaviour is associated with a series of events which is checked continuously. When any of them occur, an associated series of actions is executed. This means that components are modelled using both modelling methods: continuous and discrete.

EL's internal processes ensure that both the continuous and discrete parts execute in a co-ordinated way.

EL has three types of statements: continuous, discrete and sequential. The first two are those described above. Sequential statements are those used in classical programming languages like FORTRAN and C and are always executed sequentially. This Manual distinguishes clearly the three types and in some cases, such as in component definition, they are expressed in separate blocks. Sequential statements are described in their own section; discrete statements are described in the section on the component's DISCRETE block; continuous statements are described in the section on the CONTINUOUS block.

1.5. Hidden Power

As this Manual shows, EL is a natural modelling language which simplifies the component modeller's work. But EL's true power lies hidden in its internal algorithms and mathematics. EL statements which clearly describe a component or port are elaborated internally to check for consistency and validity. Continuous equations are sorted, modified symbolically and checked to ensure there are no problems such as singularity, high index, redundant variables, etc. Discrete statements are elaborated to create discontinuity conditions of the continuous system and provide powerful event handling which interacts with the continuous statements in a transparent way.



This means that the most complex part of EL is its internal processing of components and ports, hidden from the user. EL incorporates 35 years' experience of dozens of engineers in the field of continuous simulation of complex systems and the latest advances in numerical solution of differential-algebraic equations, optimum sorting algorithms, etc.

These are some of EL's internal math capabilities:

- Intelligent connection ports with automatic generation of equations in multiple connections
- Manipulation of differential-algebraic equations (DAE) and ordinary differential equations (ODE)
- Symbolic equation processing
- Wizard for solving algebraic loops
- Wizard for suggesting boundary conditions
- Wizard for resolving high index mathematical problems
- Intelligent equation sorting, optimizing the number of unknown variables
- Detection and resolution of linear combinations between equations
- Automatic detection of equivalent variables in models with complex topologies
- Automatic handling of discontinuities in the continuous model
- Analysis of structurally singular Jacobian functions
- Powerful root-finder mechanism
- Robust, tried and tested differential-algebraic equation solvers (DASSL)
- Handling of scalar variables, vectors and matrices of any dimensions
- Handling of data tables with 1, 2 and 3 dimensions
- Use of powerful mathematical operators to express equations (SUM, EXPAND, etc)
- Assertions to check the integrity of variables at any time

1.6. EcosimPro Modelling Files

When modellers want to create a model in EcosimPro they have two options: either create a text file with the components in EL or graphically design the component using the graphics tool.

An EL text file (.el) can include one or many components, ports, global variables, enumeration types or functions. Experiments are an exception; an experiment file can contain only one experiment and the functions used in the experiment.

The EL text file has to be compiled in the EcosimPro GUI and all the items of this file (components, functions, etc) are stored in binary format in the corresponding library. The user can then change these components to create experiments associated with them.

The other way to create components is graphically. In this case the modeller uses the graphics tool to create a schematic and saves it in a file. This file contains a schematic which in EcosimPro will be a component. Refer to the User Manual to read about how to create components graphically.

1.7. How to Use This Manual

This manual is not intended to be read chapter by chapter; you can use it according to your needs.

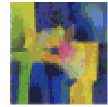
We propose that you then read quickly through chapters 2 and 3 to get a rough idea about the different data types allowed in EL and expression construction. Read chapter 4 about libraries in EL. Chapter 5 presents the different types of allowed statements; to get an overall view, this chapter could be read after reading chapters 6 and 7.

Read chapter 6 about connection ports. Because the connection port type can range from very simple to very complex, you shouldn't try to understand all the concepts at once. Throughout your entire career as a simulation programmer you will probably never require many of these capabilities.

Chapter 7 talks about components in EL. This is the most important chapter in this manual, because the components are the kernel of EL. From here you can go on to read in Chapter 5 about the different types of statements allowed in EcosimPro: sequential, continuous and discrete.

Chapter 8 shows you the functions in EL. They are quite similar to functions in other languages like FORTRAN or C. Read carefully about the different default functions provided in EcosimPro.

Chapter 9 explains the concept of partition. A partition is the mathematical model generated from a component based on user interaction. The partition may be the most abstract concept in EL since it is an intermediate stage between components and experiments, but after you have made your first few models in EL it will come to you naturally.



Experiments are explained in Chapter 10. They are the final EL files for creation of simulation cases. They are simple to use and the GUI provides very convenient default experiments.

Chapter 11 explains the classes, another way to organise information in EL based on the classical concept of languages such as C++ or Java. Chapter 12 explains how to connect external C++ classes to EL, a very useful feature for connecting the tool to many existing software tools and libraries.

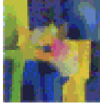
Chapter 13 explains how to make steady state calculations in the language.

Chapter 14 presents some useful classes already defined in EL.

The appendices contain a lot of useful information about regular expressions, error lists, language grammar, etc.

1.8. Definitions, Acronyms and Abbreviations

- **EA**: Empresarios Agrupados
- **EL**: Ecosim Pro language
- **MMI**: Man-machine interface
- **NR**: Newton-Raphson



2. Language Elements

2.1. Characters

The basic elements of EL code are written with characters. EL represents characters internally using ASCII standard and ISO format. Legal characters are:

Upper case:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Lower case:

a b c d e f g h i j k l m n o p q r s t u v w x y z

Numbers:

0 1 2 3 4 5 6 7 8 9

Special characters such as white space, tab, newline and:

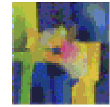
. _ " + - * / > < = , ( ) { } [ ] ! \ % $

2.2. Separators

EL allows the use of a number of special characters as separators. The most important separator is the newline character as it is used to separate statements. This means that EcosimPro cannot have two statements on the same line, as the newline character is the delimiter. For example, the following is not correct:

```
REAL x INTEGER i    -- ERROR !
   This should be written:
REAL x
   INTEGER i
```

There is a special separator character to put a statement on several lines; this is the character '\'. For instance the user can write:



```
x' - 3*y = log(z) -                    \
      6*k
```

This is a unique statement on two lines. The table below specifies all the separators permitted in EL and their usage:

Symbol	Name	Used to
[newline]	newline	Separate statements and comments
[white space]	white space	Separate names and reserved words
[tab]	tabulator	Separate names and reserved words
( )	parentheses	Include initialization of construction parameters, data, functions, arguments, etc
[ ]	square brackets	Create array indexes
{ }	brackets	Initialize arrays
,	comma	Separate declarations, dimensions in array indexes, ENUM elements, initializations
;	semicolon	Separate blocks in FOR statements
.	dot	Compose aggregate names, create decimal values
'	quotation mark	Write derivative names
&	ampersand	Separator of EL statements (it can be used as an alternative to newline separator). It is used as well to separate regular expressions
%	percent	In PRINT functions, to specify a variable name
$	dollar	In PRINT functions, to specify a variable value
\	back slash	Continue a statement on the next line

2.3. Comments

Program code should be documented using comments to describe each action and the purpose of each variable, component, etc. Large programs can be difficult to understand, even for the programmer who wrote the code. A well-commented program is the mark of a good programmer. A program that is well documented and described will be easier to maintain and will allow new programmers to understand the complexity without undue effort.

EL is not only a conventional sequential language, it is also a language for continuous and discrete modelling. This means that comments are included not only to describe the next algorithm, but also to describe the physical equations of the continuous part and the events in the discrete part. As EL is not purely sequential, it is not easy to follow and the use of adequate comments soon becomes a necessity.

Comments can be inserted in any part of EL code. The format is simple: all comments begin with two minus signs "--" and end with the newline character. For example:

```
   pi.v - po.v = pi.i * r    -- Ohm's law
REAL v                        -- voltage
   CONST REAL STEFAN  = 5.67e-8  --Stefan Constant
```

Also valid are the C/C++ special characters for comments: "//" and "/* … */". The first type of comment is identical to the previous one; it ends at the end of the line. The second is used when the modeller wants to comment several lines in one go. Some examples:

```
pi.v - po.v = pi.i * r   // Ohm's law
/*  next lines are commented
COMPONENT foo
END COMPONENT
*/
```

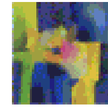
When the user declares aggregation of variables and objects of components (construction parameters, connection ports and data), ports and functions, he can use the following format to include comments:

```
REAL speed                 "Speed of rocket at launch (m/s)"
   BOOLEAN isOpen             "Is the valve open?"
```

The advantage of this format over the previous one is that the comment you introduce will be stored internally and shown each time you document the library, call up an object editor in the graphical tool, etc. Try always to put the units in brackets "("and")" and the system will extract them automatically. We recommend putting the units at the end of the comment; e.g.:

```
REAL  sp                  "speed of vehicle (m/s)"
```

Using this format, the units will be extracted for creating the library documentation.



2.4. Names

Identifiers are used in EL to identify variables, components, ports, etc. An identifier can be any string of letters, numbers and an underscore '_', but the first character must be a letter and the last character must be a letter or number. All names must follow the rule:

```
IDENTIFIER    ::=    letter ( '_' | letter | digit )*
                     letter | digit
letter     ::=  'a' .. 'z' | 'A' .. 'Z'
digit      ::=  '0' .. '9'
```

EL is case sensitive. This means that: **speed**, **Speed** and **SPEED** are different identifiers. Names can use a mix of uppercase and lowercase letters but this manual follows the convention of writing user names in lowercase. EL does not allow the use of any of its reserved words as identifiers.

EL also allows the use of the character “'” to indicate derivatives; e.g., v' would be the first derivative of v, v'' the second, v''' the third, and so on.

The following examples are valid identifiers:

```
speed    v_134    PI    FirstValue  G_POSITION1    var0
this_is_a_variable_with_long_name   v'    v''    v'''
```

The following examples are not valid:

```
567   $val  _speed  6vmax  WHILE   while While .name 'v  t_
```

2.5. Reserved Words

Reserved Words are identifiers with a special meaning in EL. All reserved words are in uppercase. As described above, they cannot be used as identifiers of variables, components, etc. EL reserved words are (in alphabetical order):

```
ABSTRACT, AFTER, ALG, AND, ASSERT, BODY, BOOLEAN, BOUND, BOUNDS, CARDINALITY,
CATALOGUE, CLASS, CLOSE, COMPONENT, CONNECT, CONST, CONTINUOUS, DATA, DECLS,
DISCR, DISCRETE, ELSE, ELSEIF, END, ENUM, EQUAL, ERROR, EXCEPT, EXPAND,
EXPAND_BLOCK, EXPERIMENT, EXPL, EXTERN,   FALSE, FATAL, FILEPATH, FOR,
FUNC_PTR, FUNCTION, IF, IMPL, IN, INCLUDE, INIT, INSERT, INTEGER, INVERSE,
IS_A, KILLPOINT, LIBRARY, METHOD, METHODS, NO_TYPE, NOT, NOTE, OBJ_PTR,
OBJECTS, ON, OR, OTHERS, OUT, PATH, PORT, PORTS, PRIORITY, PRIVATE, RANGE,
REAL, RETURN, SEQUENTIAL, SET_OF, SINGLE, STOP, STRING, SUM, TABLE_1D,
TABLE_2D, TABLE_3D, THEN, TO, TOL, TOPOLOGY, TRUE, UNITS, USE, VERSION,
WARNING, WHEN, WHILE and ZONE
```

2.6. Literals

Constant values in EL are represented literally. Integers, floating point numbers, Boolean and strings are used frequently in EL code.

2.6.1. Integer Literals

These represent integer values (positive and negative). They are normally used to represent discrete quantities; for example, 1, 2, 67. They are decimal values represented in base 10. As described below, they are declared as type INTEGER. EL represents them in memory as 32 bit integers.

2.6.2. Floating Point Literals

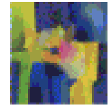
These represent real number values. They are used in EL to represent continuous quantities and correspond to the REAL type. They can be written in scientific or decimal notation. Decimal notation consists in using a number of decimal digits followed by a decimal point (.) followed by a number of fractional digits. For example 34.78, -2.65. With scientific notation the floating point number is expressed as a multiple of the largest power of the base less than the given number. The exponent is indicated with a letter 'e' or 'E' followed by a decimal number which can be positive or negative. For example, 8.334E12, 434434e7, -2e+34. In EL, floating point numbers are 64 bit.

2.6.3. Boolean Literals

A Boolean value can only take the values TRUE or FALSE and correspond to the BOOLEAN type. Unlike other languages, EL does not allow boolean values to be mixed with integers; they are incompatible and have a different purpose.

2.6.4. String Literals

Character strings in EL are enclosed in double quotation marks (" "). For example, "this is a string" represents a string which can be represented in memory and which has a series of allowed operations. In EL there are two types: STRING and FILEPATH.



2.6.5. Aggregate Literals

This data type is for aggregate data. For example:

```
{1, 2, 3}
{ {3,4, 87,7}, {3.33 , 4.0, 3.}}
```

These are used to initialize arrays and tables. The format is that each dimension is grouped in brackets, the values separated with commas. There are as many blocks as there are dimensions in the array or table. The practicalities of using these types are described below in the section on this data type.

2.7. Expressions

An expression is a formula which defines the calculation of a value. EL supports arithmetic, relational and logical expressions. Expressions are used as operators to link actions and generally to create the power necessary to build any type of arithmetical or logical operation.

Expressions are grouped using brackets, in the order in which the expression is to be evaluated. They are normally used when operator precedence alone is not sufficient to define an expression unambiguously.

Operators are special words or characters which indicate the execution of some operation on the operands. Some operators take a single value and are called unary operators (e.g., NOT flag, -8). Most operators are placed between two operands and are called binary operators (e.g., 8 + 9, v AND b). EL provides arithmetic, relational and logical operators.

Functions that return values can be called from within functions, and are treated as operands. Some of these functions can even modify call parameters.

2.7.1. Arithmetic Expressions

These are used in arithmetic operations. The BNF syntax is given in Appendix B. The syntax can be used to represent all types of mathematical operations. It allows the creation of expressions of any degree of complexity using numeric literals, variables, function calls or the summation operator SUM.

The table below shows the arithmetic operators allowed in EL in descending order of precedence

Operator	Description	Expression	Input type	Output type	Precedence - Associativity
**	power	expr ** expr	Numeric	Numeric	right
+	unary plus	+ expr	Numeric	Numeric	none
-	unary minus	- expr	Numeric	Numeric	none
*	multiply	expr * expr	Numeric	Numeric	left
/	divide	expr / expr	Numeric	Numeric	left
+	add	expr + expr	Numeric	Numeric	left
-	subtract	expr – expr	Numeric	Numeric	left

For each operator the table specifies which types of expressions are allowed (input type), and the type of the evaluation result (output type). For example, for addition the '+' operator takes two numeric expressions, adds them and generates a new numeric expression.

Operator precedence is very important and defines the order of evaluation of expressions. The power operator has highest precedence, then the unary + and - operators, while the addition and subtraction have the lowest. If two similar operators are adjacent, either the left- or the right-hand operator has precedence. The table shows which operator has precedence: to the left in all cases except for power (**) which is to the right. For example, the expression x** y**z is equivalent to x**(y**z), but if precedence was to the left it would have been equivalent to ((x**y)**z) which would be incorrect.

Be aware that in an expression like:

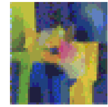
```
-2**2= -4
```

the operator ** has higher precedence than the unary minus. This is a legacy of old languages like FORTRAN. In the event that users want to consider -2 as negative, they can write:

```
(-2)**2= 4
```

For clarity, in many cases parentheses should be used. The following examples show expressions interpreted according to precedence only, and the equivalent written with parentheses:

```
a + b * c  = a + ( b * c )
a / b / c = ((a / b) / c)
a**b**c / d = (a**(b**c)) / d
```



The following are examples of valid arithmetic expressions, assuming a, b and c are numeric variables.

```
a + (b / c )
a**2 + b**3 – c**2
a+b-c 7*b/( 1 -b**2)  ((((a – 1) + 6) * 8.8e4) ** 2)
```

EL has a standard mathematics library (refer to Appendix B) for creating all kinds of complex arithmetic expressions using sins, tangents, square roots, etc.

2.7.1.1. SUM Operator

This is a special numeric operator. It is a language construct which is valid in any numeric expression. Basically it represents an algebraic sum.

$$\sum_{i=1}^{n} \exp r$$

For example, the expression:

```
dp_inertia = SUM (i IN 1,3; inertia_n[i])
```

would be equivalent to:

```
dp_inertia= inertia_n[1] + inertia_n[2] + inertia_n[3]
```

The variable does not have to be declared before it is used, as it is only in scope in the SUM block. This variable takes the values of a range, in this case 1 to 3, and sums the expression to the right i times

2.7.2. Relational Expressions

These compare operands. The operands must be valid for the corresponding comparison operator. They are used to compare expressions; the output type is always a BOOLEAN. EL provides the following relational operators:

Operator	Description	Expression	Input type	Output type	Associativity
>	greater	Expr > expr	Numeric	BOOLEAN	left
>=	greater or equal	Expr >= expr	Numeric	BOOLEAN	left
<	lower	Expr < expr	Numeric	BOOLEAN	left

<=	lower or equal	Expr <= expr	Numeric	BOOLEAN	left
==	equal	Expr == expr	Numeric, String, ENUM, Boolean	BOOLEAN	left
!=	not equal	Expr != expr	Numeric, String, ENUM, Boolean	BOOLEAN	left

The following examples are valid relational expressions:

```
7 > ( x - y)
name == "capacitor"
8.88e-3 <= getPrecision(x)
```

2.7.3. Logical Expressions

Like mathematical expressions, EL allows logical expressions.

The table below shows the logical operators provided by EL in descending order of precedence:

Operator	Description	Expression	Input type	Output type	Precedence
NOT	not operator	NOT expr	BOOLEAN	BOOLEAN	left
AND	and operator	Expr AND expr	BOOLEAN	BOOLEAN	left
OR	or operator	Expr OR expr	BOOLEAN	BOOLEAN	left

All logical expressions use and return Boolean expressions. The NOT operator has greatest precedence and OR the least. The following shows some valid logical expressions:

```
((NOT b) AND (NOT w)) OR x
(x > 9.8) AND NOT ( (n != 7) OR (m == 6) )
```

3. Basic Data Types

3.1. Concepts

All variables must be declared before use so that the compiler knows what type they are. All identifiers have an associated type, which determines the operations which may be performed on them and how those operations should be interpreted. For example:

```
INTEGER i
REAL v[3]
```

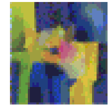
Identifier i will be associated with an integer variable; identifier v is declared as an array of three integer elements. The indexing operator [] is meaningful for v, but not for i; in other words, each identifier has an associated series of operations which are valid, and no other operations are.

There are four categories of data types in EL:

- Fundamental types such as REAL, BOOLEAN, etc. These are simple types commonly used for all types of logical and arithmetic operations
- Enumeration types, ENUM
- Derived types: arrays and SET_OF
- Complex data types: COMPONENT, PORT and CLASS. In reality these are items with a high level of abstraction which contain internal variables, equations, etc

This chapter describes the first three categories in detail; complex types are explained in later chapters. The allowed types in the first three categories are:

TYPE
NO_TYPE
REAL
INTEGER
BOOLEAN
STRING
FILEPATH
TABLE_1D, TABLE_2D, TABLE_3D



ENUM
Arrays
SET_OF
FUNC_PTR
OBJ_PTR

NO_TYPE is a special type which is only used to indicate that a function does not return a value. Variables cannot be declared as NO_TYPE.

3.2. Fundamental Data Types

EL has a series of fundamental basic types which are used both in functions and in components and ports. The table below shows these fundamental types:

Name	Comments	Value ranges
REAL	floating point (64 bits)	+/-1.7E +/- 308 (15 digits)
INTEGER	integers (32 bits)	–2,147,483,648 to 2,147,483,647
BOOLEAN	Boolean (32 bits)	TRUE, FALSE
STRING	character arrays	any ASCII string
FILEPATH	character arrays	any ASCII string (used for file paths)
TABLE_1D	one dimensional table	Only REAL values
TABLE_2D	two dimensional table	Only REAL values
TABLE_3D	three dimensional table	Only REAL values
FUNC_PTR	function pointer	
OBJ_PTR	Object pointer (to classes)	
NO_TYPE	Void returns in a function	

The variable declaration format is basically as follows[1]

```
[CONST] data_type name[arrayIndex] [= init_expression] [units]
[range] [description]
```

where:

- CONST indicates a constant (optional)
- data_type is the data type (e.g., REAL, STRING, etc)

1 For the exact format, refer to Appendix B

- name is the identifier of the variable
- arrayIndex indicates the dimensions for an array; e.g., v[6] (optional)
- init_expression is the initial value (optional)
- units are the units of this variable (optional)
- range is the valid range of values (optional)
- description is a short explanation of this variable (optional)

Examples:

```
REAL x
REAL x = 1.2387 RANGE 0.5,1.5
REAL speed= 3.34  UNITS "m/s" RANGE 0,100 "This is the aircraft
speed"
INTEGER k = 5
```

Multiple variables can be declared together, but only one initialization value, which is always associated with the last variable. For example:

```
REAL x,y = 7.8
```

Variables x and y will be initialized to 7.8. The following code is incorrect:

```
REAL x= 4, y= 5   -- Syntax error
```

This should be written as follows:

```
REAL x= 4
REAL y= 5
```

3.2.1. REAL

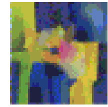
This type represents the floating-point representation of a number. REAL is equivalent to DOUBLE PRECISION in Fortran and double in C/C++.

The following are examples of declarations of the REAL numeric types:

```
REAL speed, maxLimit
REAL v[4]= { 0.23, 23.45, 87.3, .8 }
```

Allowed operations on numeric variables are:

- Assignment; e.g., speed = 0.878
- Logical comparison with the operators ==,!=, >,>=,<,<= e.g., IF(speed >= maxLimit)



- Arithmetic operations with the operators '+' (add), '-' (subtract), '*' (multiply), '/' (divide) and "**" (power); e.g., (speed * x) / (1 – e**2)

Examples of valid REAL data are:

9.344 .877 1. 1.2e10 -1.23e-8 6E3 44444444444.444e-03

3.2.2. INTEGER

The INTEGER type is used for representation of integer numeric values such as 4, -3, 0, etc. It is equivalent to INTEGER*4 in Fortran and int in C/C++.

The following are examples of declarations of the INTEGER numeric types:

```
INTEGER numberTanks= 1
INTEGER  i, j, k
INTEGER  k[4,8,2]
INTEGER  j[2] = { 0, 1 }
```

Allowed operations on numeric variables are:

- Assignment; e.g., numberTanks= 4
- Logical comparison with the operators ==,!=, >,>=,<,<= e.g., IF(numberTanks>= 5)
- Arithmetic operations with the operators '+' (add), '-' (subtract), '*' (multiply), '/' (divide) and "**" (power); e.g.,. (numberTanks* x) / (1 – e**2)
- Array index for an integer variable; e.g., v[numberTanks]

Examples of valid INTEGER data are:

9999 0 -23 1

3.2.3. BOOLEAN

The BOOLEAN type can be either TRUE or FALSE (uppercase). This corresponds to Fortran's LOGICAL*4 and bool in C++.

Examples:

```
BOOLEAN b
BOOLEAN isOpen= FALSE
BOOLEAN val[3]              "array of three Booleans"
```

Allowed operations on Boolean variables are:

- Assignment; e.g., isOpen= FALSE
- Comparisons with == and != operators: e.g., IF (isOpen == TRUE) THEN
- Logical operations with AND, OR and NOT; e.g., IF (reachedLimit AND NOT isOpen) THEN
- Internal representation of Boolean is similar to C/C++: 0 for FALSE and 1 for TRUE. Fortran requires a conversion as FALSE is 0 in Fortran, but TRUE is -1. This has to be taken into account, using a conversion function before making the call

3.2.4. STRING

STRING type allows working with characters. They are mainly used to indicate component states, comments on events, etc. For example:

```
STRING initialState= "START"
STRING finalState=   "END"
```

Allowed operations on string variables are:

- Assignment. For example

```
initialState = "END"
initialState= finalState
```

- Comparison, using the operators == and !=. For example

```
IF ( initialState == finalState ) THEN
```

Strings are also useful for entering information about variable declarations like that described before; for example:

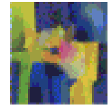
```
REAL maxAcceleration  "defines the maximum acceleration (m/s**2)"
```

3.2.5. FILEPATH

The FILEPATH data type is identical to STRING and allows the same operation to be carried out. The only difference is that if a type is defined as FILEPATH, when the object editor is used it allows the user to select a file path (relative or absolute) using an adequate browser.

The FILEPATH type will normally be used to represent the path of a file. An absolute path, such as the following, can be represented:

```
FILEPATH myFile =”C:/my programmes/files/file3455.txt”
```



The relative path of a library can also be represented with the form @LIB_NAME@. For example:

```
FILEPATH myFile ="@ELECTRICAL@/data/file45.txt"
```

There are three very useful functions for dealing with file paths:

```
STRING expandFilePath(IN FILEPATH path)
```

This function, given a relative path to a library, will return an absolute one replacing the relative path by the absolute one. For example, assuming that the ELECTRICAL library is located at D:/myLibs/ELECTRICAL, and the following is typed:

```
finalPath= expandFilePath( myFile )
```

the variable finalPath will contain the absolute path "D:/myLibs/ELECTRICAL/data/file45.txt"

```
BOOLEAN existsFile(IN FILEPATH filePath)
```

This function expands the file path and returns TRUE if the file passed to it as an argument exists, and FALSE if it does not exist.

For example:

```
INIT
      if ( existsFile( myFile ) == TRUE )
             -- the file has been found
```

```
FILEPATH checkFileExists(IN FILEPATH filePath)
```

This function is a combination of the two previous ones. The user passes a relative or absolute path to it, the function expands the path if it is a relative one and checks if the file exists. If it does not exist it emits an error message which halts the application, and if it does exist it returns the complete path.

This function is very useful and can be used when an external function has to be called that has to read an external file, and the user wants to ensure that the file exists. If it does not exist an error message will be emitted within the program as opposed, managing the error externally which is always more complicated and out of control.

For example, given an external function in C++ with a file name as an argument, it is possible to pass it the absolute path of a file after having checked that it exists:

```
 "C++" FUNCTION NO_TYPE readFile (IN FILEPATH file)

COMPONENT test
DECLS
```

```
     FILEPATH fpath= "%ELECTRICAL%/data/file45.txt"
INIT
     readFile ( checkFileExists( fpath ) )
END COMPONENT
```

3.2.6. TABLES

EL allows the use of more complex data types such as tables. These are groups of REAL data, which for given input values have an associated return value.

EcosimPro provides interpolation functions to access those tables (see Appendix B). EL can handle 1, 2 and 3 dimensional tables.

Dimension 1: These are declared with the keyword TABLE_1D. They are data tables for which one input value returns a single output, so they have the form a= table(x). For example:

```
TABLE_1D tab= { { 0,      0.5,    1},     -- X values
                {67.23,4.83, -3.23} }     -- output
```

They can be represented like this:

X	0	0.5	1
Y	67.23	4.83	-3.23

Dimension 2: These are declared with the keyword TABLE_2D. These are tables of real values, for which a pair of input values returns a single output. They have the form a= table(x,y). For example

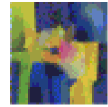
```
TABLE_2D t = {  { 1,   2},    -- X values
          { 0.9 ,1.0, 0.8 },       -- Y values
          { {4,6,7} , { 2, 3, 2 } } }-- output
```

They can be represented in a table like this:

x\y	0.9	1.0	0.8
1	4	6	7
2	2	3	2

Dimension 3. These are declared with the keyword TABLE_3D. These are tables of real values, for which a triplet of input values returns a single output. They have the form a= table(x,y,z). For example

```
TABLE_3D tbl = {  {  1.,  2. },          -- X values
                  { -1., -2., -3.},      -- Y values
                  {  4,   5 },           -- Z values
```



```
                  { {{ 1.,2.} ,{ 3.,4.} ,{ 5.,6.} } ,   -- output
                         {{ 11.,12.} , {13.,14.} , {15.,16.}}}
               }
```

They can be represented as two tables, each dependent on the value of a dimension (X for example):

for X = 1

Y\Z	4	5
-1	1	2
-2	3	4
-3	5	6

for X = 2

Y\Z	4	5
-1	11	12
-2	13	14
-3	15	16

3.2.7. FUNC_PTR

EL also allows function pointers (the only pointers it does allow). They can only be used as function arguments. This means that a function or method can declare a call argument as follows:

```
FUNCTION NO_TYPE optimize( FUNC_PTR func)
```

This means that when the function is called, the call argument must be the name of a valid function, for example:

```
optimize( evaluateMaxValue )
```

where the function evaluateMaxValue has been declared and is in scope. The argument which is passed, func in the above example, could not be used directly in EL and is only valid for use in external functions in C, C++ and Fortran. This mechanism is explained in detail in Chapter 9.

3.2.8. OBJECT_PTR

EL also allows class object pointers They can only be used as function arguments. This means that a function or method can declare a call argument as follows:

```
FUNCTION NO_TYPE optimize( OBJ_PTR obj)
```

Where obj should be a valid instance of a CLASS

3.3. Enumeration Types

Enumerative types can be defined by means of ENUM data types. They are valid for representing a set of names valid for this type. The values are valid identifiers in a comma-separated list. Once the type is defined, variables can be declared of that type, and the only allowed values are those in the enumeration list defined for the type.

For using ENUM the valid syntax is:

```
ENUM  IDENTIFIER '='  '{' enum_elem ( ',' enum_elem )* '}'
```

For example, the following defines a type to represent different chemical compounds:

```
ENUM Chemicals= {N2, H2O, CO2 }
```

The defined type “Chemicals” creates a new data type which can only take one of the three values defined for it.

Now it has been defined, variables of the “Chemicals” type can be declared. The syntax is similar to that of the fundamental types, but always preceded by the ENUM keyword. Ranges cannot be declared. For example:

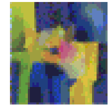
```
ENUM Chemicals v, w
CONST ENUM Chemicals water = H20
```

Variables v, w and water will be of the” Chemicals” type; water will also be a constant.

The allowed operations on these variables are:

- Assignment; e.g.,

```
v = H2O
v= w
```



- Comparison, using the operators == and !=. Note that unlike some other languages, EL does not allow greater than or less than ('>') comparisons on enumeration types; however, the following example is valid:

```
IF ( v == w ) THEN
```

- Array indexing; e.g.,

```
REAL v[Chemicals]
v[CO2]= 7.3244
```

3.4. Derived Types

3.4.1. Arrays

EL supports data arrays. An array is a collection of data items stored contiguously in memory, each accessed by its index. Arrays are dimensioned with constants or parameters of type integer, enumeration ENUM or SET_OF derived types.

Normally a one-dimensional array is called a vector, and when it has more dimensions it is called a matrix.

For an array dimensioned with an integer value the index is one-based. For example, declaring

```
REAL v[3]
```

valid indexes are v[1], v[2] and v[3].

For an array dimensioned with an enumeration type or a SET_OF derived type, the index is from the first element of the type up to the last. For example:

```
ENUM Chemicals = { CO2, H2O, O2 }
REAL v[Chemicals]
```

Valid indexes are then [CO2], v[H2O] and v[02].

EL supports multi-dimensional arrays. Dimensions are written as a comma-separated list. For example, REAL v[3,4,Chemicals] declares a three-dimensional array (matrix). The first dimension has three elements, the second has four and the third has three (the number of elements in the enumeration type Chemicals). There is no limit on the number of dimensions. Valid indexes are v[2,4,H2O] and v[1,2,O2].

Array elements can be of types REAL, INTEGER, BOOLEAN, STRING and ENUM.

The general syntax is:

```
array_decl::= IDENTIFIER( '['expression_s']' ) ? ( '='init_expression )?
expression_s    ::=  expression ( ',' expression )*
init_expression ::=  '{' init_expression_s '}' | expression
init_expression_s::= init_expression ( ',' init_expression )*
```

Where IDENTIFIER is the name of the array, expression is any integer expression valid for initializing expressions and init_expression is the initialization of the array if required.

The syntax above shows how arrays can be initialized when they are declared.

The following are examples of different types of valid arrays:

```
REAL v[2,3]
INTEGER i[2,3,2]
BOOLEAN b[2]
ENUM Chemicals v[2]
ENUM Chemicals w[Chemicals]
STRING Colors[3]
```

These are the same examples, but initialized:

```
REAL v[2,3]= { {1,2,3},{ 4,5,6 } }
INTEGER  i[2,3,2]= { {
{1,2},{3,4},{5,6}},{{7,8},{9,10},{10,11}}}
BOOLEAN b[2]= { FALSE, TRUE }
ENUM Chemicals v[2]= { H2O, H2O }
ENUM Chemicals  w[Chemicals]= {CO2, H2O, CO2}
STRING colors[3]= { "red", "white", "black" }
```

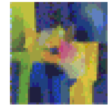
All array dimensions must be known beforehand, although they may be dimensioned using variable expressions. The only restriction is that all the model data must be known when the final model is generated. Later on, we describe the use of construction parameters to define dimensions of components and ports, and show the use of construction parameters to dimension arrays at compile time. This is a very powerful feature of EL. The user can delay the value of the dimension at the time the component is used.

There is an alternative way of initializing an array, in case the user wants to initialize all elements with the same value:

```
REAL v[2] = 2.3
```

This is equivalent to writing:

```
REAL v[2] = { 2.3, 2.3 }
```



This is very useful for initializing arrays of unknown dimensions when they are declared, for example:

```
COMPONENT foo (INTEGER N)
DATA
     REAL v[N]  = 0
```

In this example, when declaring array v the dimension is unknown (N), but the modeller can provide an initial value (0) to all elements.

3.4.2. SET_OF Dynamic Enumeration Types

The SET_OF is very useful for simulating chemical components. If this is not your area of work, you may prefer to skip this section. This description is brief, as the true power of this type is covered later on where the use of construction parameters to define components and ports is described.

SET_OF types are another derived type, based on enumeration types. They are similar to ENUM and have only two allowed operations. They are mainly used as construction parameters passed to components and ports. As the name suggests, they are sets of elements, but in EL they are always sets of enumeration types. They are a kind of ENUM but their elements can vary dynamically. They are mainly useful for dimensioning arrays dynamically. For example, when a chemical component works with different compounds, sets are used as an input construction parameter to the component and are also used to dimension arrays and evaluate ranges. SET_OF types are only useful as construction parameters of components and ports.

The syntax to define a set of enumeration types is:

```
setof::=SET_OF(enum-id)IDENTIFIER '=''{' enum_elem( ','
enum_elem )* '}'
```

Where **enum-id** is the enumeration type it supports, **IDENTIFIER** is the name of the set, and **enum_elem** are the different values separated by commas.

A set can only be defined in the DECLS block of a component or as a global in a library. For example:

```
ENUM Chemicals = {N2, H2O, CO2, N2, O2, H2SO4 }
SET_OF(Chemicals) Air = {N2, O2, H2O, CO2}
SET_OF(Chemicals) Water = {H2O}
```

The air set contains four elements of the Chemicals type and the water set has one element.

As already mentioned, they are mainly used as construction parameters to components and ports. The syntax for that is:

```
SET_OF( enum-id ) IDENTIFIER
```

Although the explanation of how components are defined is given later, the example below shows the use of SET_OF as a construction parameter to a component:

```
COMPONENT composer (SET_OF(Chemicals) mix)
DECLS
     REAL v[mix]
```

The size of v depends on the number of members of the set mix.

Enumeration set types have only very limited operations and are normally used for:

- Dimensioning arrays. An array is dimensioned to the number of members of the set and the elements are accessed by their index, which is a member of the set. If mix contains 2 elements, r will have a dimension of 2. Its elements can then be accessed using valid indexes; e.g., r[CO2]

```
DECLS
REAL r[mix]
```

- To generate index ranges. The IN operator can be used to iterate all the members of a set. In the following example, if mix contains the members O2 and H2O, the counter i will take those values and perform the programmed operation:

```
FOR( i IN mix)
     r[i] = 0.0
END FOR
```

equivalent to r[O2]= 0.0 and r[H2O]= 0.0

3.4.2.1. Accessing SET_OF elements

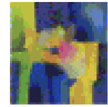
EL provides several useful functions when the modeller wants to refer to specific elements inside a SET_OF by the position of an element inside the SET_OF.

The function "setofElem(SET_OF(t) set,int n)" is used to return the nth element inside the SET_OF set. For instance, given the next SET_OF:

```
SET_OF(Chemicals) Mix = {N2, O2, H2O}
```

the statement

```
setofElem(mix,1))
```



will return the element N2.

The function "setofPos(SET_OF(t) set,ENUM t elem)" returns the position of element "elem" inside the SET_OF "set". If it returns 0, it means that it has not been found in "set". For instance:

```
setofPos(mix, H2O)
```

returns 3. If we write

```
seofPos(mix,NO2)
```

it returns 0 since it is not found in "mix".

The function "setofSize(SET_OF(t) set)" returns the number of elements in "set". For instance

```
setofSize(mix)
```

would return 3.

3.4.3. SET_OF Default Values

As in the case of construction parameters of the INTEGER type, BOOLEAN type, etc, the SET_OF values also admit default values when declared. For example:

```
ENUM Chemicals = { CO2, H2O, O2}
SET_OF(Chemicals) mset = {O2}



COMPONENT Foo (SET_OF(Chemicals) mix = mset)
DATA
      REAL v[mix]
END COMPONENT
```

When this component is used within another one, initialization of the “mix” parameter is optional. If we initialize it, it uses the new SET_OF value; if not, it will use the default value. For example:

```
COMPONENT Final
DECLS
    SET_OF(Chemicals) setBis = {CO2, H2O}
TOPOLOGY
   -- using the defined in this component
   Foo (mix = setBis) foo1
```

```
            -- using the default mset
foo foo2
END COMPONENT
```

3.5. Characteristics of Variables

3.5.1. Constants

EL allows the use of constants. When a variable is declared with the qualifier CONST, this indicates that it should remain constant.

As constants cannot be changed, they must be initialized when they are declared.

To declare a constant, simply precede the declaration with the keyword CONST. For example:

```
CONST REAL PI= 3.141592
CONST INTEGER  numberOfValves= 5
```

The following would be illegal:

```
CONST REAL PI    -- ERROR, it needs initialization !
```

3.5.2. Units

When a variable is declared, the user may specify the units worked by using the reserved work UNITS followed by a string. For example:

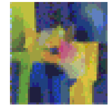
```
   REAL speed        UNITS "m/s"
   REAL acceleration UNITS "m/s**2"
```

It is important to note that the string representing the units is taken literally as a comment so it is the modeller's responsibility to ensure that it is correct.

This units string is used later by different editors and in reports, etc, and provides valuable information to users.

It is possible to create STRING type constants and then use them as units. For example:

```
CONST STRING u_speed= "m/s"
```



```
CONST STRING u_accel= "m/s**2"

REAL speed        UNITS u_speed
REAL acceleration UNITS u_accel
```

In this way the constant can be used each time the units have to be written.

3.5.3. Description of Variables

We recommend using the description of variables to improve the readability of the code. It is simple to do; just add a comment like a string after the declaration, for example:

```
REAL speed UNITS "m/s" "This is the aircraft speed"
```

This description is used later in any report, wizard, etc, to help the user to understand the models.

3.5.4. Variable Range

For numeric variables (REAL or INTEGER) of components or ports, EL allows you to define a range of validity. During simulation, if the value of any variable exceeds the defined limits, EcosimPro generates a warning. This range is defined using the RANGE keyword. For example:

```
REAL v RANGE 0.5,1.5
```

During the simulation, each time the variable v goes out of interval [0.5,1.5] EcosimPro generates a warning. This is an extremely useful capability to check the integrity of the simulation at any time. On the other hand, modellers must be aware that it introduces a small simulation delay time to check it.

3.5.5. Scope of Variables

A declaration introduces an identifier in a scope. In other words, an identifier can only be used in a specific part of a program. For example, if a variable is declared in a function, that variable is only in scope in the BODY of that function, but nowhere else; it has local scope. An identifier declared outside a function, component or port in a library can be used from anywhere in that library or from outside it by specifying the library name; it has global scope.

By default, a global element (component, port, etc) is in scope to the other items present in the working library and in the system library SYSTEM_MDL_LIB.

EL has three levels of scope:

- Local scope. An identifier declared inside a component, port, function or experiment. It is referenced using the name of the variable or constant
- Global scope. The identifier has been declared global inside a library. It is referenced using the name of the variable, but if the name is declared more than once (for example, if two libraries use the same global variable name) the library name must be prefixed:

```
LIB_NAME.IDENTIFIER
```

- Expression scope. An identifier used in a SUM, EXPAND or certain FOR expressions is only in scope in that expression. Once the expression has been evaluated the name is no longer valid. This allows successive expressions to use the same identifier. If an internal identifier conflicts with one with the same name in another component, the internal one is always used. The following are valid examples:

```
EXPAND(i IN 1,3)
     v[i]= 0.0
  --equivalent to v[1]= 0.0 v[2]= 0.0 v[3]= 0.0
x= SUM(i IN 1,2; inertia[i])
  --equivalent to x = inertia[1] + inertia[2]
FOR(i IN Chemical)
     v[i]= 0.0
  -- equivalent to v[CO2]= 0.0 v[H2O]= 0.0
```

The USE statement enables you to reference items in other libraries. The clauses in the following example allow references to items in the FLUID and CONTROL libraries:

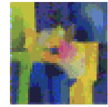
```
USE FLUID
USE CONTROL
```

If an item is used which is present in more than one library in scope, you must specify which is to be used. For example, to use the pump component in the ECLSS library when the FLUID library is in scope, which also contains a pump component, prefix the library name as follows:

```
ECLSS.pump
```

By the language definition, when there is a naming conflict in a scope, the nearest variable is always used. For example, if you declare a variable in a component with a name that already exists with global scope, the local is used. For example, in the following equation:

```
REAL x                    -- global variable
COMPONENT test
```



```
DECLS
     REAL v, x          -- local variables
CONTINUOUS
     v= x  + 5          -- equation
END COMPONENT
```

In the above equation, the component's local variable is used. To use the global, the library name is prefixed to the variable name. For example, if the variable is in MATH, the equation becomes:

```
v= MATH.x + 5
```

and the global variable is used. In general this is not recommended practice because it is prone to errors.

Function arguments are considered to be valid variable declarations. The following code would generate an error for redefinition of the variable x:

```
FUNCTION NO_TYPE test(REAL x)
DECLS
       REAL x            -- ERROR, x is redefined
```

However, it is valid to initialize a variable, prior to the variable that causes the redefinition, with a global constant of the same name. For example:

```
CONST REAL x= 5
FUNCTION NO_TYPE test ()
DECLS
     REAL v= x       -- OK!, equivalent to v= MATH_LIB.x
     REAL x          -- definition of local x
BODY
     x= 8 -- it refers to local x
```

But it is not valid to initialize a variable with a constant of the same name in the example above.

```
DECLS
     REAL x= x     -- ERROR ! not allowed
```

But the following would be valid:

```
DECLS
     REAL x= MATH.x     -- OK!
```

By default, the results of an EL compilation are stored in the user's current working library. To use a different library for the compiled results, include the following directive in the file header:

```
LIBRARY libName
```

Where **libName** is the name of the library.

3.5.6. Global Variables and Types

Any variable or enumeration type declared outside of a component, port or function is considered global to the working library. This means that it can be used in any component in the library in which it is stored. The syntax for declaring variables and constants as global enumeration types is the same as that shown before.

Global declarations are often used to define enumeration types and constants which are shared. For example, in chemical modelling it is normal to have an enumeration type which contains the chemical compounds. For example:

```
ENUM Chemicals= { CO2, H2O, O2 }
```

It is also common to define shared global constants:

```
CONST REAL PI= 3.141592
CONST REAL G= 9.8
```

3.6. TIME, TSTOP and CINT Global Variables

Although EL is a modelling language for expressing algebraic-differential equations, the independent variable is always time. Any derivative defined is understood to be of the type:

```
x' = dx / dt
```

This variable t in EL is identified as TIME (uppercase) and is accessible from all components, ports, experiments and experiment functions.

The variable TSTOP(uppercase) is also in scope. It is needed to indicate the time at which integration should stop.

The variable CINT(uppercase) is used to indicate the interval to produce output results of simulation. For instance if CINT = 1, EcosimPro will generate an output every second (if second is the unit used for the independent variable TIME).

4. Libraries

Libraries are a mechanism used to organize design information and to provide other functionalities such as name encapsulation and obsolescence control. They are a natural way to group components, functions, etc, which are related to a discipline. For instance, resistors, capacitors, etc, should be stored in an electrical library.

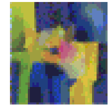
EcosimPro provides a very powerful library mechanism for handling different kinds of components, etc, while maintaining a simple syntax to express it.

4.1. What is an EL Library?

A library in EL is a set of items. The items could be components, port types, global variables, classes and functions which are related to the same area. For example, EcosimPro provides demo libraries for CONTROL, FLUID and ELECTRIC disciplines. Each of these libraries can contain:

- Components
- Port types; e.g., fluid, electric, thermal, etc
- Enumeration types (ENUM and SET_OF)
- Functions
- Classes
- External pre-declaration functions (FORTRAN, C and C++)
- Global variables (REALs, INTEGERs, BOOLEANs, STRINGs, tables and multi-dimensional arrays)
- USE statements (to reuse components from other libraries)

For instance, the ELECTRIC library contains components like resistor, capacitor, diode, etc, port types like the electrical port "elec" and global variables like "Troom" and "Tref".



4.2. Library versions

The libraries have a mechanism whereby it is possible to tag them with a version number. It is a special structure called #LIBRARY_INFO, and consists of creating an EL file for each library as follows:

```
#LIBRARY_INFO
     #VERSION = "1.0"
     #DATE = "12/02/2009"
     #AUTHOR= "None"
     #COMMENTS= "no comments"
END #LIBRARY_INFO
```

The following can then be indicated within the structure:

- Current version of the library with the format "MAV.MIV(.UPV)", where MAV (Major Version), MIV (Minor Version) and UPV (Upgrade Version) are positive integers. The major and minor versions are mandatory in any case. For example, the following are valid version numbers: "1.2", "3.4.55", "4456.22" and"0.0", whereas the following are not valid: "-3.1", "45", "A.1" and "1.1.5.3"
- Date of creation. A string in free format
- Author. A string in free format
- Comments. A string in free format. If more than one line is required, the symbol "\" has to be placed at the end of each line. For example:

```
#LIBRARY_INFO
     #VERSION = "1.0"
     #DATE = "12/02/2009"
     #AUTHOR= "None"
     #COMMENTS= "This is the first \
               Version of the Programme, \
                Ready to use by external users"
END #LIBRARY_INFO
```

This information will be used at different points within the program.

- Library documentation. Each time the generation of the library documentation is requested, this information will be printed.
- Partition documentation, reports, etc. Information is given on the library version when they are generated.
- For conditional compilations with the USE statement (see next point).

4.3. Using Libraries in EL

By default EcosimPro will compile all items of a file into the library under which the file is located. In other words, if the user has a file named myFile.el under ELECTRIC\sources directory, the components in the file will be added to the ELECTRIC library.

If they are items from other libraries, the user must predeclare their usage with the statement USE such as:

```
USE CONTROL
COMPONNET foo
TOPOLOGY
    Cntrl_PID pid    -- from CONTROL library
END COMPONENT
```

The USE clause advises that the library CONTROL must be loaded because some code will make reference to components (or other items) from that library. In this case it uses the CONTROL.Cntrl_PID component.

If the name of the component may be repeated both in the CONTROL and the current library, the user must write LIB_NAME.ITEM_NAME to define the exact item to be used. For example:

```
USE CONTROL
COMPONNET foo
TOPOLOGY
    CONTROL.Cntrl_PID pid    -- from CONTROL library
END COMPONENT
```

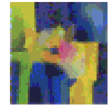
The USE statement allows the user to specifiy a required library version. For example:

```
USE CONTROL VERSION "3.4"
```

In this case, when the file is compiled, the program will check if the current CONTROL library is version “3.4” or compatible. If it is not, an error code will be generated and compilation halted.

Compatibility between versions is defined as follows:

- If the major version and the minor version of the current CONTROL library matches the required one (“3.4” in the previous example)
- If the major version of the current CONTROL library matches the required one (e.g., 3) and the minor version of the current one is greater than or equal to the minor version of the required (e.g., >=4)
- Any other combination is not compatible and an error code is generated



For example, for the previous USE it would be compatible if the version of the CONTROL library is “3.4”, “3.4.1”, “3.4.32”, “3.5”, “3.12.34”, etc. However, it would not be compatible if it were “2.1”, “3.3”, “3.3.4”,”4.0”, “12.34”, etc.

In the event of incompatibility, it is the responsibility of the modeller to check if the components are still compatible and update the version in the USE statement.

4.4. Forcing compilation into a library

In some cases the user can force the library to be used as a depositary for the items instead of the current library. This is done by means of the LIBRARY statement. For instance, suppose we have a file named myFile.el under the sources directory of the CONTROL library, if the user writes:

```
LIBRARY ELECTRIC
CONST REAL PI = 3.141592
COMPONENT test1
END COMPONENT
```

the items PI and test1 will not be added to the database of the CONTROL library (as expected) but to the ELECTRIC library. In general, this practice is not recommended because the programming will lose clarity since some files of a library can modify the database of a different library. This only has real application when programming outside the graphical user interface which is beyond the scope of this manual.

4.5. Manipulation of Libraries

EcosimPro allows the user to manipulate the libraries by creating new ones, deactivating, etc. This is explained in the User Manual.

4.6. System Libraries

The modelling environment provides two special libraries not visible to the developer: SYSTEM_MDL_LIB and SYSTEM_EXP_LIB. These contain special language items included by default. They cannot be modified.

SYSTEM_MDL_LIB contains the items needed at all times such as the variables TIME, TSTOP, CINT, and the standard mathematical functions such as sin(), cos(), etc. An appendix at the end contains a description of all the default variables and functions. The

functions in this library can be used at any time, in components, ports and experiments. The variables have a more restricted scope.

SYSTEM_EXP_LIB contains items useful in experiments, such as INTEG, REPORT_LIST, etc, as these variables and functions are only used in experiments and function experiments.

See Appendix B for a full description of these libraries.

4.7. Automatic Documentation

EcosimPro provides a utility from the Library tab to automatically generate the documentation for a library. The generated document can be in HTML or in ASCII format. This utility is explained in the User Manual.

In order to provide as much information as possible for documenting the library, the language offers the capability to include a string explanation for each function, component, port and variable declaration in the language, as explained in previous sections. For instance

```
COMPONENT Pipe "Describes a pipe"
END COMPONENT

REAL  length           "length of pipe (mts)"
```

This format (inserting the units in parentheses) allows EL to extract the information when automatically generating the documentation for a library. These strings are allowed in:

- Components
- Ports
- Functions
- Classes
- Port and component variable declarations
- Ports used by the components
- Function variables

If HTML format is chosen, the user can add further HTML files to the DOCS directory so that they are automatically extracted and added to the file generated. These back-up files are basically of three different types:

- General documentation of a library. A file called LIBRARY_doc.htm should be created in the DOCS library which contains a general description of the library including, for example, an introduction, units used, conventions, hierarchical diagrams of components, etc. For example, for the CONTROL library the file would be called CONTROL_doc.htm
- Description of a component or port. An HTML file would be created in the DOCS directory which contains the general description of a component. The file will be called by the convention COMPONENT_doc.htm, where COMPONENT will be the name of the component; for example, Turbine_doc.htm
- Description of the formulation of a component, port or function. This HTML file would normally describe the formulation of the equations or sentences which form the component. The back-up file will be in the DOCS directory and it will be given a name based on the format COMPONENT_for.htm; for example, Turbine_for.htm

For example, given the electrical capacitor, the formulation in EL is:

```
------------------------------------------------------
PORT Elec "electrical port"
       SUM REAL c       "current (Amperes)"
       EQUAL REAL v     "voltage (Volts)"
END PORT

------------------------------------------------------
ABSTRACT COMPONENT TwoPins "Abstract component with an input and an output"
    PORTS
        IN  Elec e_p     "Input port"
        OUT Elec e_n     "Output port"
    DECLS
        REAL v     "voltage difference (Volts)"
    TOPOLOGY
        PATH e_p TO e_n
    CONTINUOUS
        e_p.i = e_n.i
        v = e_p.v - e_n.v
END COMPONENT
------------------------------------------------------
COMPONENT C IS_A TwoPins "Electrical capacitor"
    DATA
        REAL C= 1.e-7  RANGE 0, Inf   "capacity at refer. temp. (Farads)"
        REAL TC1 = 0.                "coeficient for linear influence of temp. on
capacity(Farads/C)"
        REAL TC2 = 0.                "coefficient for linear influence of temp. on
capacity(Farads/C)"

    DECLS
        REAL DTemp                   "temperature deviation (C)"
        REAL Cval      RANGE 0, Inf  "capacity (Farads)"
CONTINUOUS
        DTemp = Troom - Tref
        Cval = C + DTemp * TC1 + DTemp * TC2**2
        v'= e_p.i / Cval                     --voltage derivative
END COMPONENT
```

The documentation generated automatically for component "C" is:

```
COMPONENT: C : INHERITED FROM COMPONENT TwoPins
DESCRIPTION
Electrical capacitor
```

PORTS:

NAME	TYPE	CONSTRUCTION PARAMETERS	DIRECTION	DESCRIPTION
e_p	Elec		IN	Input port
e_n	Elec		OUT	Output port

DATA:

NAME	TYPE	INITIAL	RANGE	DESCRIPTION	UNITS
C	REAL	1E-007	0,Inf	capacity at refer. temp.	Farads
TC1	REAL	0		coefficient for linear influence of temp. on capacity	Farads/C
TC2	REAL	0		coefficient for linear influence of temp. on capacity	Farads/C

DECLS:

NAME	TYPE	INITIAL	RANGE	DESCRIPTION	UNITS
v	REAL			Voltage difference	Volts
Dtemp	REAL			temperature deviation	C
Cval	REAL		0,Inf	capacity	Farads

5. Statements

Three different types of statements can be used in EL, depending on the context: sequential, continuous or discrete. Classical languages like FORTRAN and C++ only allow sequential statements. Other event-oriented modelling languages only allow discrete statements, while yet other equation system solving languages only allow continuous statements. EL allows all three because even though EcosimPro's main purpose is continuous modelling, it can also be used to simulate discrete components and to execute sequential code. The use of the three types of statements may be summarised as follows:

- Sequential statements are used for initializations, functions and discrete events bodies which require a strict execution order
- Continuous statements are used to express sets of differential-algebraic equations, where the order in which they are written is not important, as they will be sorted later by the internal algorithms. They form the core where the continuous physical models are defined
- Discrete statements are used to express events. These events are controlled by conditional statements which indicate when an event occurs

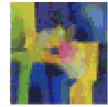
Each one is described below in detail, although the continuous and discrete ones can only be completely understood when they are fully explained in the sections dealing with ports and components.

5.1. Sequential Statements

Sequential statements are executed progressively and allow you to control the flow of the program. The flow can go down a branch if a condition is met, go into a conditional loop, call a function, etc.

The permitted sequential statements are the typical ones found in programming languages such as C++, FORTRAN, ADA, etc, and consist of assignments, iterative loops and conditional statements. In EL, these statements can be used in all the sequential parts; ie:

- BODY block for functions
- BODY block for experiments



- INIT block for components
- Discrete WHEN statement actions for components

5.1.1. Assignments

These are the most simple statements in the language, and basically consist of the assignment of an expression to a pre-declared variable. The variable must be of the same type as the expression (numeric, Boolean, etc). For example:

```
x= 6
x= y + 5.66
x= z / (maximum(z,y) * 0.45)
```

The left-hand side must always be a valid variable (it can be an array element), and the right-hand side can be any expression of a compatible type. The general syntax is:

```
assign_stms_s ::= IDENTIFIER '=' expression (AFTER expression)? |
      IDENTIFIER "+=" expression (AFTER expression)?  |
      IDENTIFIER "-=" expression (AFTER expression)?
```

The operators used are: '=' direct assignment

'+=' increment by expression

'-=' decrement by expression

For example:

```
x = 8
x += 8   -- equivalent to x = x + 8
x -= 8    -- equivalent x = x - 8
```

For components (not for functions), delayed assignments are possible by using the AFTER statement. This means that when the statement is executed, the assignment will become effective after the delay you specify. For example:

```
a= b * c AFTER 2.5
```

This assignment statement specifies that the value of "a" shall be equal to the expression "b*c" when 2.5 seconds have passed from the current time; ie, if the current time is 10 seconds, the assignment will become effective when the time is 12.5 seconds.

Complex expressions with calls to functions which return values or which change the values of the variables given as arguments are also allowed. Below are some examples

of assignments, where a, b, c and d are real variables, v is an array of reals, and e, f, g and h are Booleans:

```
a= 8.543534e8
a= v[0] + (b**2 - c**3)
a= fun(b,c,d)
a += b - 1 - fun(b,c,d) AFTER 7 + 8.34
a= b + min(c,d)
e= f AND NOT (a > 0 ) OR (b <= 10.8)
e= TRUE
```

5.1.2. Function Calls

EL also allows function calls as sequential instructions. For example:

```
evaluateParameters(x,y)
max(x,y)
```

Functions in EL behave in a similar manner to other languages which use functions; ie,. the function is executed and control then returns to the next instruction. In the functions section it will be explained how the arguments of the function can be input or input-output, with which it is possible to modify arguments in the function call during its execution.

You can call EL functions and external functions in C, C++ or FORTRAN.

5.1.3. Special Functions for Printing Messages

5.1.3.1. WRITE(), WRITEF and WRITES() Functions

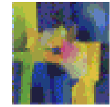
The WRITE(), WRITEF() and WRITES() functions are a family of functions to print strings in the standard output into a file or into a STRING variable. These functions allow you to format the output strings with the desired format. They are similar to the printf(), fprintf() and sprintf() of C language and they use the same terminology for specifiying the output format.

The function signatures are as follows:

- Written directly on the screen (similar to printf() in C)

```
WRITE  (CONST STRING format, arg1, arg2,...,argN)
```

- Written to a file (similar to fprintf() in C)



```
WRITEF(CONST STRING fileName, CONST STRING format, arg1, arg2,…,argN)
```

- Written in a STRING variable (similar to sprintf() in C)

```
WRITES(STRING strName, STRING format, arg1, arg2,…,argN)
```

Where:

Format: The format argument consists of ordinary characters, escape sequences, and (if arguments follow format) format specifications. The ordinary characters and escape sequences are copied to the standard output in the order of their appearance. For example “The final value of x is %f\n”. The valid arguments are described in an appendix to this document.

Arg1, arg2,...,argN: Variable list of arguments. They indicate the variables or formulae to be printed.

- fileName: Name of file into which data must be written
- strName: Name of the STRING variable whose value is to be modified

The WRITE function formats and prints a series of characters and values to the standard output. If arguments follow the format string, the format string must contain specifications which determine the output format for the arguments. WRITEF behaves identically except that WRITE writes output to standard output rather than to a destination FILE whose name is fileName. WRITES writes output to an EL STRING variable.

Format specifications always begin with a percent sign (%) and are read left to right. When WRITE (or WRITEF or WRITES) encounters the first format specification (if any), it converts the value of the first argument after format and outputs it accordingly. The second format specification causes the second argument to be converted and output, and so on. If there are more arguments than there are format specifications, the extra arguments are ignored. The results are undefined if there are not enough arguments for all the format specifications.

For example, from the following experiment:

```
EXPERIMENT exp1 ON foo1.default
DECLS
       INTEGER count = -9234
       STRING str= "computer"
       REAL fp =  251.7366
BODY

   // Display REAL numbers.
   WRITE("Real numbers:\n\t %f\tRegular format\n\t %.2f\tTwo decimals\n\t %E\tExponential
format\n", fp, fp, fp)

   // Display integers
   WRITE("Integer formats:\n\tDecimal: %d  Justified: %.6d \n",count, count, count)

   WRITE("Decimal %d as:\n\tHexadecimal: 0x%Xh  Octal: 0%o\n",count, count, count)
```

```
   // Display in different radixes
  // WRITE("Digits 10 equal:\n\tHex: %i  Octal: %i  Decimal: %i\n",0x10, 010, 10)

   // Display strings
   WRITE("Strings justified:\n%25s\n%25.4hs\n\n",str, str)

END EXPERIMENT
```

The following output will be obtained:

```
Real numbers:
     251.736600     Regular format
     251.74         Two decimals
     2.517366E+002  Exponential format
Integer formats:
     Decimal: -9234  Justified: -009234
Decimal -9234 as:
     Hexadecimal: 0xFFFFDBEEh  Octal: 037777755756
Strings justified:
                 Computer
                     Comp
```

The use of WRITEF() is the same as WRITE(), except that as an initial argument we state the name of the destination file to which the output is to be saved. For example:

```
WRITEF("myFile.rpt","An example\n")
```

The name of the file may also be a constant of the CONST STRING type. For example:

```
CONST STRING fname = "myFile.rpt"
...
WRITEF(fname,"An example\n")
```

In addition, EcosimPro has the following routines to extract names from the runtime variables and the BOOLEAN and ENUM values:

```
"EL" FUNCTION STRING gnameReal(REAL x)
"EL" FUNCTION STRING gnameInt(INTEGER x)
"EL" FUNCTION STRING gnameBool(BOOLEAN x)
"EL" FUNCTION STRING gnameString(STRING x)
"EL" FUNCTION STRING gnameEnum(ENUM EnumType x)
"EL" FUNCTION STRING gvalBool(BOOLEAN x)
"EL" FUNCTION STRING gvalEnum(ENUM EnumType x)
"EL" FUNCTION STRING gvalEnum2(STRING enumType, ENUM EnumType x)
"EL" FUNCTION STRING gvalString(STRING x)
```

The gname… functions are used to extract the name of the variable. Assuming that the pressure is a REAL variable and v is an INTEGER variable inside a component, we can write:

```
WRITE("The value of %s is %f and %s is %d\n",
gnameReal(pressure) ,pressure,gnameInt(v),v)
```

Let us assume that the variables belong to a component called "subs1" and that this, in turn, belongs to another called "compo1". The printout could be as follows:

```
The value of compo1.subs1.pressure is 5.3445 and compo1.subs1.v is 34
```

To print BOOLEAN or ENUM values, special functions must be used beginning with gval.... The functions always return a STRING which can be printed like any other string. For example, with the following statements:

```
BOOLEAN v= FALSE
ENUM liquids = { water, methane, ammonia }
ENUM liquids liq= ammonia
```

and the following sentences

```
WRITE("Boolean example: %s = %s\n", gnameBool(v), gvalBool(v))
WRITE("ENUM example: %s = %s\n", gnameEnum(liq), gvalEnum(liq))
```

the following output is produced:

```
Boolean example: compo1.subs1.v = FALSE
ENUM example:  compo1.subs1.liq = ammonia
```

The function gvalEnum2 should be used inside functions since the function gvalEnum only works inside components. The first argument of gvalEnum2 is a string with the name of the enumerative type, for instance:

```
WRITE("ENUM example: %s = %s\n", gnameEnum(liq), gvalEnum2 ("liquids",liq))
```

Use of the WRITES() function is similar to that of WRITEF(), but instead of writing to a destination file it writes to a STRING variable. For example:

```
COMPONENT test
DECLS
     REAL x= 9
     STRING st
INIT
     WRITES(st,"It is a trial and includes x= %f",x)
END COMPONENT
```

This example shows how to initialize the value of the STRING variable st to the value "It is a trial and includes x= 9.0000".

5.1.3.2. *PRINT() and PRINTF() functions*

The functions PRINT() and PRINTF() are similar to the WRITE() family but they do not allow you to format the output strings. They should be used only in certain cases where the output format is not important; normally the WRITE() family should be used since they are more powerful.

The EL format of these functions is:

```
PRINT  (CONST STRING format)
PRINTF (CONST STRING fileName, CONST STRING format)
```

Some examples:

```
PRINT("Reached top high")
```

produce a message like this:

```
<TIME: 6.34214> Reached top high
```

The PRINT family automatically introduces two things:

- The current TIME
- A newline character at the end of every call

The string on the right is the message. Two special characters can be used inside the string to obtain more metainformation:

- The '%' character before a variable name expands the full name of the variable when it is printed
- The '$' character before a valid variable name prints the value of this variable at the time of printing

For example, if you type the following:

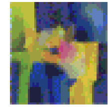
```
PRINT("variable %voltage is $voltage")
```

the following could be printed, depending on the position of the statement:

```
<TIME: 6.342> variable b1.resistor3.p_in.voltage is 3.45525
```

You can also first create the format in a generic string constant and use is like this:

```
CONST STRING format= "variable %voltage is $voltage"
...
PRINT( format )
```



Remember that if the variable name is not found, an error will be displayed at runtime, not at compile time, as the value is searched for dynamically.

Note: If you call the function from an experiment, you have to specify the variable names completely, as it does not make sense to use the '%' character for incomplete names. However, you can still use the '$' character to extract the value of a variable. For example:

```
PRINT( "b1.resistor3.p_in.voltage is $b1.resistor3.p_in.voltage")
```

There is another special character which allows the same PRINT command to continue on the next line; that character is '\'. For instance

```
PRINT ("The value of %speed is $speed \
         and the value of %high is $high")
```

You should avoid the most common errors with this capability:

- ☐ Be sure there are no extra characters after the character '\'
- ☐ The next line must contain the continuation of the string; ie, comments, empty lines, etc, are not allowed

For instance:

```
PRINT("The value of %speed is $speed \ -- first part
 and the value of %high is $high")     --ERROR

PRINT("The value of %speed is $speed \

and the value of %high is $high")        --ERROR
```

The function PRINTF() is identical to PRINT() except that you also specify a file name as the first argument and it doesn't print the current time The file name is where the message will be printed. The syntax is as follows:

```
PRINTF( STRING fileName, STRING format)
```

For example, the following call:

```
PRINTF("miFile","Variable %temp is $temp")
```

would print in the file "myFile.rpt" the corresponding message and the current time:

```
Variable c1.temp is 6.334
```

assuming that the variable is in the "c1" instance.

The file name can take a constant value like:

```
CONST STRING file = "miFile.txt"
```

and can then be used as follows:

```
PRINTF( file, "Variable %temp is $temp")
```

As before, you can also create a generic format:

```
CONST STRING format= "Variable %temp is $temp"
...
PRINTF( file, format )
```

5.1.4. IF-THEN-ELSE Statement

EL has the classic IF-THEN-ELSE programming structure. Its purpose is to select for execution one of a sequence of statements, depending on the Boolean value of one or more conditions. The syntax is as follows:

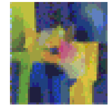
```
IF condition THEN
     seq_stm_s
(ELSEIF condition THEN seq_stm_s)*
( ELSE seq_stm_s )?
END IF
```

where **condition** is the logical entry condition, and **seq_stm_s** are the sequential instructions that are executed. You can define as zero, one or many ELSEIF blocks and one optional ELSE block.

IF or ELSEIF statements can contain nested IF statements. In the execution of an IF statement the conditions are evaluated sequentially and when one evaluates to TRUE the associated statements are executed.

For example:

```
IF (signal == SIN) THEN
    value =  sin (2 * 3.14159 * x)
ELSEIF (signal == SQUARE) THEN
    value = isqr
ELSE
    value = 1.
END IF
```



5.1.5. WHILE Statement

In a WHILE statement, a block of statements is executed for as long as the entry condition evaluates to TRUE. The syntax is as follows:

```
while_stm        ::= WHILE condition seq_stm_s END WHILE
```

The entry condition is expressed in the logical expression "condition" (which can optionally be placed within parentheses). The statements to be executed are the sequential instructions "seq_stm_s".

Examples:

```
WHILE  index < maxIndex
     b[index]= 0.0
     index= index + 1
END WHILE
WHILE ( a > 10 AND NOT flagIn )
     w[index] += 0.1
     index += 1
END WHILE
```

5.1.6. FOR Statement

The FOR statement represents a compact way of expressing loops. There are two types: the first is similar to the FOR loop in C++ and the second is a powerful way of making a variable iterate within a range of values. The syntax is as follows:

```
FOR  '(' seq_for_schema  ')'  seq_stm_s END FOR
seq_for_schema  ::=
  assign_stm_s? ';' expression ';' assign_stm_s? |
  IDENTIFIER IN range1 ( EXCEPT expression_s )?
```

In the first version you have to declare the variables in the actual FOR block. This iterative loop has an initial assignment, a final condition and an intermediate assignment. Below are some examples of the first type (similar to C++):

Declaration of variables

```
DECLS
     INTEGER i, j
```

valid FOR statements in EL:

```
FOR( i=0; i < 5; i= i+1 )
```

```
     k[i]= 0.0
END FOR
FOR (i=0, j=0; i < 5 AND j < 5; i= i+1, j= j+2 )
     k[i]= 0.0
END FOR
```

In the second type, only one iteration variable may be used; you do not have to declare the variables beforehand. Valid examples are:

```
FOR( i IN 1,5 )  -- from 1 to 5
     k[i]= 0.0
END FOR
FOR( i IN 1,5 EXCEPT 2)  -- from 1 to 5 except 2
     K[i]= 0.0
END FOR
```

It is also possible to iterate on an enumeration type or SET_OF type:

```
ENUM Chemicals= { H2O, CO2, O2, H2 }
FOR( i IN Chemicals EXCEPT CO2 )
     k[i]= 0.0 -- equivalent to k[H2O]=k[O2]=k[H2]= 0.0
END FOR

SET_OF(Chemicals) Mix = { CO2, H2 }
FOR( i IN mix)
  k[i] = 0.0     -- equivalent to k[CO2]=k[H2]= 0.0
END FOR
```

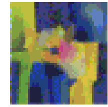
FOR and WHILE statements are variations of the same basic construct. Any WHILE loop may be expressed as a FOR loop and vice versa. FOR loops are normally used to iterate over a range of values, whereas WHILE loops are used when the entry condition is a logical expression.

5.1.7. ASSERT Statements

EL has a powerful method for checking integrity based on assertions. The user may include ASSERT statements in the sequential instructions block, to ascertain whether a condition is met or not. The syntax is as follows:

```
ASSERT expression (NOTE | WARNING | ERROR | KILLPOINT | FATAL) message
```

When the expression evaluates to FALSE, the assertion is activated and a message, which you specify is displayed. Assertions are a way of controlling the consistency of the model by periodically checking on the values of fundamental variables. There are different types of assertions which give rise to different actions:



- **NOTE**: Simple emission of a message to the user. The simulation continues
- **WARNING**: Something unexpected has occurred, but it is not serious. The simulation continues
- **ERROR**: A serious assertion. The simulation continues
- **KILLPOINT**: A serious assertion. The simulation kills the current calculation but it continues with the following ones.
- **FATAL**: A serious assertion which does not allow the simulation to continue

Example:

```
tp_out.radiation= evalRadiation(tp)
ASSERT (tp_out.radiation < maxRad 0.) FATAL "DANGER, RADIATION!"
```

In this example, the variable tp_out.radiation has been calculated as well as its integrity, which must be lower than a limit. If the condition is FALSE, the specified message is emitted and the simulation halts.

5.1.8. RETURN Statement

The RETURN instruction is a special statement only used by functions in order to return a value (no value if the function returns a NO_TYPE data). Its syntax is as follows:

```
RETURN expression
```

where the expression may be any valid expression, compatible with the type of the value returned by the function.

Examples:

```
RETURN 4
RETURN a / b / c / d / 5.67546
RETURN flag AND NOT up
```

The use of this statement will be explained later, in the section on functions.

5.1.9. STOP Statement

The STOP statement can be used to halt a simulation at any time. The syntax is as follows:

```
STOP  "The ball reaches the ground"
```

The following example shows the behaviour of a rubber ball falling from a certain height and the simulation halts when the height of the bounce becomes less than a certain value:

```
COMPONENT bouncingBall
   DATA
      REAL g = 9.80665            "Gravity (m/s^2)"
      REAL k = 0.8                "Restitution coefficient (-)"
   DECLS
      REAL h = 10.                "Height (m)"
   DISCRETE
      -- Event when bouncing on the ground
      WHEN (h < 0) THEN
         h' = -k * h'
      END WHEN
      -- Event to detect stopping the simulation
      WHEN (h < -1e-6) THEN
            STOP "****** End of simulation: height < -1e-6"
      END WHEN
   CONTINUOUS
      h'' = -g
END COMPONENT
```

In the above example, when the variable ‘h’ becomes negative the simulation halts and the following message is shown on screen:

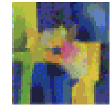
```
--- STOP SIMULATION ---
****** End of simulation: height < -0.003
```

5.2. Continuous Statements

Continuous statements are those which define the mathematical models in the continuous part. Basically, they are differential-algebraic equations which act continuously, not sequentially. This means that the order in which the equations are written is not significant, as the internal mathematical algorithms will sort them later, using different criteria.

Continuous statements define continuous models, but these models will be affected by events in the discrete part (see Discrete Statements). When the continuous model is being integrated, any events that you indicate are observed. If any specified event transpires, the integrator halts and the event is handled. This is a key process in EcosimPro and will be explained in more detail later in the description of the components.

Continuous statements can only be placed in the CONTINUOUS block for components and ports. The following is a description of the different continuous statements allowed in EL. A valid concurrent statement is:



```
vOut - vIn = r * i
```

This expresses an equation for Ohm's law. The following paragraphs explain in more detail the different concurrent statements allowed in EcosimPro.

5.2.1. Mathematical Equations

Mathematical equations are used to describe physical behaviour. Differential-algebraic equations (DAEs) with the following structure may be used:

$$0 = f(t, x(t), x'(t), z(t), d)$$

where:

- t is the independent variable, which is always time. In EL, time is always denominated TIME (uppercase)
- x(t) is the set of algebraic variables which are differentiated
- x'(t) is the set of derivatives (with respect to t) of the x(t) sets used
- z(t) is the set of algebraic variables which are not differentiated
- d is the set of discrete constants and variables which do not change in the continuous model

The BNF notation which defines a mathematical equation is:

```
conc_stm        ::=  implicit? expression  '='  expression EOL
```

Examples of valid mathematical equations are shown below. As can be seen, the common form is almost identical to the EL form.

Common mathematical formulation	In EL language
$x = 3y$	x= 3*y
$x - 2z + sin(tw) = 0$	x – 2*z + sin(TIME*w) = 0
$y''' + 3y' - y = \cos(t)$	y''' + 3*y' + y = cos(TIME)
$x^2 - \log(x) = y'$	x**2 – log(x) = y'

5.2.2. Continuous EXPAND Statement

The user can insert multiple equations in one go by using EXPAND statements. They act like a FOR statement; the difference is that the EXPAND statement will populate them as if the user were to insert different and independent equations. The FOR statements maintain the loop structure until the execution time.

None, one or many continuous EXPAND statements may be inserted in the continuous part, depending on the type of EXPAND. The syntax is as follows:

```
contExpandStm ::=EXPAND '(' loop_schema  ')'continuous_stm |
```

The EXPAND generates multiple statements based on an index variable which changes its value. The syntax of loop_schema is as follows:

```
loop_schema::=IDENTIFIER IN range(EXCEPT expression_s )?
```

For example, you can save space for inserting the next four equations. If you wished to generate the following equations in the continuous part:

```
h_out[1]= h_in[1]
h_out[2]= h_in[2]
h_out[3]= h_in[3]
h_out[4]= h_in[4]
```

you could do it with a single EXPAND statement as follows:

```
EXPAND( i IN 1,4)  h_out[i]= h_in[i]
```

It is not necessary to predeclare the variable i, as it is local to the loop; it will be iterated for the values 1, 2, 3 and 4. It is also possible to iterate over an enumeration type:

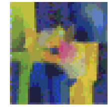
```
EXPAND( i IN Chemicals) h_out[i]= h_in[i]
```

In this case, i will iterate over all the values of the enumeration type “Chemicals”; ie, CO2, H2O, O2. Both EXPANDS allow for exceptions, or skipping over a value:

```
EXPAND( i IN Chemicals EXCEPT CO2,O2) h_out[i]= h_in[i]
```

Here, i will iterate over all the values of “Chemicals” except for CO2 and O2. This is also possible for numeric data types. The example below shows how to transpose a matrix using two nested EXPANDS:

```
EXPAND( i IN 1,3)
     EXPAND(j IN 1,4 EXCEPT i)
          newMatrix(j,i) = matrix(i,j)
```



There is also a block version of the EXPAND statement called EXPAND_BLOCK continuous. This statement allows several equations to be introduced in the same EXPAND. The syntax is:

```
EXPAND_BLOCK  '(' loop_schema  ')'
     continuous_stm_s
END EXPAND_BLOCK
```

For example:

```
EXPAND_BLOCK ( i IN 1,3 )
     x[i]= y[i] + 4
     x1[i]= y1[i] + 5
END EXPAND_BLOCK
```

This has the same effect as writing:

```
EXPAND ( i IN 1,3 )
     x[i]= y[i] + 4
EXPAND ( i IN 1,3 )
     x1[i]= y1[i] + 5
```

The modeller can also use nested EXPAND_BLOCK statements:

```
EXPAND_BLOCK ( i IN 1,3 )
   EXPAND_BLOCK ( j IN 1,2 )
     EXPAND_BLOCK ( k IN 1,2 )
           x[i,j,k]= y[i,j,k] + 4
           x1[i,j,k]= y1[i,j,k] + 5
     END EXPAND_BLOCK
   END EXPAND_BLOCK
 END EXPAND_BLOCK
```

The usage of EXPAND_BLOCK is quite convenient when two or more equations are expanded according to the same expression.

5.2.3. Conditional Insertion of Equations

EL allows conditional equations to be introduced in several ways. There are two ways of using conditional equations in EL:

- The complete insertion of a set of equations is modelled for one or several conditions. No equation changes are allowed during the simulation. It is done by means of the IF () INSERT statement

- An equation is modelled that can change during the simulation. In this case a continuous verification must be performed to check which of the alternatives provided is valid, and that will be the equation used. Within this category there are two variants: one where the system copes with the non-linearities (if any) automatically (ZONE statements) and another where the non-linearities are not dealt with (IF equation statements)

EL allows a set of equations to be introduced conditionally in the model at the beginning. Since the final equations must be known when the models are generated, the conditions that can be applied for these conditional insertions must only be based on the use of construction parameters, because they need to be established when the models are generated.

The statement used for this purpose is of the IF type. Its exact syntax is as follows:

```
     if_eqts_stm      : IF expression EOL* INSERT conc_stm_s
              (ELSEIF expression INSERT conc_stm_s?)*
                       (ELSE conc_stm_s)?  END IF
```

An example of its use would be as follows:

```
COMPONENT tinsert(INTEGER sw = 1)
 DECLS
  REAL x
  REAL y
 CONTINUOUS
    IF( sw == 1 ) INSERT
       3*x - 6*y = 9
       4*x - 4*y = 9
    ELSEIF( sw == 2 ) INSERT
        2*x + 2*y =  545
        4*x - 4*y = 54
    ELSE
        7*x + 6*y =  34
        8*x - 8*y = 75
    END IF
END COMPONENT
```

This component proposes three different sets of equations depending on a condition that uses the “sw” construction parameter. No data or variables can be used in this condition because it must be definitively evaluated when the mathematical model is generated.

For example, the following component would not be valid:

```
COMPONENT tinsert
 DATA
  INTEGER sw = 1
```

```
 DECLS
  REAL x
  REAL y
 CONTINUOUS
    IF( sw == 1 ) INSERT
       3*x - 6*y = 9
       4*x - 4*y = 9
    ELSEIF( sw == 2 ) INSERT
        2*x + 2*y =  545
        4*x - 4*y = 54
    ELSE
        7*x + 6*y =  34
        8*x - 8*y = 75
    END IF
END COMPONENT
```

In this case “sw“ is a datum and not a construction parameter. This example will be compiled, but it should produce an error when generating the mathematical model:

```
ERROR:
Expression (sw == 1) needs to be fixed at generation time.
Only construction parameters and closed data/variables (with CLOSE) can be
used in these expressions
```

This means that the expression “sw == 1” depends on a datum that can change during the simulation and is therefore not allowed for this statement.

It is also possible to nest conditions, as in the following example:

```
COMPONENT t2 (INTEGER sw1= 1, INTEGER sw2= 1)
 DECLS
  REAL x
  REAL y
 CONTINUOUS
    IF( sw1 == 1 ) INSERT
       IF( sw2 == 1 ) INSERT
           2*x - 3*y = 77
           5*x - 4*y = 35
       ELSE
           4*x - 6*y = 46
           4*x - 9*y = 84
       END IF
    ELSEIF( sw1 == 2 ) INSERT
        2*x + 2*y =  545
        4*x - 4*y = 37

    END IF
```

```
END COMPONENT
```

5.2.4. Use of Conditional Equations with Detection of Events

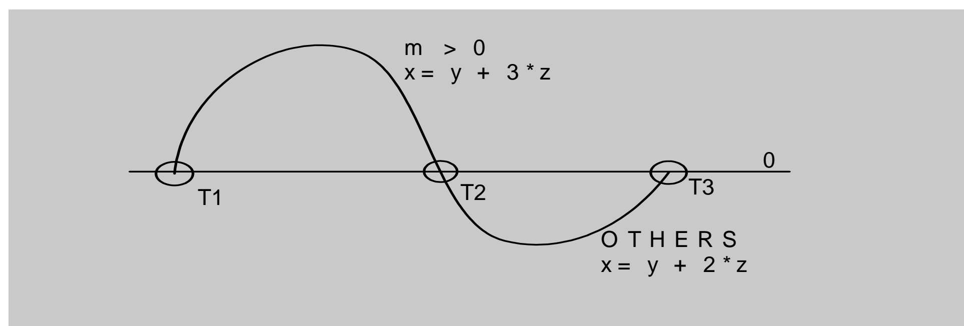
EL allows the use of continuous equations with dynamic changing of the valid equation depending on certain conditions and detecting the exact events of crossing. The ZONE statement is used for this purpose. The syntax of ZONE statements is shown below:

```
concStm ::= implicit? expr1  '='
( ZONE '(' expr2 ( TOL realNumber )? ')'  expr3 ) +
OTHERS  expr4
```

The term “implicit” is explained later (see IMPL operator in this chapter), “expr1” is the left-hand side of the equation made equal to a variable term. This variable term may contain any number of ZONE statements. Each ZONE statement has a logical satisfaction condition “expr2” and an associated tolerance “realNumber”. When the first of these conditions is met, “expr3” is taken as the right-hand side of the continuous equation. If no conditions are met, then “expr4” is taken as default (the OTHERS term is obligatory). An example is shown below:

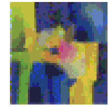
```
x=  ZONE ( m > 0 ) y + 3*z
         OTHERS    y + 2*z
```

When the value of m is greater than zero, the valid equation is x = y + 3*z , and when it is less than zero, the valid equation is x= y + 2*z. In the following figure, note the importance of times T1, T2 and T3; at these times the equations change and a re-initialization occurs.



In this way, a continuous equation can have a variable expression, depending on certain conditions. If the ZONE statement is cascaded as follows:

```
x = ZONE ( m > 0 ) y + 2*y
     ZONE ( m > 2 ) y + 3*y
     ZONE ( m > 4 ) y + 4*y
```



```
      OTHERS y + 5*y
```

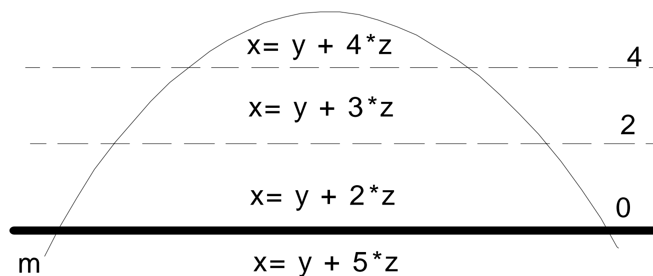
The first condition to evaluate to TRUE will become active. In this example, when m > 0, the first branch is activated, and if not, the last branch is activated. In order to activate the second or third branch, the equation must be rewritten:

```
x = ZONE ( m > 0 AND m < 2 ) y + 2*y
      ZONE ( m > 2 AND m < 4 ) y + 3*y
      ZONE ( m > 4 ) y + 4*y
      OTHERS y + 5*y
```

Or even better:

```
x = ZONE ( m > 4 ) y + 4*y
      ZONE ( m > 2 ) y + 3*y
      ZONE ( m > 0 ) y + 2*y
      OTHERS y + 5*y
```

Now the desired effect is implemented. The following figure shows 4 zones, with a different valid equation in each zone.



5.2.5. Use of Conditional Equations without Detection of Events

A conditional equation may be modelled with an IF statement without forcing EcosimPro to deal with potential non-linearities in the system of equations. The syntax is similar to the ZONE statement:

```
-- define an enumerative type with different modes
ENUM modes= { DESIGN, OFF_DESIGN, TRANSIENT }

-- creates the component
COMPONENT tif
 DECLS
   REAL x= 3
   ENUM modes switchMode= DESIGN
```

```
 CONTINUOUS
        x= IF    ( switchMode == DESIGN )    5.6
           ELSEIF( switchMode== TRANSIENT ) sin(TIME)
           ELSE                              sin(TIME) + cos(TIME)
END COMPONENT
```

The conditional equation for x shall be valid as a function of the condition:

- If the switchMode variable is DESIGN, the valid equation is:

  x = 5.6

- If the switchMode variable is TRANSIENT, the valid equation is:

  x = sin(TIME)

- And if switchMode has a different value

  x = sin(TIME) + cos(TIME)

In this case, unlike the ZONE statement, the system will not solve the entire equations system with the new equation, so the change will be immediate and it will not introduce any delay. This form of modelling is adequate for conditional equations that do not change during simulation (typically for steady states), as when the model is for switches that are only started at the beginning of the simulation and do not change later (as in the above example).

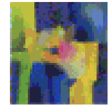
The general form of this statement is as follows:

```
implicit? expression  '='
     IF '(' expression ')' EOL* expression EOL*
     ( ELSEIF '(' expression ')' EOL* expression EOL*)*
     ELSE  EOL* expression EOL
```

There is no limit to the number of IF branches, but the last one must always be an ELSE branch so as to use the equation by default if none of the conditions is fulfilled.

During the simulation, every time the value of a branch changes, the system will display a message with the new active equation (similar to the ZONE).

This IF statement is faster than the aforementioned ZONE because it avoids the need to solve the system for the new active equation. However, in turn, it does not implement the control of non-linearities that is carried out by ZONE. If the discontinuity is not very big, the solver can sometimes carry out the changes using IF, but it all depends on the model.



5.2.6. Sequential Blocks of Equations

EL allows sequential blocks to be placed inside the CONTINUOUS block. The SEQUENTIAL block is supplied for this purpose and works as though everything it contains is executed in a purely sequential manner.

With this block inside the CONTINUOUS block, code can be entered sequentially inside the continuous set of equations. In fact, the effect is similar to that when we call an external function which performs a calculation and returns the control. For example, with the function limitVariable() and the following component "foo":

```
FUNCTION NO_TYPE limitVariable(OUT REAL result,
                          IN REAL x,
                          IN REAL lowerLimit,
                          IN REAL upperLimit)
BODY
     IF( x < lowerLimit) THEN
          result= lowerLimit
     ELSEIF ( x > upperLimit) THEN
          result= upperLimit
     ELSE
          result= x
END FUNCTION
COMPONENT foo
DATA
     REAL lowerLimit= 0.2
     REAL upperLimit= 0.5
DECLS
     REAL x
     REAL y
CONTINUOUS
     x= sin(TIME) + 9
     limitVariable (y,x,lowerLimit,upperLimit)
END COMPONENT
```

This example calculates the value of "x" and then the value of "y" in the call to function "limitVariable()" taking into account the logic of the function.

We could obtain the same behaviour with a SEQUENTIAL block inside the component:

```
COMPONENT foo
DATA
     REAL lowerLimit= 0.2
     REAL upperLimit= 0.5
DECLS
     REAL x
```

```
      REAL y
CONTINUOUS
      x= sin(TIME) + 9
      SEQUENTIAL
            IF( x < lowerLimit) THEN
                  result= lowerLimit
            ELSEIF ( x > upperLimit) THEN
                  result= upperLimit
            ELSE
                  result= x
            ENDIF
      END SEQUENTIAL
END COMPONENT
```

There are at least two advantages in modelling it in this way:

- There is direct access to all the component variables. If it is done in a function, they would have to be passed as arguments of the function
- The component encapsulates all its behaviour

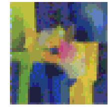
As a general rule it could be said that it makes sense to use an external function when the calculation it performs is general and applies to more components. If the logic is only used by one component, the modeller could introduce it in the component with a SEQUENTIAL block.

Modellers can insert as many SEQUENTIAL blocks as they like inside a single component. Each one of them will be treated as compact blocks of code EcosimPro will sort as another equation.

SEQUENTIAL blocks are also very useful when the modeller wants to impose a computational causality because EcosimPro will never try to symbolically transform the content of these blocks; in other words, if a SEQUENTIAL block calculates a variable “x”, EcosimPro will always calculate the variable “x” inside that block and will never solve it in another equation.

Another advantage is that EcosimPro will always arrange the SEQUENTIAL blocks the same way as it does with other equations, so that the variables which are used in this block are calculated first. For example, given the component:

```
COMPONENT foo
DECLS
      REAL x
      REAL y
CONTINUOUS
      SEQUENTIAL
            x= y + sin(TIME)
      END SEQUENTIAL
```



```
    y = cos(TIME)
END COMPONENT
```

EcosimPro detects that variable “y” needs to be calculated first and it will sort the final equations as:

```
y= cos(TIME)
SEQUENTIAL
    x= y + sin(TIME)
END SEQUENTIAL
```

A disadvantage of SEQUENTIAL blocks is the discontinuities the modeller can introduce using statements like IF. It is the responsibility of the modeller to carefully handle these cases if the model is complex and could lead to convergence problems. Do not forget that to handle discontinuities, EL provides the ZONE and WHEN statements which can deal with them in a smart way.

5.3. Discrete Statements

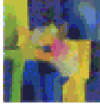
As described above, EcosimPro allows events to be defined in the continuous part. These events cause the continuous model to halt so that they can be dealt with. After they have been dealt with the continuous model resumes. As a trivial example, let us assume that we are modelling a ball bouncing on the ground: the following position equation may be used, but it is obvious that when the ball touches the ground, we have to express an event so that the direction of movement can be changed. This is done by means of discrete statements, which may only be inserted in the DISCRETE block of the components.

The basic statement is the WHEN statement, which declares an event by waiting for a condition to become TRUE, at which point the associated code is executed. There are other discrete statements, which are also described in this section:

5.3.1. WHEN Statements

As mentioned above, a WHEN statement defines two things: when a discrete event is activated (logical condition) and what action is taken (sequential actions). Its syntax is as follows:

```
WHEN eventCondition ( TOL expression )?  THEN
    seq_stm_s
END WHEN
```

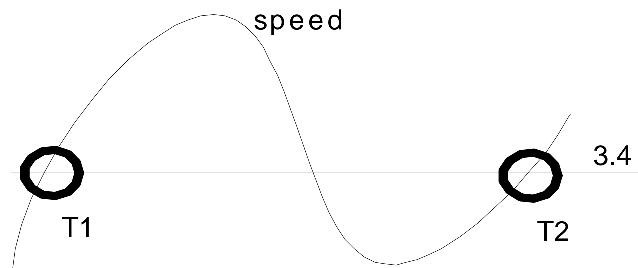


The event is activated when eventCondition evaluates to TRUE. The logical condition may use any variable in the model (continuous or discrete) and the logical operators AND, OR, NOT, ==, >, >=, <, <= and !=. When the condition is met, the integrator notes the exact time and executes the associated actions. It does not reactivate until the condition first becomes FALSE and then TRUE again.

For example:

```
DISCRETE
     WHEN( speed > 3.4 ) THEN
          reachedMaxSpeed= TRUE
     END WHEN
```

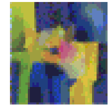
When the condition “speed > 3.4” is met, the event is activated and the associated action is executed. This event will not be reactivated until the speed first becomes less than 3.4 and then greater than 3.4 again. The following graph shows this concept:

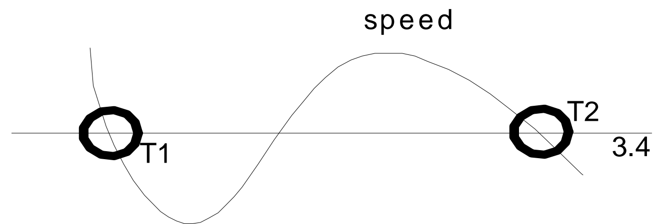


The event is only activated at times T1 and T2, which is when the condition is met. The following are examples of the meaning of each one of the relation operators:

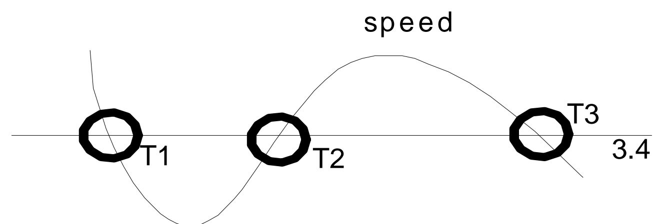
">" y ">=" have already been explained above. The only difference between the two is that ">=" considers the crossover at the exact point of the cut, while the ">" operator considers it at a higher epsilon value

"<" y "<=" activate the condition when they cross from the positive axis to the negative axis; ie, the opposite of the ">" operator. In the drawing below, they are only activated at T1 and T2





"==" y "!=" activate the condition when they cross from the positive axis to the negative axis, and from the negative axis to the positive axis. For example, for “speed == 3.4”, it would activate at T1, T2 and T3



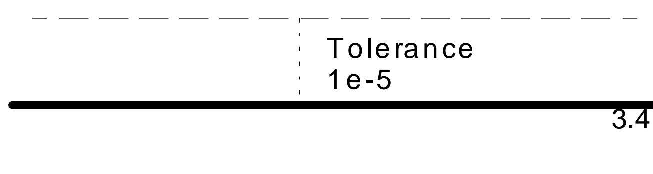
WHEN conditions can be composed of several simple conditions. For example:

```
WHEN( speed > 3.4 AND height <= 5.6 AND NOT flag)
```

Optionally, a numerical tolerance may be associated with the condition. Basically, this is equivalent to defining a numerical margin around the mathematical conditions. This is useful depending on the type of data being modelled. For example:

```
DISCRETE
     WHEN( speed > 3.4 ) TOL 1e-5 THEN
          reachedMaxSpeed= TRUE
     END WHEN
```

The specified tolerance value is the permitted tolerance around 3.4 in this case; ie, the following tolerance margin is created:



This is used to give the crossovers a numerical tolerance. Note that there is a difference between crossovers with variables which normally have values close to zero, and variables whose values oscillate around some high value. The first case requires a smaller tolerance as crossovers around zero are more sensitive, while in the second case a larger tolerance is possible. EL sets a default tolerance of 1e-7.

When the WHEN statement is activated, the associated sequential actions are executed. For example:

```
DISCRETE
     WHEN( speed > 3.4 ) TOL 1e-5 THEN
          IF ( speed > 3.5 AND NOT state == OFF )
               state = ON
          END IF
          FOR (i IN 1,3)
               initialSpeed[i]= 0.0
          END FOR
          reachedMaxSpeed= TRUE
     END WHEN
```

In this case, when the WHEN statement is activated, the system will execute the internal sequential statements as they are normally executed. In this case there is an IF statement, a FOR statement and an assignment statement.

5.3.2. ASSERT Statement in DISCRETE Block

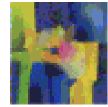
The discrete ASSERT is similar to the sequential ASSERT (see sequential statements), the only difference being the instant in time when the check is carried out. In a sequential ASSERT, the check is calculated when the execution reaches the ASSERT statement, while a discrete ASSERT is automatically calculated at the end of each integration step. For example:

```
DISCRETE
     ASSERT (tp_out.radiation > 0.) FATAL "Radiation is equal to zero"
```

In the above example, the compiler checks that the variable tp_out.radiation is greater than zero at each integration step; if it isn't, the specified message is emitted and the simulation halts.

5.3.3. Discrete EXPAND Statement

The EXPAND statements in the discrete part are similar to those in the continuous part; ie, the conditional insertion of none, one or many statements. The syntax is the same:



```
expandStm ::= EXPAND '(' loop_schema  ')' discrete_stm |
EXPAND  '(' expression ')' discrete_stm
```

The difference is that the statements which may be generated must be of the discrete type; ie, other EXPAND, WHEN or ASSERT statements. Remember that in the case of continuous expand statements, equations were generated. It will not explain each type of EXPAND as they are similar to the continuous EXPAND statements. The following are a few examples:

- Generates an event for a component if a condition is met:

```
EXPAND( flag == TRUE)
     WHEN( speed > 3.4 )
          reachedMaxSpeed= TRUE
     END WHEN
```

This segment of code inserts the WHEN statement in model generation time, if the condition is met. The condition must be evaluable in model generation time.

- Generates a series of events based on an index:

```
EXPAND( i IN 1..5)
     WHEN( v[i] > 67.0 )
          reachedMaxSpeed= TRUE
     END WHEN
```

This EXPAND generates 5 WHEN statements, one for each condition of the index.

- Simultaneous use of logical EXPAND and index EXPAND:

```
EXPAND ( flag == TRUE )
     EXPAND( i IN 1,5)
          WHEN( v[i] > 67.0 )
               reachedMaxSpeed= TRUE
     END WHEN
```

Generates 5 WHEN statements if the logical condition is met.

- Use of two nested EXPANDS:

```
EXPAND ( compo IN Chemicals )
     EXPAND( i IN 1,5)
          WHEN( v[compo,i] > 67.0 )
               reachedMaxSpeed= TRUE
     END WHEN
```

Assuming that the enumeration type “Chemicals” has 3 elements, 15 WHEN statements will be generated, one for each index of the condition.

□ Use of an index EXPAND with exceptions:

```
EXPAND( compo IN Chemicals EXCEPT CO2,H2O)
     WHEN( v[compo] > 67.0 )
          reachedMaxSpeed= TRUE
     END WHEN
```

In this case as many WHEN statements will be expanded as there are elements in the enumeration type “Chemicals”, except for elements CO2 and H2O.

There is also a block version of the EXPAND statement called EXPAND_BLOCK discrete. This statement allows several discrete events to be introduced in the same EXPAND. The syntax is:

```
EXPAND_BLOCK  '(' expression ')'
     discrete_stm_s
END EXPAND_BLOCK
```

For example:

```
EXPAND_BLOCK ( i IN 1,3 )
     x[i]= y[i] + 4
     x1[i]= y1[i] + 5
END EXPAND_BLOCK
```

This statement allows the modeller to introduce several discrete statements in the same EXPAND, for example:

```
EXPAND_BLOCK ( i IN 1,2 )
        WHEN( x[i] > 9 ) THEN
               x[i]= 87
        END WHEN
        ASSERT(x[i] < 7) FATAL "Fatal error”
        WHEN( x[i] > 9 ) THEN
               x[i]= 88
        END WHEN
END EXPAND_BLOCK
```

This unique EXPAND_BLOCK is equivalent to the next three EXPAND:

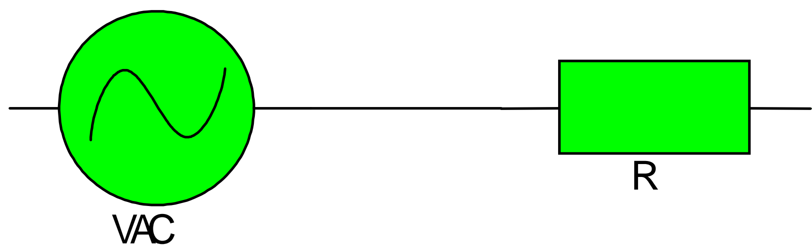
```
EXPAND ( i IN 1,2 )
     WHEN( x[i] > 9 ) THEN
          x[i]= 87
     END WHEN
EXPAND ( i IN 1,2 )
     ASSERT(x[i] < 7) FATAL "Fatal error”
EXPAND ( i IN 1,2 )
     WHEN( x[i] > 9 ) THEN
```

```
        x[i]= 88
    END WHEN
```

The EXPAND_BLOCK can also be nested.

6. Ports

A connection port connects components, both by the values that each needs to communicate with its environment and by the restrictions and equations applicable to the connections. By defining new ports you can model any kind of exchange of values between components. Ports are part of a component's interface. For example, to model an electrical circuit you can use an electrical port to automatically handle the voltage and current variables. This allows you to use an abstract electrical port to connect, for example, a voltage source to a resistance (see figure). There's no need to be concerned about specifying the variables which are connected between components; and ports have the intelligence necessary to handle multiple connections (e.g., output from a generator to the input to two resistances).

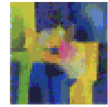


The basic mechanism for modelling a system in a modular way is to connect components together using their ports. The connections create a composite component, which may in turn have ports which are connected internally to ports of the components which have been instantiated. From outside, the composite component can be used to connect to its external ports.

Each port of a component must have a mode, either IN or OUT. Modes have three purposes:

- They serve as a reference for the sign of the flow-variables. A positive IN port implies inward flow. A positive OUT port indicates outward flow
- They are a criterion for applying connection equations
- They are used to check for violations of connection restrictions specified in the port definition

Different port types should be defined for different kinds of connections. Some typical EcosimPro ports and their port variables are:



- Fluid Ports: mass flow, pressure,…
- Thermal Ports: temperature, heat flow,…
- Electrical Ports: voltage, current, …
- Shaft Ports: rotational speed, power, torque, …

6.1. Port Definition Syntax

The syntax is as follows:

```
port_def:  PORT ID( IS_A scoped_id_s )? param_def_s? (SINGLE (IN | OUT)? )?
                STRING_VALUE?
                DECLS? port_var_decl_s
               (CONTINUOUS conc_stm_s)?
          END PORT

port_var_decl: (BOUND|EXPL|DISCR)? ( (SUM|EQUAL) (IN|OUT)? )?  obj_decl
                 ( '=' init_expression )? obj_range? STRING_VALUE?

param_def_s      : '(' param_def ( ',' param_def )*  ')'

param_def        : obj_decl ( '=' expression ) ? STRING_VALUE?
                 | enum_set_decl

obj_decl         : data_type IDENTIFIER ( '[' expression_s  ']' ) ?
```

The name of the port is declared using an identifier. A port can also contain special information about:

- Inheritance. This is done using the keyword IS_A. It indicates that a port inherits from one or more ports
- Construction parameters. These are special data that cannot be changed during simulation
- The different variations on the SINGLE modifier are used to specify restrictions. Refer to the section **Port Connecting Restrictions**
- An optional description of the port in a textual format (e.g., "Electrical port")
- DECLS. This block declares the port variables. The different variants of SUM and EQUAL modifiers indicate how to construct the connection equations for each port variable. Also, the presence or absence of these modifiers affects the restrictions applicable to the connections between this type of port. Refer to the sections **Connecting Equations and Connecting Restrictions**

- CONTINUOUS: This block contains the continuous behaviour of the port. These equations are written in terms of differential-algebraic equations. They allow you to calculate the values of certain port variables in terms of other port variables

The construction parameter list is used to generalise the formulation of the port. Construction parameters should be assigned in the declarations of the component ports and should be evaluable when the simulation model is generated.

For example, a construction parameter should be an integer value used to dimension an ARRAY type port variable.

```
PORT a_port_type ( INTEGER N ) "example of port"
DECLS
      REAL v[N]
```

Another example is the use of a generic mix to formulate a fluid component:

```
PORT fluid ( SET_OF(Chemicals) mix ) "fluid port"
REAL conc [mix]  "Concentration of Chemical compounds in the fluid"
```

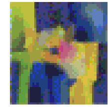
6.2. Inheritance

The inheritance in the ports is simple to use. Basically it allows everything to be inherited from one or more parent ports. For instance:

```
PORT Basic1 "basic port1"
DECLS
   SUM REAl a
END PORT

PORT Basic2 "basic port2"
DECLS
   EQUAL REAL b
END PORT

PORT myPort IS_A Basic1, Basic2
DECLS
   SUM REAL c
END PORT
```

This is equivalent to writing:

```
PORT myPort
DECLS
   SUM REAl a
   EQUAL REAl b
```



```
   SUM REAL c
END PORT
```

Inheritance is useful for creating basic ports for any discipline and then reusing them by inheriting a more complex version of the ports.

6.3. CONTINUOUS Block

The statements in the CONTINUOUS block of a port definition are generated for each port of that type, except for one special case. This is when an IN port and an OUT port are connected but are not connected to any other node; in this case the two ports are considered completely equivalent so the statements of the CONTINUOUS block are only generated once.

6.4. Port Connecting Equations

These are necessary in two cases:

- When a port is connected to more than one port
- When a port is connected to another port on the same node. IN-IN or OUT-OUT

In both these cases the connected ports cannot be considered equivalent, so the simulator must add the connection equations to relate the variables to the ports involved.

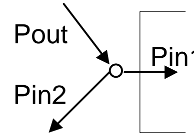
The connection equations are generated for each port variable. The equation generated depends on the multiplicity modifier specified in the port_var_decl.

There are three basic behaviours:

- EQUAL: a variable with this behaviour maintains the same value for all the ports of a connection, regardless whether it's a multiple connection or whether it connects ports at the same node. In other words, it forces equivalence of the variables
- SUM: this is associated with flow type variables and indicates that on one connection the sum of the incoming flows is equal to that of the outgoing flow
- No indication: no connection equation is generated for the variable

For example, an electrical model with two port variables: voltage and current. The voltage variables have the same value at both connections. The current variables are flows.

```
PORT Elec "electrical port"

DECLS

    REAL i   -- "current (A)"

    REAL v   -- "voltage (V)"

END PORT
```



Equations:

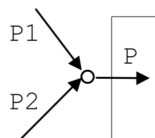
Pin1.v = Pin2.v = Pout.v
Pin1.i + Pin2.i = Pout.i

In some cases, the behaviour of a port variable differs depending on whether it has multiple connections at an IN port or at an OUT port. These cases are covered by adding the auxiliary modifiers IN or OUT to SUM or EQUAL. It means that a variable will have the SUM or EQUAL behaviour only if the port has the same direction as the auxiliary modifier. If not, the connecting equation is not generated.

A tiny fluid port illustrates this concept:

```
PORT fluid      "fluid port"
DECLS
      SUM REAL w                "mass flow"
      EQUAL REAL p         "pressure"
      SUM IN REAL E        "energy flow"
      EQUAL OUT REAL T          "temperature"
CONTINUOUS
      E = w * T
END PORT
```

Multiple input port



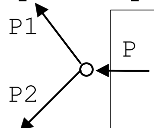
Connecting Eqts:

```
P.w = P1.w + P2.w
P.p = P1.p = P2.p
P.E = P1.E + P2.E
```

CONTINUOUS Eqts:

```
P1.E = P1.w * P1.T
P2.E = P2.w * P2.T
P.E  = P.w * P.T
```

Multiple output port

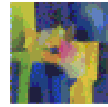


Connecting Eqts:

```
P.w = P1.w + P2.w
P.p = P1.p = P2.p
P.T = P1.T = P2.T
```

CONTINUOUS Eqts:

```
P1.E = P1.w * P1.T
P2.E = P2.w * P2.T
P.E  = P.w * P.T
```



6.5. Port connecting restrictions

In some kinds of models it is useful to specify some restrictions on how ports can be connected.

There are two groups of restrictions:

- Limits on the number of ports that can be connected together
- Limits on the modes of ports connected together

The first limitation implies the second; in other words, if a port type does allow multiple connections, this implies that one of the port's connections must be mode IN and the other mode OUT. The contrary is not always true: multiple connections are possible but only if several IN mode ports are connected to an OUT port, or vice versa. In a port definition, the restriction conditions for an IN mode port may be different to those for an OUT mode port. For instance, in control models, it is natural to transmit an output signal to multiple points, but it is meaningless to have several signals connected to an input.

Restrictions may be included for two reasons:

- The developer includes them
- Lack of information in the definition of the variables and the port's CONTINUOUS block

The SINGLE modifier expressly forces all connections for a port type to be single. If no mode is specified, the restriction applies to all ports of this type, regardless of mode. If either of the modifiers IN or OUT are added, the restriction only applies to ports with that mode. For instance, the control port allows multiple connections to an output but it does not allow multiple connections to an input. The modifier SINGLE IN is used to forbit multiple connections at IN ports:

```
PORT control SINGLE IN "control port example"
DECLS
     EQUAL REAL signal
END PORT
```

If there are no EQUAL or SUM modifiers on any of the variables, only non-multiple connections from an OUT port to an IN port are allowed.

If there are multiplicity modifiers on some but not all the variables, multiple connections are valid if the port_definition's CONTINUOUS block has equations which define variables with no multiplicity information as a function of other port variables.

Even when multiple connections are allowed, connections to ports of the same type (IN to IN, OUT to OUT) are not if any of the multiplicity modifiers have a mode restriction. For example, the fluid port allows multiple connections but not IN to IN or OUT to OUT

as there are mode restrictions on the energy flow and temperature variables: E (SUM IN) and T (EQUAL OUT).

6.6. Bound and Expl Operators

The meaning of these operators in the PORT declaration is explained later in the DECLS block of the components. The contents are also applied to the variables in the ports.

6.7. Examples of Port Types

In this section several real examples of port types are shown. For instance, a thermal port with temperature and heat flow:

```
PORT thermal  "thermal port example"
DECLS
     EQUAL REAL T         "Temperature (C)"
     SUM REAL Q           "Heat flow (W)"
END PORT
```

An electrical port with current and voltage:

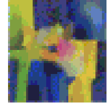
```
PORT elec  "electrical port example"
DECLS
     SUM REAL c           "current (Amperes)"
     EQUAL REAL v         "voltage (Volts)"
END PORT
```

An analog signal port with a unique signal. In this case the port type has the constraint that the input connection must be single; it means that several outputs cannot be connected to an input port, but many output connections can be connected. The signal has an "EQUAL" behaviour but only when it is an output port:

```
PORT analog_signal SINGLE IN "analog signal"
DECLS
     EQUAL OUT REAL signal          "control signal"
END PORT
```

The same applies to a Boolean signal port, used to model gates like AND, OR, etc. In this case the signal is not REAL but a BOOLEAN value.

```
PORT bool_signal SINGLE IN   "boolean signal"
```



```
DECLS
     EQUAL OUT BOOLEAN signal "boolean signal"
END PORT
```

7. Components

7.1. Introduction

Components are the most important items of EL. They are the basic instrument for representing both complex and simple models. A component is a natural way to describe the behaviour of both the continuous and discrete behaviour of a model.

Normally, components are created for individual physical entities such as a capacitor, a valve or an engine. In other words, components represent physical objects which act independently but are connected to others. A component can be as simple as a single continuous equation, or merely the handling of a discrete event, or simply an aggregation of other components, without its own equations. All modelling items in EL are components.

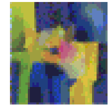
7.2. Syntax

The general syntax for defining components is as follows:

```
component_def   ::=  ABSTRACT? COMPONENT IDENTIFIER
                        ( IS_A IDENTIFIER (,IDENTIFIER)*   )?
                        ( '(' parameters_s ')' )?
                        STRING_VALUE?  EOL*
                  ( PORTS port_decl_s )?
                  ( DATA var_decl_s )?
                  ( DECLS comp_decl_s )?
            ( OBJECTS class_instace_stm_s  )?
                  ( TOPOLOGY topology_stm_s )?
                  ( INIT seq_stm_s )?
                  ( DISCRETE discrete_stm_s )?
                  ( CONTINUOUS labelled_stm_s )?
                 END COMPONENT
```

The name of the component is declared using IDENTIFIER. A component can also contain special information about:

- Abstract components, using the ABSTRACT keyword



- Inheritance. This is done using the keyword IS_A. It indicates that a component inherits from another or from others (multiple inheritance)
- Construction parameters. They are ONLY used in the construction of the component and for the creation of the component. They are mainly used to dimension arrays and pass Chemical components
- A short description of the component in a textual format (e.g., "Valve component")
- PORTS: Connection ports. These are the interface with the universe outside the component. Ports must be of a type already declared
- DATA: Component data. These are known data items of the component
- DECLS: The component‘s local variables. They are used to model the component‘s physical behaviour
- OBJECTS: The instances of classes used in this component
- TOPOLOGY: Aggregation of other components and their connections
- INIT: Initialization of the component. These are written as sequential statements
- DISCRETE: Discrete behaviour of the component. These are written as discrete statements
- CONTINUOUS: Continuous behaviour of the component. These are written as continuous statements

As all the blocks are optional, the simplest component would look something like this:

```
COMPONENT foo "example"
END COMPONENT
```

Obviously this component does not help as it does not define any behaviour, but it is perfectly valid syntactically. A component could contain only a continuous part. For example, a simple component that models a delayed variable:

```
COMPONENT delayed "delayed simple component"
      DATA
            REAL tau= 0.1
      DECLS
            REAL x, y
      CONTINUOUS
            y' = ( x - y ) / tau
END COMPONENT
```

But you could also create a component with only a discrete part. The following example shows the modelling of a flip-flop device using only discrete behaviour.

```
-- defines a simple digital port to carry on a signal
PORT digitalPort  "Definition of a digital port"
```

```
      EQUAL BOOLEAN s
END PORT
-- defines the flip_flop device
COMPONENT flip_flop (REAL delayTime) "Flip-flop example"
      PORTS
            IN digitalPort pi             "input"
            OUT digitalPort po                  "output"
            IN digitalPort clock          "from the clock"
            IN digitalPort reset          "to reset the flip-flop"
      DISCRETE
         -- when clock signal is FALSE generates an
         -- output with a delay (except when reset is ON)  WHEN ( clock.s ==
FALSE ) THEN
            IF ( NOT reset.s ) THEN
                  po.s = NOT pi.s AFTER delayTime
            END IF
      END WHEN
         -- if reset is ON, changes the output to FALSE
      WHEN ( reset.s == TRUE ) THEN
            po.s = FALSE AFTER 0
      END WHEN
END COMPONENT
```

It is not necessary to understand now the full behaviour of this component. It is only meant as an introduction to the powerful capabilities of discrete modelling in EcosimPro.

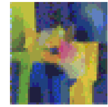
You can also create a component that contains only an aggregation of other components and their topology. Below is an example of an electrical circuit which only instantiates already developed components, but it doesn't add any new behaviour:

```
COMPONENT circuit1 "electrical circuit example"
      TOPOLOGY
            generator s1( ampl=10 )
            resistor r1 (1000)
            capacitor c1 ( 1e-1 )
            earth e1
      CONNECT s1.po TO r1.pi
      CONNECT r1.po TO c1.pi
      CONNECT c1.po TO s1.pi,e1.pi
END COMPONENT
```

The following sections explain each of the optional elements of a component.

7.3. Abstract Components

Components in EL are either abstract or concrete. Abstract components describe a behaviour which on its own does not represent any physical component, and can only be used as a base component for other components. For example, in electrical



components we can define a common interface component which defines an input and output port typical for most electrical components:

```
ABSTRACT COMPONENT twoPorts
     PORTS
          IN elec pi          "input port"
          OUT elec po         "output port"
END COMPONENT
```

This is called an interface component as it contains only connection ports to the outside universe. Any component which inherits from it will inherit the same interface. There would be no point in using this component alone to generate a model.

But an abstract component can also contain equations in the same way as a normal component. An abstract component can be just like a normal one, except that it cannot be used to create a final model.

They are very useful in any field of modelling, as they save many lines of code by bringing together the interface and equations shared by multiple components.

7.4. Inheritance

EL supports single and multiple inheritance between components. This is equivalent to exposing a child component to all the data and equations of the parent component(s). This mechanism allows you to unify in abstract or concrete components any behaviour which is common to components of the same type. For example:

```
-- Resistor component with the Ohm' Law
COMPONENT resistor IS_A twoPorts
     DATA
          REAL r= 1000        "resistance (ohms)"
     CONTINUOUS
          pi.i = po.i
          pi.v - po.v = pi.i * r        -- Ohm's law
END COMPONENT
```

The "resistor" component inherits all the data from the "twoPorts" component; in this case it inherits the two connection ports "pi" and "po" which are used in the child component as if they had been declared in it. This is equivalent to:

```
COMPONENT resistor
     PORTS
          IN elec pi          "input port"
          OUT elec po         "output port"
```

```
     DATA
          REAL r = 1000       "resistance (ohms)"
     CONTINUOUS
          pi.i = po.i
          pi.v - po.v = pi.i * r        -- Ohm's law
END COMPONENT
```

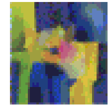
The component resistor itself can be used as a base component for new components, and so on. The inheritance mechanism is very powerful and EcosimPro users will benefit a lot by reusing components and interfaces from other existing components.

The user can also take advantage of multiple inheritance; in this case the component inherits construction parameters, ports, data, variables and behaviour from several parent components. There is one important constraint; the parent components cannot have a variable with the same name, otherwise it would be impossible to identify a variable in the child component. For instance, let's assume we have three simple components like:

```
   -- defines first base component
COMPONENT  base1
     DECLS
          REAL x
END COMPONENT
   -- defines second base component
COMPONENT base2
     DECLS
          REAL y
END COMPONENT
   -- defines third base component
COMPONENT base3
     DECLS
          REAL z
END COMPONENT
   -- defines the child component, inheriting variables
   -- and behavior from last three components
COMPONENT child IS_A base1, base2, base3
     CONTINUOUS
          x = y + z
END COMPONENT
```

This example shows the multiple inheritance of the "child" component. It is trivial because the base components don't provide any intelligent behaviour, but for a complex component the concept is identical.

For instance, if base3 has a variable called "x", an error will be issued when compiling the "child" component, because another variable with the same name was declared in



base component "base1", and in the equation "x = y + z" it would not be determined whether the variable "x" belongs to base1 or base3.

7.5. PORTS Block

The component's interface with the outside universe is through the connection ports. This block specifies which external ports are used by a component. These are the connectors to other components and they can only be used for that purpose. Instances of components can be used as normal variables and all the port's local variables are visible. The declaration syntax is:

```
PORTS
     [IN | OUT] port_type IDENTIFIER [,IDENTIFIER]
[(parameters_init)]  ( CARDINALITY range )?  STRING_VALUE?
```

where:

- □ qualifier IN indicates that the port will be used for input and OUT for output
- □ "port_type" is the defined type of the port
- □ IDENTIFIER is the name of the port (several can be declared on the same line)
- □ "parameters_init" is an optional list of the init construction parameters (if the port requires them)
- □ "CARDINALITY range" is an optional flag to specify the minimum and maximum number of allowed connections to this port. If this flag is not present there are no restrictions on the number of allowed connections. Two formats are valid: CARDINALITY min,max and CARDINALITY value. The first format is used to specify a range of minimum and maximum number of allowed connections (e.g., CARDINALITY 0,2). The second format imposes a unique valid number of connections (e.g., CARDINALITY 1). The second format is equivalent to CARDINALITY value, value.
- □ "STRING_VALUE" is an optional field to write a short description of the port object

For example, the "twoPorts" component above, which declared the use of an instance of an electrical input (IN) port called "pi" and an output (OUT) port "po":

```
ABSTRACT COMPONENT twoPorts
     PORTS
          IN elec pi                              "input port"
          OUT elec po    CARDINALITY 0,2          "output port"
     CONTINUOUS
          pi.i = po.i
```

```
END COMPONENT
```

The variable "pi" will be taken as the input port and "po" as the output port. The port "po" can be connected to none, 1 or 2 other ports; in case of more connections EcosimPro will issue an error (port pi has no limits). If the port requires a constructor parameter initialization, they are passed as arguments:

```
COMPONENT ChemicalReactor(SET_OF(Chemical) mix)
     PORTS
          IN fluid fin(mix)
          OUT fluid fout(mix)
```

7.6. Categories of Variables

The fundamental variables in the components fall into three categories:

- In the Parameter block there are construction parameters used only to instantiate the component (public)
- In the DATA block there are component variables which are normally known (public)
- In the DECLS block there are variables local to the component (private)

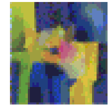
The first two are used to define the initial state of the component, and DECLS defines variables local to the component.

7.6.1. Construction Parameters

Construction parameters are passed for creation of the component or port whose values do not have to be given when defining it, but which are necessary when it is used. When a parameterised port or component is instantiated, the value of the construction parameters must be specified and they cannot be changed during the simulation. One of the ways in which they differ from DATA is that DATA can change its value during an experiment, but construction parameters never do. Construction parameters behave as initialization constants of the component (without the values of which it cannot be generated).

The BNF grammar for this block is as follow:

```
param_def_s : '(' param_def ( ',' param_def )*  ')'

param_def: data_type id ( '=' initVal )? (UNITS STRING_VALUE)? STRING_VALUE?
         enum_set_decl  '='  '{' enum_elem_s '}' STRING_VALUE?
```



It allows to declare parameters (data_type can be either REAL or INTEGER or STRING or FILEPATH or ENUMERATIVE or BOOLEAN) with an initial value, units (in STRING format) and description. As well is allowed to declare SET_OF variables with an initial value and description.

The construction parameter formats are passed as arguments of the port/component; they don't have a special block. For instance:

```
COMPONENT mixCompo(INTEGER  n)
```

When the component mixCompo is instantiated, the syntax to provide the construction parameters is:

```
mixCompo (6) myCompo
```

The construction parameters can have a default value; e.g.:

```
COMPONENT mixCompo(INTEGER  n= 4)
```

If the user does not specify the value of n, the system will take 4.

They are normally used to:

- Pass sizes to arrays
- Pass flags for selecting equations
- Pass a SET_OF enumeration elements (e.g., chemical compounds)
- For example:

```
COMPONENT mixCompo(INTEGER n=4)
     DATA
          REAL v[n]
     …
END COMPONENT
```

The size of array “v” is “n”. When this component is used from another component it can be changed. For instance:

```
COMPONENT big
     TOPOLOGY
          MixCompo(n= 15) m1
     …
END COMPONENT
```

This example will use an array “v” of 15 positions.

7.6.2. DATA Block

DATA variables are component data which are normally known; ie, their value does not change over time. For instance, the value of R for a resistance, the diameter of a pipe, etc. Variables declared in the DATA block are initialized at the time of the declaration; they can also be initialized when the components are instantiated. They must always have a concrete value before starting the simulation, as it is assumed that they define a concrete value.

The BNF grammar for this block is as follow:

```
data_s : (CONST? data_type general_id_s ( '=' init_expression )?
          (UNITS STRING_VALUE)? obj_range? STRING_VALUE?)*
```

It allows to declare data (REAL, INTEGER, STRING, FILEPATH, ENUMERATIVE or BOOLEAN) with an initial value,units (in STRING format), range of validity and description.

For example, for a component which reuses the pipe component, the DATA variables are initialized as follows:

```
COMPONENT FluidCircuit
     TOPOLOGY
          Pipe p1 (length = 0.5, diameter = 2)
```

It is not allowed to declare new Enumeration types in the DATA block.

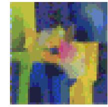
7.6.3. DECLS Block

The variables and enumeration types of any component are declared in the DECLS block. These are variables that have no connection to the outside universe and are normally used to calculate inside the component.

The BNF grammar for this block is as follow:

```
decl_s : ( (BOUND|EXPL|DISCR|ALG)? var_decl
         | ENUM IDENTIFIER '=' '{' EOL* enum_elem_s EOL* '}' STRING_VALUE?
         | enum_set_decl  '='  '{' EOL* enum_elem_s EOL* '}' STRING_VALUE?
         | CLOSE identifier_s ( '=' init_expression )?
var_decl: CONST? data_type id_s ( '=' initVal )? (UNITS STRING_VALUE)?
               obj_range? STRING_VALUE?
```

It allows to declare variables (REAL, INTEGER, STRING or BOOLEAN) with an initial value, units (in STRING format), range of validity and description.



Also it is allowed to define new enumerative and SET_OF types and CLOSE statements.

By default all REAL variables will be considered continuous (the other types are always discrete), the modeler can use the operator DISCR for declaring explicitely discrete a REAL variable. For the continuous variables (REAL variables without the qualifier DISCR) the modeler can use the following operators:

- BOUND: in case of having more variables than equations suggest to use this variable as boundary.
- EXPL: in case of algebraic loops, try to find this variable explicitely (not as algebraic).
- ALG: in case of algebraic loops, try to use this variable as algebraic.

In all cases the operators are suggestion for the sorting algorithms.

The main difference between the DATA and DECLS blocks is that DECLS variables do not assume a fixed value; rather, they are calculated in the component. Although constants can be defined, they are only local and cannot be configured from outside, unlike DATA which can be. The allowed types of declaration are:

- Variable or array of REAL variables which calculates the value of the continuous part (must be REAL). For example

```
DECLS
     REAL v[3]
CONTINUOUS
          V[1]= v[2] / v[3] * 0.34
```

- Discrete variable types, such as REAL, INTEGER, BOOLEAN, STRING, enumeration and arrays (do not change value in continuous part of component). For example

```
DECLS
     INTEGER  numberOfCartridges
DISCRETE
     WHEN ( numberOfCartridges  == 0 )
```

- Constant values using the keyword CONST. For example

```
DECLS
     CONST REAL PI = 3.1415
```

- Declaration of enumeration types. For example

```
DECLS
     ENUM Chemicals = {CO2, H2O, O2}
```

- Sets of enumeration types. For example

```
    DECLS
         SET_OF(Chemicals) mix = {CO2, H2O}
```

- 1, 2 and 3 dimensional tables used locally. For example

```
    DECLS
         TABLE_1D tab = { { 0.0, 0.5, 1.0}, -- X values
                        {67.23, 4.83, -3.23}} -- Output
```

7.6.3.1. DISCR Operator

Variables which are calculated explicitly in EcosimPro and do not need to be calculated in EcosimPro's CONTINUOUS block are known as discrete variables (DISCRETE type). EcosimPro automatically takes some to be discrete, such as INTEGER, BOOLEAN, STRING and ENUM. If modellers want a REAL variable to behave as a discrete variable, then they must give it the qualifier DISCR. For example:

```
COMPONENT foo
DECLS
     DISCR REAL x  -- x will behave as discrete
     REAL y
     REAL z
INIT
     x= 9
     y=8
     z= 7
CONTINUOUS
     x= y + 5*z
     3*x - 2*y = 4*z - 88
END COMPONENT
```

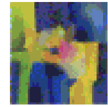
In this component, the variable "x" will be taken as discrete and it will not be calculated in the CONTINUOUS block. It will not even be taken into account as a boundary variable because it is the modeller who is responsible for calculating it (normally in the INIT block).

7.6.3.2. BOUND Operator

The user can specify that a variable will be a boundary in the final mathematical model. This is done by marking the declaration with the BOUND word, for instance:

```
BOUND REAL Temperature
```

It means that this variable will be selected as boundary condition for the final mathematical system.



This is done only when the user knows from the beginning that this variable should behave as boundary in all cases.

7.6.3.3. EXPL Operator

Sometimes we need to calculate a variable in an explicit format; ie, we don't allow the mathematical algorithms to transform the original equation to find other variables based on the same equation. To declare a variable as explicit, write:

```
EXPL REAL rho
```

Then the continuous part of the component would normally have an equation to find this variable; e.g.:

```
Rho = p / rgas
```

This equation will maintain this format and it will not be transformed symbolically into another (e.g., p = rho * rgas)

7.6.3.4. ALG Operator

Sometimes modellers can detect that an algebraic loop will probably occur and they can use the best algebraic variable to solve the loop. They can use the ALG qualifier to mark a variable as candidate to become an algebraic variable (only in case an algebraic loop is detected, otherwise it will be a regular variable). For example:

```
COMPONENT foo
DECLS
     ALG REAL x  -- candidate to be algebraic
     REAL y
     REAL z
CONTINUOUS
     x – y**2 – sin(z) = 98
     cos(z**2) + x = 3*y
     x – 5*z = 9
END COMPONENT
```

In this code the modeller suggests that the variable “x” be taken as algebraic. When EcosimPro tries to solve this equation system it detects that the variables cannot be calculated explicitly and that an algebraic loop is needed. EcosimPro will propose variable “x” as the first candidate to become an algebraic variable when creating a partition.

7.6.3.5. *Closing a variable to a final value*

There is the option to close any construction paramenter or datum in any child component with a concrete value. This is useful sometimes to fix a datum (or construction parameter) and not allow to change it anymore. For instance, given the Resistor componet

```
COMPONENT Resistor
DATA
     REAL R
DECLS
     REAL i, v
CONTINUOUS
     R=V*i
END COMPONENT
```

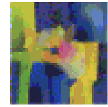
It is possible to create two customized child component fixing the R value, for instance:

```
COMPONENT Resistor5000
DECLS
     CLOSE R= 5000
END COMPONENT
COMPONENT Resistor80000
DECLS
     CLOSE R= 80000
END COMPONENT
```

From now on when a component Resistor5000 is instantiated the datum R has disappear from the public part and it is fixed internally to 5000.

Also CLOSE can be used to hide a port in a child component. For example:

```
PORT mport
    SUM REAL a
    EQUAL REAL b
END PORT

COMPONENT aba
PORTS
     IN mport pin
END COMPONENT

COMPONENT foo IS_A aba
DECLS
     CLOSE pin
END COMPONENT
```



In this example the port “pin” disappears from the public interface in foo component.

7.7. OBJECTS Block

The OBJECTS block allows instances of classes (described later) to be declared. The classes could be either internal classes defined in EL or external classes in C++. For example, given the following external class:

```
EXTERN CLASS mmath
  DECLS
           REAL value
  METHODS
           EXTERN METHOD REAL add(IN REAL a, IN REAL b)
           EXTERN METHOD REAL subs(IN REAL a, IN REAL b)
END CLASS INCLUDE "<mmath.h>" IN "testCpp.lib"
```

An example of usage of this class in a function is:

```
COMPONENT foo
DECLS
     REAL x
OBJECTS
     mmath m1
INIT
     m1.value= 4.5
     x= m1.add(2,2)
CONTINUOUS
     x= cos(TIME) + m1.add(3,4) - m1.value
END COMPONENT
```

In the example, the object m1 is the INIT and the CONTINUOUS block is either accessing an object variable or calling a method.

The BNF syntax for this block is:

```
object_s: (PRIVATE? scoped_id  id_s (STRING_VALUE?)*
```

It allows to declare a set of public (default) or private objects (using PRIVATE) with a description.

7.8. TOPOLOGY Block

The topology block can encompass four types of statements:

- Declaration of objects (instance names of other components; e.g., Tank t1(c= 9000))
- Statements which connect these objects (e.g., CONNECT t1.p1 TO t2.p2)
- EXPAND and EXPAND_BLOCK statements to make multiple connections in one go
- A statement which defines a unique path for the component (e.g., PATH s_in TO s_out)

There is no pre-established order for statements and they can be intercalated (there can only be one PATH statement!). This is explained in further detail below.

7.8.1. Use of Aggregated Components

Users writing in EL can use as many component instances as they wish to define a new component. The only restriction is that the instances used have to be from components which are already defined.

As many instance names of a component as required can be defined in a single line, for example:

```
Tank t1
Tank t2[3]
Tank t3, t4[chemicals,3]
```

However, if these objects are declared in the same line (e.g., t3 and t4), they must share the same construction parameter and data initiation. The construction parameters are entered in parentheses immediately after the component name and the data are entered at the end of the declaration of the last object. For example, assuming that the component “Tank” has two construction parameter integers denominated “n” and “k” and two real data denominated “c1” and “c2”, a valid initiation would be:

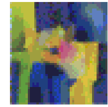
```
Tank(n = 4, k = 5) t3, t4[chemicals,3] (c1 = 4.5, c2 = 3.9)
```

We can also omit the names of the integers and data, and enter them in the order in which they were declared:

```
Tank(4,5) t3, t4[chemicals,3] (4.5, 3.9)
```

In this case, the component “Tank” will have to have been declared as:

```
COMPONENT Tank(INTEGER n= 2, INTEGER k= 3)
```



```
     DATA
          REAL c1 = 7.8
          REAL c2 = 8.9
     ...
END COMPONENT
```

If the user wishes to assign different values either to the construction parameters or to the data, he must declare them on separate lines. The following, for example, would be incorrect:

```
Tank(4,5) t3(3,5), t4[chemicals,3] (7,8)  -- ERROR
```

It would have to be entered as follows:

```
Tank(4,5) t3(3,5)
Tank(4,5) t4[chemicals,3]  (7,8)
```

Since Tank has default values for the construction parameters (c and k) and data (d1 and d2) it is possible to declare an instance without making any initialization and assuming the default values (c=2,k=3,d1=7.8,d2=8.9)

```
Tank t3
Tank t4[chemicals,3]
```

In general, it is recommended that each declaration be written on a separate line. This makes the code a lot clearer and easy to follow.

Remember: the construction parameters are set only once when defining the instances but they can never be changed during simulation, whereas the data are values assumed to be constant but they can be changed during simulation (for instance the diameter of a pipe).

7.8.2. Using EXPAND for Multiple Connections

EXPAND and EXPAND_BLOCK operands can be used to make multiple connections in one go. It can be used in nesting form. For example:

```
TOPOLOGY
   Tank t4[3,chemicals]
   Tank t5[3,chemicals]

   EXPAND (I IN 1,3)
     EXPAND(J IN chemicals)
       CONNECT t4[I,J].pout TO t5[I,J].pin
```

The EXPAND_BLOCK operand can also be used for making multiple connections. For example:

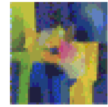
```
COMPONENT foo (SET_OF(chemicals) mix, INTEGER N)
PORTS
   IN Fluid pin[N]
   OUT Fluid pout[N]
TOPOLOGY
   Tank t4[N,mix]
   Tank t5[N,mix]

   EXPAND (I IN 1,N)
     EXPAND_BLOCK (J IN mix)
       CONNECT pin[I] TO t4[I,J].pin
       CONNECT t4[I,J].pout TO t5[I,J].pin
       CONNECT t5[I,J].pout TO pout[I]
     END EXPAND_BLOCK
```

In this example we can see that two construction parameters, mix and N, have been used to parameterise both the port arrays and the component arrays. Of course, it could also be done with constant values, for example:

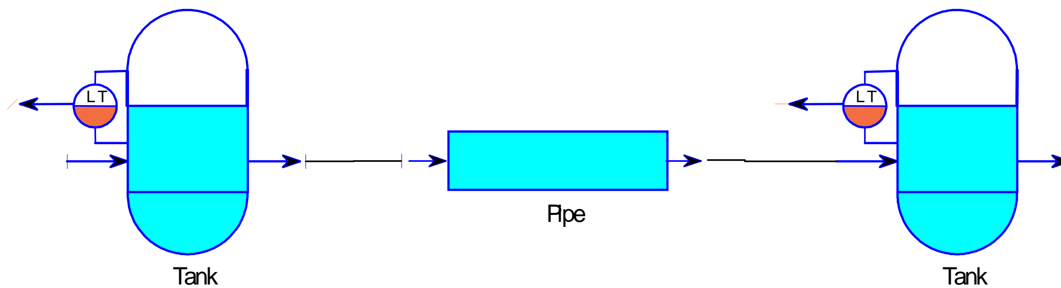
```
COMPONENT foo
PORTS
    IN Fluid pin[3]
    OUT Fluid pout[3]
TOPOLOGY
   Tank t4[3,chemicals]
   Tank t5[3,chemicals]

   EXPAND (I IN 1,3)
     EXPAND_BLOCK (J IN chemicals)
       CONNECT pin[I] TO t4[I,J].pin
       CONNECT t4[I,J].pout TO t5[I,J].pin
       CONNECT t5[I,J].pout TO pout[I]
     END EXPAND_BLOCK
END COMPONENT
```

7.8.3. PATH Statement

The PATH statement is used to connect the different objects declared, as seen in the previous point. In other words, we can use it to create the internal topology of a component. Typically, if we want to create a new component using a topological diagram of other objects, it will be done with this statement which, as we will see, is very easy to use.



As an example we will create a very simple topological component: a water tank connected to a pipe, the other end of which is connected to another tank. The equivalent graph would be as follows:



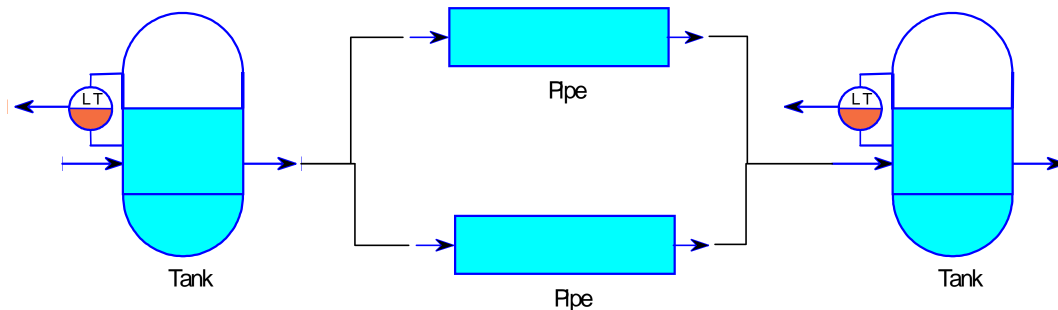
Let's assume that the definition of "Tank" includes the declaration of two ports and the pipe out:

```
COMPONENT Tank
     PORTS
          IN fluid f_in              "fluid input port"
          OUT fluid f_out            "fluid output port"
     …
END COMPONENT
COMPONENT Pipe
     PORTS
          IN fluid f_in              "fluid input port"
          OUT fluid f_out            "fluid output port"
     …
END COMPONENT
```

We can make the new component designated "balance" as follows:

```
COMPONENT balance
     TOPOLOGY
          Tank t1
          Tank t2
          Pipe p
     CONNECT t1.f_out TO p.f_in
     CONNECT p.f_out TO t2.f_in
END COMPONENT
```

When PATH is explained, we will see how these CONNECT statements could be simplified further. In addition, we could have connected the output of tank 1 to two pipes and the output of these two pipes could have been connected to the input of the second tank:



In this case the CONNECT statement makes life easier for us; we can make numerous connections in the same statement, as follows:

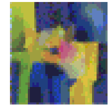
```
COMPONENT balance
     TOPOLOGY
          Tank t1
          Tank t2
          Pipe p1
          Pipe p2
     CONNECT t1.f_out TO p1.f_in, p2.f_in
     CONNECT p1.f_out, p2.f_out TO t2.f_in
END COMPONENT
```

We can see that with the use of commas, we have connected the output of the first tank to the input of the two pipes, and we have connected the output from the two pipes to the input to the second tank.

The PATH statement is used to simplify the connections. It allows us to create a default path for the component so that if no port is specified for connection, it is understood that the port declared in the PATH will be used.

For example, in the first of the previous examples, and assuming we have defined a PATH for the tank and the pipe, we have:

```
COMPONENT Tank
     PORTS
          IN fluid f_in              "fluid input port"
          OUT fluid f_out            "fluid output port"
     TOPOLOGY
          PATH t_in TO f_out         -- defines a path from the
input
                                     -- to the output port
     …
END COMPONENT
COMPONENT Pipe
```



```
     PORTS
          IN fluid f_in        "fluid input port"
          OUT fluid f_out      "fluid output port"
     TOPOLOGY
          PATH t_in TO f_out        -- defines a path from the
input
                                    -- to the output port
     …
END COMPONENT
```

Now the component “balance” could have been written as:

```
COMPONENT balance
     TOPOLOGY
          Tank t1
          Tank t2
          Pipe p
     CONNECT t1.f_out TO p TO t2.f_in
END COMPONENT
```

In a single CONNECT sentence we have now defined the topology. Internally the system understands that as we have not defined ports as “p”, we are using the port declared in the PATH.

If the component has only one port, a “special” PATH can be defined for it. The following is an example of an earth:

```
COMPONENT G
     PORTS
          IN Elec e_p
     TOPOLOGY
          PATH e_p TO e_p
     CONTINUOUS
          e_p.v = 0
END COMPONENT
```

Now the user can use only the instance name of this component to connect it. For example:

```
Earth e
     CONNECT e TO d1.e_p
```

7.9. INIT Block

A component may require an initialization block which specifies a series of sequential instructions to be executed before starting the simulation. It is used to initialize some variables which require an initial value before they start the simulation.

The INIT block can contain only sequential statements (IF, FOR, etc). Obviously all the statements are executed sequentially, as in the case of a classical program language.

For example:

```
COMPONENT test
     DECLS
          REAL x
          REAL y
          REAL v[3]
     INIT
          x = 9.8
             -- y = v[1] + v[2] + v[3]
          y = SUM(i IN 1,3; v[i])
             -- v[1]=0,  v[2]=0 and v[3]=0
          FOR(i IN 1,3)
               v[i] = 0
          END FOR
     …
END COMPONENT
```

7.9.1. Priorities for INIT Blocks

Priorities can be assigned to INIT blocks in order to be executed. By default, the priority of an INIT block is zero and they follow a set of general rules or execution. To change the priority, the user can use the clause “PRIORITY value” after the word INIT, where value can be any positive or negative integer number. The following statements are valid:

```
INIT PRIORITY 100           -- The INIT block has a priority of 100
INIT PRIORITY -40           -- The INIT block has a priority of -40
INIT                        -- The INIT block has a priority of 0
```

When the final INIT blocks of all the components are sorted, the following sorting rules are followed:

- First, they are ordered by priority

- If the priorities are equal, the INITs of parent components go first, then those of the final components, and lastly those of the aggregated components

For example, given the following example:

```
COMPONENT Parent
DECLS
      REAL x1
INIT
      x1= 1
END COMPONENT

COMPONENT Compo1
DECLS
      REAL x2
INIT
      x2= 2
END COMPONENT

COMPONENT Compo2
DECLS
      REAL x3
INIT PRIORITY 100  -- changed the priority to 100
      x3= 3
END COMPONENT

COMPONENT Final IS_A Parent
DECLS
      REAL x4
TOPOLOGY
      Compo1 obj1
      Compo2 obj2
INIT
      x4= 4
END COMPONENT
```

When a partition is made and the INIT blocks are sorted, it will be done as follows:

```
--init(obj2.Compo2,100)
obj2.x3 = 3
-- init(Parent,0)
x1 = 1
-- init(Final,0)
x4 = 4
-- init(obj1.Compo1,0)
obj1.x2 = 2
```

As can be seen, the first one is from Compo2, as it has the highest priority (100). The rest have been sorted following the general rules: first the parent component, then the final component, and lastly the aggregate component with the same priority (all with priority 0).

7.10. DISCRETE Block

All the discrete modelling parts of the component go into the DISCRETE block. Discrete information consists of events which must be detected during the simulation. Once an event is detected, the system executes an associated sequence of actions which can in turn activate new events in a chain reaction. When there are no more events, the system carries on with the continuous part if it exists; otherwise it halts.

Only discrete statements can be used in this part (see Chapter 3). The most important discrete statement is the WHEN statement.

As already mentioned above, you can use EcosimPro for pure discrete modelling; ie, with no associated continuous equations. The type of application decides whether you use it or not. Normally, however, the discrete part of the components will establish events which affect the continuous part. These events are usually logical conditions on continuous or discrete variables. When the conditions are met, the continuous part halts and a series of instructions are executed. As these instructions usually have an effect on the variables of the continuous model, you will have to reinitialize it once the discrete events have been executed.

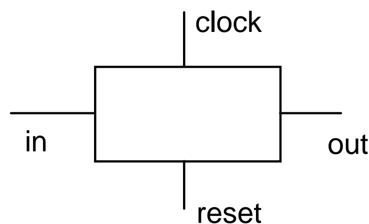
Discrete events are evaluated at each instant in time that the continuous part advances. When a condition is met, the associated actions are executed. If one of these actions gives rise to another discrete event, it is in turn dealt with until the are no more left, at which point the system resumes with the continuous part.

Example 1: Represent an AND logic gate. It is defined as an EL component with two inputs and one output. When both inputs are TRUE, the output is TRUE, otherwise it is FALSE:

```
   -- defines a simple digital port, using a unique boolean
   -- variable
PORT digitalPort SINGLE IN
         -- when the port is OUTPUT, the signal should be equal in
         -- all the connections
      EQUAL OUT BOOLEAN s
END PORT
   -- AND logical port
COMPONENT Gate_AND
      PORTS
            IN digitalPort in1           "input port 1"
            IN digitalPort in2           "input port 2"
```

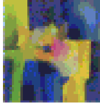
```
        OUT digitalPort out     "output port"
  DISCRETE
     -- detect event, when both are TRUE, changes the output
        WHEN (in1.s == TRUE AND in2.s == TRUE) THEN
              out.s = TRUE             -- output is TRUE
        END WHEN
     -- detect event, when any input is FALSE
        WHEN  (in1.s == FALSE OR in2.s == FALSE) THEN
              out.s = FALSE            -- output is FALSE
        END WHEN
END COMPONENT
```

Example 2: A T_flip_flop that inverts an input signal when the clock is on the falling edge. Like a real flip-flop, the output signal is generated with a delay. Introduce a reset signal as well to change the output to FALSE. The graphical representation is:



In EL this is:

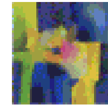
```
COMPONENT T_flip_flop (REAL delayTime) -- Delay typical for device
      PORTS
            IN digitalPort pi         "Input port"
            OUT digitalPort po              "Output port"
            IN digitalPort clock      "Clock signal input"
            IN digitalPort reset      "Reset signal input"
      DISCRETE
      -- when clock is in falling edge, generates an output
      -- with a delay (if reset is not pressed)
            WHEN (clock.s == FALSE) THEN
                  IF (NOT reset.s) THEN
                        po.s = NOT pi.s AFTER delayTime
                  END IF
            END WHEN
      -- if reset is pressed, changes output to FALSE inmediately
            WHEN (reset.s == TRUE) THEN
                  po.s = FALSE AFTER 0
            END WHEN
END COMPONENT
```

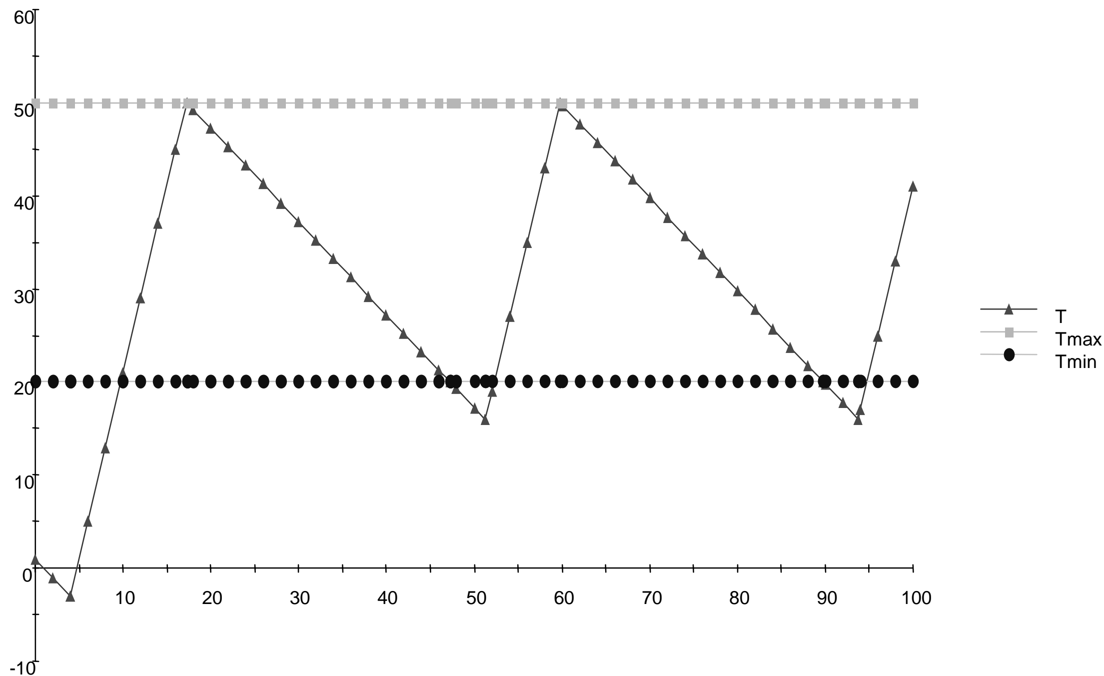


Example 3: Create a component in EL to regulate the temperature in a room. When the temperature is lower than the minimum Tmin, connect the heater after some time (4 seconds); when the temperature is greater than Tmax, disconnect the heater.

```
COMPONENT WHEN_Example
     DATA
         REAL Tmin = 20    "Lower limit temperature"
         REAL Tmax = 50.   "Upper limit temperature"
     DECLS
         BOOLEAN HeaterON  "heater is connected/disconnected"
         REAL HeaterPower  "Heater power"
         REAL T = 10.      "current temperature"
     DISCRETE
       -- when T is below Tmin, activate the heater after a delay
         WHEN (T < Tmin) THEN
             HeaterON = TRUE
             HeaterPower = 50. AFTER 4.
         END WHEN
       -- when T is greater than Tmax, disconnect the heater
         WHEN (T > Tmax) THEN
             HeaterON = FALSE
             HeaterPower = 0.
         END WHEN
      CONTINUOUS
                  -- defines a law for the heater
          T' = 0.1 * (HeaterPower - 10)
END COMPONENT
```

When we compile and simulate this component in EcosimPro, we get the following graph showing all the temperatures:





7.11. CONTINUOUS Block

The CONTINUOUS block contains the continuous part of the component. You can only use continuous statements in this block (see chapter 3). Here is where you define the differential-algebraic equations and insert permitted statements like ZONE and EXPAND.

The mathematical problem passed to the EcosimPro numerical solver can be stated as:

Explicit algebraic equations:

xi = Gi (t, y1 , y2 ,...,yn , x1 , x2 ,..., xi-1 , p1 , p2 , ..., pl) i= 1, 2 ,..., m

Implicit differential/algebraic equations, also called semi-state equations:

$$F_{j_1}\left(t, y_1, y_2, ..., y_n, \frac{dy_1}{dt}, \frac{dy_2}{dt}, ..., \frac{dy_n}{dt}, x_1, x_2, ..., x_m, p_1, p_2, ..., p_t\right) = 0 \qquad j = 1,2,...,n$$

where:

- **xi** are the explicit algebraic variables
- **yi** are the semi-state variables (those whose derivative appears in the mathematical formulation) plus the implicit algebraic variables (those whose derivative does not appear in the mathematical formulation and whose value cannot be explicitly calculated)
- The set of explicit variables **xi** plus the set of semi-state variables **yi** is equal to the complete set of unknown variables zi
- **t** is the time or independent variable
- **m** is the number of explicit equations
- **n** is the number of implicit equations (**k=m+n**)
- **pl** are parameters (non-explicit variables)

The y variables which are algebraic unknowns can be exchanged with the p parameters in the experiment.

The variables which are calculated in the equations are defined as continuous because their value follows the pattern given by the mathematical equation, as opposed to changing sporadically. The following are some examples of components with continuous parts:

Example 1: Define an ideal diode in EL. It conducts when the voltage is greater than zero and current is not lower than or equal to zero

We need an input port and an output port from type "elec", and we need to define a conditional equation in the CONTINUOUS block for the voltage. The final modelling code is:

```
   -- electrical port
PORT elec
     SUM REAL i      "current (Amperes)"
     EQUAL REAL v    "voltage (Volts)"
END PORT
   -- a theoretical electric diode
COMPONENT diode
     PORTS
          IN elec pi    "input port"
          OUT elec po   "output port"
     DATA
          REAL rlow = 0.1    "Low resistance (Ohms)"
          REAL rhigh = 1.e6    "Very high resistance (Ohms)"
     DECLS
```

```
          REAL v    "Voltage difference  (Volts)"
     CONTINUOUS
          v = pi.v - po.v
          0 = ZONE (v > 0 AND NOT pi.i <= 0) v - pi.i * rlow
               OTHERS v - pi.i * rhigh
          pi.i = po.i
END COMPONENT
```

If "(v > 0 AND NOT pi.i <= 0)" evaluates to TRUE, the equation would be as follows:

```
0 = v - pi.i * low
```

and if it evaluates to FALSE:

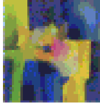
```
0 = v - pi.i * rhigh
```

This condition is evaluated at each instant of the integration (as shown in the above diagram). When a different branch is entered, the model changes and reinitializes automatically. The user does not have to be concerned about this as EcosimPro handles it internally.

All the equations from the CONTINUOUS block will be sorted properly in order to solve the continuous modelling based on those equations. This work is done internally and it is one of the most outstanding capabilities of the tool.

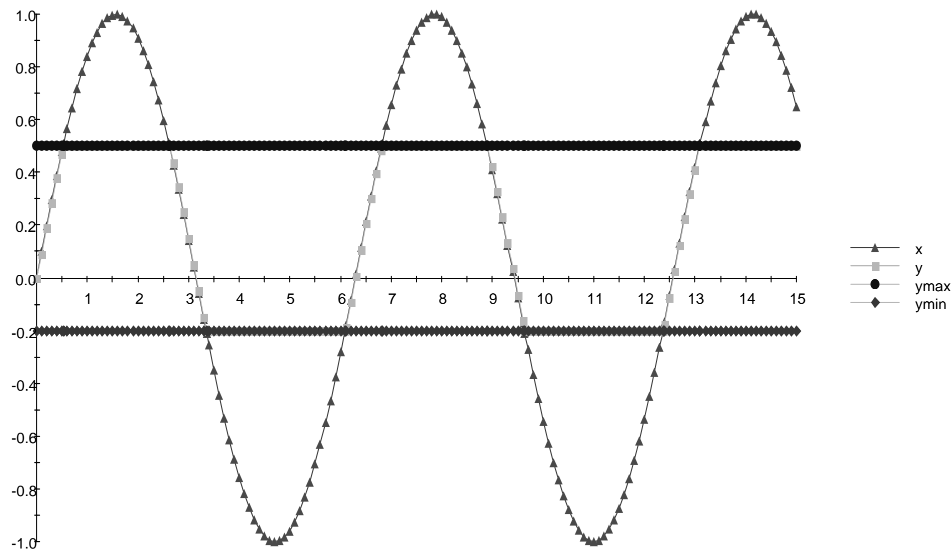
Example 2 creates an EL component with a variable going up and down until certain limits. The law this variable follows is that its first derivative should vary when any of the above are reached or when they fall within a certain margin. We can model this component as follows:

```
COMPONENT limits
    DATA
        REAL ymax = 0.5
        REAL ymin = -0.2
        REAL tau = 0.01
    DECLS
        REAL dy
        REAL x
        REAL y
    CONTINUOUS
        x = sin(TIME)
        dy = (x - y) / tau
        y' = ZONE (y > ymax AND dy > 0) 0.
             ZONE (y < ymin   AND dy < 0) 0.
             OTHERS   dy
END COMPONENT
```



EA Internacional

When we simulate this component in EcosimPro we obtain the following plot for variables y, ymax and ymin.

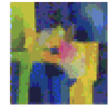


We can see how the y variable follows the x variable, but when it reaches the upper and lower limits it stops and follows the limits.

7.11.1. Virtual Equations

The equations in the continuous part can be labelled and later replaced by others when you create an inherited component from a parent component. For example, you may want to create a new component based on an existing one in which some of the continuous equations are not the same; in this case the new equation would have the same label as the one to be replaced. Labels are just equation identifiers. For example, a parent component has several equations and a child component is created:

```
COMPONENT baseComponent
     DECLS
          REAL x, y, z
     CONTINUOUS
          z' = 8 * x
          <eqn1> x = y / 0.3
END COMPONENT
COMPONENT derivedComponent IS_A baseComponent
     CONTINUOUS
          <:eqn1> x = y / 0.4
END COMPONENT
```



In this case, when the component derivedComponent is instantiated, the system would use this equation:

```
x = y / 0.4
```

and reject the parent equation. The syntax <Name> is used to refer to the original label and <:Name> to indicate that we want to replace the equation with this name inherited from previous components.

The overriding mechanism gives precedence to the closest equation in the line of inheritance. For example, if you create a new component inherited from derivedComponent like this :

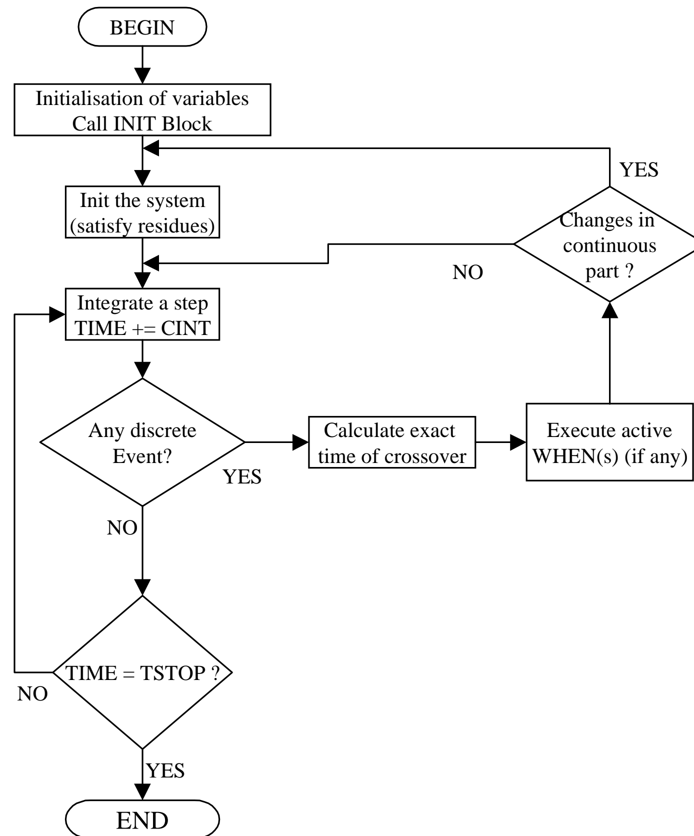
```
COMPONENT finalComponent IS_A derivedComponent
     CONTINUOUS
          <eqn2> t = z + y
END COMPONENT
```

This new component would take eqn1 from derivedComponent and not from the parent and the actual eqn2.

This mechanism allows you to reuse existing components by means of inheritance but modifying certain physical equations which are different in the child component; otherwise you would have to create two practically identical components with minor differences.

7.12. Continuous and Discrete Modelling

Iteration of the continuous and discrete modelling constitutes the main core of EcosimPro. The flow in this process is:



To detect crossover conditions, WHEN statements are used in the discrete part and ZONE statements in the continuous part. Every condition which uses continuous variables in one of these statements generates a crossover condition.

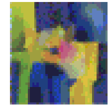
The diagram illustrates how the continuous process is stopped by the discrete events every time a new cross condition from any WHEN or ZONE statement is detected.

When the model has many discrete events the simulation will run slowly because EcosimPro will need to reinitiate the system and this takes time.

7.13. Advanced Modelling

7.13.1. INVERSE Statement

The modeller will sometimes want to introduce aids for the symbolic equation-handling algorithms. This is especially true when the modeller knows beforehand that if EcosimPro finds a variable in an equation, it could give rise to numeric problems. To help the tool, the user can employ the INVERSE statement in the following way:



```
INVERSE(var)  equation
```

If the mathematical algorithms find the variable “var” in an equation, this statement will discard the original equation and substitute it for “equation”. It is the modeller who is ultimately responsible for ensuring that the equation is well formulated and obtains the value of “var” correctly.

EcosimPro may not try to find that variable, in which case the INVERSE statement is not taken into account. If it does try to find it and the variable “var” is within a function, EcosimPro may or may not know how to find it. Let’s take a look at an example:

```
x = sin(y)
```

In this case, if EcosimPro has to find “y” it will know how to do it because it knows the inverse trigonometrical functions and would transform it into:

```
y = asin(x)
```

However, if it happens to be a user function (defined in EL, C++ or FORTRAN), EcosimPro would not know how to calculate the inverse function and would resolve the equation by iteration and create a residual. For example:

```
x = myFun(y)
```

would be transformed into:

```
0 = x - myFun(y)
```

However, this will sometimes bring about big numeric problems for the solver, which is why the user is provided with the INVERSE statement. In this case, the user could specify:

```
x = myFun(y)
INVERSE(y) y = evaluateY(x)
```

This means that if EcosimPro wants to find “y” it takes the equation "y = evaluateY(x)" and forgets about "x = myFun(y)". In this way, the user becomes responsible for calculating "y" correctly.

As mentioned before, if nothing is indicated the equation will be solved by iteration and this will give rise to underperformance and numeric problems dependent upon initialisation. In general, if we see that this could lead to numeric problems and know the inverse function, we will put it into the code, writing the alternative equation for when such variable is to be calculated. Here is another example with two variables:

```
x - usrfun(y,z) = 0
INVERSE(y) y = inv_y_uf(z, x)
INVERSE(z) calc_z_as_OUT_argument(x, y, z)
```

- If x is calculated, EcosimPro will generate the equation: x = usrfun(y, z)
- If y is calculated, EcosimPro will generate the equation: y = inv_y_uf(z, x)
- If z is calculated, EcosimPro will call the argument to the function: calc_z_as_OUT_argument(x, y, z)

Remember: modellers assume complete responsibility for the INVERSE statements they write. EcosimPro will limit itself to copying the INVERSE statement as it is, without any consistency checking. Dependencies are taken from the main statement. INVERSE statements are ignored during mathematical processing until final code generation.

INVERSE can also be used to resolve numeric problems in which functions do not intervene. Let's look at a physical example. We take the equation:

```
E = w * T
```

where E is the flow of energy, w is the volume and T is the temperature. If the temperature has to be calculated, the equation which would be generated would be:

```
T = E / w
```

This equation would give rise to numeric problems when the flow begins to approach zero. We may assume it to be correct that the temperature remains constant when there is very little flow. Assuming this is valid, we could write the equation like this:

```
E = w * T
INVERSE(T) divx(E, w, T)
```

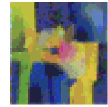
where the function divx could be implemented in the following way:

```
FUNCTION NO_TYPE divx(REAL x, REAL y, OUT REAL z)
     BODY
          IF( abs(y) > 1e-7 ) THEN
               z = x / y
          END IF
END FUNCTION
```

In this case, when EcosimPro wants to find “T” it will use the function given in the INVERSE statement and there will be no problems when the flow reaches zero.

7.13.2. IMPL Operator

The IMPL operator is used to guide the method of calculating the subsystems of the coupled equations. If a continuous statement with the IMPL flag is coupled to other statements it will be transformed into an implicit equation.



The user must specify a variable next to the IMPL flag. If this variable forms part of the set of variables coupled to the subsystem, that variable is selected as the algebraic variable. For example:

```
IMPL(x) x + y = 3*z
```

In this declaration, the user specifies that if this equation is coupled with other equations, it will be transformed into an implicit form, using x as the algebraic variable.

The use of this operator is not recommended unless absolutely necessary, or unless you are very sure of the mathematical implications; ie, sensitivity of the selected statement to perturbations, sensitivity of the equation subsystem to perturbations of the algebraic variable, validity range of the algebraic variable, etc.

8. Functions

8.1. Introduction

Functions in EcosimPro are similar to those in other programming languages like Fortran or C. A function is a piece of code that carries out operations and optionally returns a value. Functions are used for a wide range of problems; for example:

- Evaluate a numerical formula and return a value
- Reuse tried and tested functions
- Encapsulate frequently used code

EcosimPro is supplied with a library of basic commonly-used functions, such as sin(x), cos(x), sqrt(x), etc. These functions can be used anywhere in EL where they are compatible (refer to Appendix B for a complete list of built-in functions).

Although you will write new functions in EL, if you have a library of tried and tested functions written in Fortran, C and C++, you can reuse them in EL.

Functions in EL must be defined before they are used. External functions written in Fortran, C or C++ must be predeclared.

The following example shows a statement from the continuous part of a component which uses different functions:

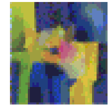
```
y = x * sqrt(x) - mySquare(x)
```

sqrt(x) is a function from the standard library

mySquare(x) is a user-defined function which returns the square of a number passed as an argument. The EL code for mySquare() can be:

```
FUNCTION REAL mySquare( REAL x ) "calculates a square"
     BODY
          RETURN x * x
END FUNCTION
```

The way to do this using C, C++ or Fortran is described later on.



8.2. Function Syntax

A function consists of an interface declaration, the definition of local variables, and the function body. The syntax is as follows:

```
FUNCTION data_type IDENTIFIER( argument-list ) STRING_VALUE?
EOL*
( DECLS Local-variables )?
( OBJECTS class_instace_stm_s  )?
BODY
     Sequential-stms
END FUNCTION
```

Function return types fall into the following categories:

- Standard types, such as REAL, INTEGER, STRING and BOOLEAN
- 1, 2 and 3 dimensional tables: TABLE_1D, TABLE_2D, TABLE_3D
- Function pointer: FUNC_PTR
- Enumeration, specifying the type
- NO_TYPE if the function doesn‘t return any value

The function name is specified in IDENTIFIER and must be unique to the library in which it is defined. A short description of the function can be written after the arguments.

8.2.1. Argument List

The argument list contains all the function's arguments, separated by commas. All declarations have the following syntax:

```
[IN | OUT] data-type IDENTIFIER (= init_value)? COMMENT?
```

If the argument has the IN prefix, this indicates that it is passed by value. OUT indicates that it is passed by reference and the value of the call variable will be updated.

For example:

```
FUNCTION NO_TYPE square(IN REAL x,OUT REAL y) "square example"
      BODY
            y = x * x
END FUNCTION
```

In the example, the argument x is passed by value and y by reference. The calculated value is returned in the second variable, y. By default, arguments are always IN and the keyword IN can be omitted. The function in the above example could have been declared as follows:

```
FUNCTION NO_TYPE square( REAL x, OUT REAL y) "square example"
```

Valid argument types are:

- Standard types, such as REAL, INTEGER, STRING, BOOLEAN
- 1, 2 and 3 dimensional tables: TABLE_1D, TABLE_2D, TABLE_3D
- Function pointer: FUNC_PTR
- Arrays
- Enumeration, specifying the type (e.g., ENUM Chemicals v)
- Sets of enumeration data, SET_OF

The modeller can provide default values for arguments like:

```
FUNCTION REAL add(IN REAL a= 2, IN REAL b=2)
BODY
     RETURN a+b
END FUNCTION
```

And then call a function such as:

```
add()
```

This would be equivalent to calling:

```
add(2,2)
```

The default values must be available from the right to the left like in C and C++. In other words, it is not correct to write:

```
FUNCTION REAL add(IN REAL a= 2, IN REAL b)
```

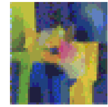
since if this function is called as:

```
add(3)
```

the compiler will not have a default value for “b”. But it is correct to express it as:

```
FUNCTION REAL add(IN REAL a, IN REAL b=2)
```

So for the call



```
add(3)
```

it is equivalent to write:

```
add(3,2)
```

It is not correct to leave an argument without a default value between two arguments with a default value. For instance, it is not correct to write:

```
FUNCTION fun1( IN REAL a, IN REAL b= 9, IN REAL c, IN REAL d= 8)
```

The argument c has no default value, but b and d do. This will produce a compilation error. We can assign a default value to the following combinations of arguments:

- only d
- d and c
- d, c and b
- d, c, b and a

If we want to provide a default value for OUT arguments, we must use the following standard names to be assigned as a default value:

```
DUMMY_REAL for REAL variables
DUMMY_INTEGER for INTEGER variables
DUMMY_BOOLEAN for BOOLEAN variables
DUMMY_STRING for STRING variables
```

For example, the following function is correct:

```
FUNCTION NO_TYPE  add(IN REAL a, IN REAL b, OUT REAL c=
DUMMY_REAL)
BODY
     c= a+b
END FUNCTION
```

The DUMMY_REAL initial value would be used if the user does not pass a valid variable name for the third argument; for example:

```
add(4,5)
```

it would be equivalent to call

```
add(4,5,DUMMY_REAL)
```

This call does nothing since the result is not stored in any model variable. Normally this calculated variable should be declared in a component, port, function, etc, as follows:

```
COMPONENT foo
DECLS
      REAL x
CONTINUOUS
      add(4,5,x)
END COMPONENT
```

In this example x is calculated in the add() function.

8.2.2. DECLS Block

The block DECLS is used to declare local variables. This section starts with the keyword DECLS. For example:

```
DECLS
      REAL x, y
      STRING st = "example"
      INTEGER i
      ENUM Chemicals v
```

This example defines the variables x and y as real, st as string, i as integer, and v as type Chemicals.

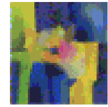
8.2.3. OBJECTS Block

The OBJECTS block allows us to declare instances of classes (described later). The classes could be either internal classes defined in EL or external classes in C++. For example, given the following external class:

```
EXTERN CLASS mmath
  DECLS
           REAL value
  METHODS
           EXTERN METHOD REAL add(IN REAL a, IN REAL b)
           EXTERN METHOD REAL subs(IN REAL a, IN REAL b)
END CLASS INCLUDE "<mmath.h>" IN "testCpp.lib"
```

An example of usage of this class in a function is:

```
FUNCTION REAL fun1(IN REAL a)
DECLS
      REAL x
OBJECTS
      mmath m1
```



```
BODY
   m1.value= 4.3
   x = m1.add(2,2)
   RETURN x
END COMPONENT
```

The object m1 has been declared in the OBJECTS block and it has been used in the INIT bloc to access a variable and call a method.

8.2.4. BODY Block

The body contains all the function's sequential instructions. The syntax is:

```
     BODY
        sequential-stms
```

All the sequential instructions defined in chapter 3 are allowed, including:

- Assignments
- Function calls
- IF-THEN-ELSE conditional statements
- FOR and WHILE loops
- Assertions with ASSERT statement
- RETURN statements to return control

For example, a simple function which calculates the greatest of two input arguments:

```
FUNCTION REAL max(REAL x, REAL y)
     BODY
          IF (x > y ) THEN
               RETURN x
          END IF
          RETURN y
END FUNCTION
```

8.3. Using External C, C++ and Fortran Functions

EL allows you to reuse existing functions written in C, C++ or Fortran. There are very powerful libraries available commercially which supply a wide range of mathematical

functions (optimisation, properties calculation, etc). Using these libraries or others you have developed yourself expands the power of the EL modelling language.

To use an external function from EL, you first have to pre-declare its interface. The only difference between using one language or another is in the header declaration:

- If the function has been compiled with a C++ compiler:

```
"C++" FUNCTION REAL mySquare(REAL x)
```

For example, the code for this function could be:

```
    double mySquare(double x)
    {
        return x * x;
    }
```

- If the function has been compiled with a C compiler, be aware that normally the compiler you use is a C++ compiler and hence this option will not be very common nowadays

```
"C" FUNCTION REAL mySquare(REAL x)
```

- If the function has been compiled in Fortran

```
"FORTRAN" FUNCTION REAL mySquare(REAL x)
```

- The previous function can be written in FORTRAN:

```
DOUBLE PRECISION FUNCTION mySquare(X)
     mySquare= X * X
     RETURN
END
```

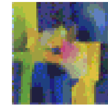
As explained in the section **Function Arguments**, you can use the modifiers IN and OUT to force variables to be passed by reference or by value. In Fortran, variables are always passed by reference.

Permitted types for external FORTRAN functions are limited: only REAL, INTEGER, STRING, FUNC_PTR and ARRAY. As return types are valid, the same types can be used, except the STRING type.

Before compiling, you need to specify the object libraries where your external functions are placed. The User Manual explains how to connect external object files.

If at coding time we know the location of the external library or object, we can add this information to the predeclaration using the following syntax:

```
"C++" FUNCTION REAL mySquare(REAL x) IN "c:\myLibs\lib1.lib"
```



This information is taken by EcosimPro and when generating an experiment it will use the library lib1.lib in directory c:\myLibs to link the final program. In any case, the user can select any other external library by selecting the option "External objects" from the EcosimPro GUI (see User Manual).

8.4. Special Functions

8.4.1. The delay() Function

EL can assign to variables the value of another variable with a delay. The delay() function returns historical values of a variable. For example:

```
x = delay(y,2)
```

This assigns to **x** the value of **y** two seconds ago; delay() is a predefined system function with the form:

```
"C++" REAL delay(REAL var,REAL delayTime )
```

Where **var** is the name of the variable whose historical value is required and **delayTime** is the time period. It returns the value of the time variable (TIME – delayTime), where TIME is the current time. The variable **var** must always be a dynamic variable declared in the DECLS block of a component. It cannot be an integer.

This function can be used ONLY inside component or port CONTINUOUS block, it cannot be used inside functions.

The following example shows a typical control component. The output signal will follow the input signal with a delay.

```
COMPONENT S_Delay
	PORTS
		IN analog_signal s_in		--inlet signal
		OUT analog_signal s_out		--outlet signal
	DATA
		REAL tdelay = 0.003
	CONTINUOUS
		s_out.signal = delay(s_in.signal, tdelay)
END COMPONENT
```

In this example the component S_Delay will output a signal which follows the input signal with a delay of 0.003 seconds.

8.4.2. Reading Tables from a File

EL allows external tables (1D, 2D and 3D) to be read from ASCII or XML files (see appendix dedicated to XML format for tables). The only difference is an integer argument that you pass to the function to read the table. The reading functions are:

```
INTEGER  readTable1D(STRING filename,
                     TABLE_1D t,
                     IN INTEGER format);
INTEGER  readTable2D(STRING filename,
                     TABLE_2D t,
                     IN INTEGER format);
INTEGER  readTable3D(STRING filename,
                     TABLE_3D t,
                     IN INTEGER format);
```

The first argument is the file name, the second is the table (it can be either TABLE_1D or TABLE_2D or TABLE_3D) and the third is an integer with two possible values: 1 means the file is in ASCII format and 2 means the file is in XML format.

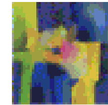
In case of using the XML format there are fields in the file not used for objects such as TABLE_1D, TABLE_2D and TABLE_3D but for a more generic table class named TABLE. Go to the specific chapter dedicated to TABLE class to understand all these fields.

The format of ASCII files for TABLE_1D is:

```
x 10
v
0.21556 1.03581
0.34721 1.10500
0.50444 1.21050
0.58864 1.28809
0.65517 1.34042
0.74116 1.40804
0.77724 1.46745
0.81336 1.52683
0.86094 1.59777
0.93168 1.69644
```

The format of XML files for TABLE_1D is:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE eds SYSTEM "DTDFILE_FULLPATH">
<table type="1D" name="PRsurge_vs_Wc_sec" description="Surge pressure ratio versus corrected
massflow rate" version="1.0" cdate="01/01/2007" mdate="01/08/2007" revision="0.1">
  <interp default="SPLINE" valid="{CONSTANT, LINEAR, SPLINE}" />
  <extrap default="LINEAR" valid="{FORBIDDEN, CONSTANT, LINEAR, SPLINE}" />
  <axis1 id="Wc" description="Corrected Massflow Rate (kg/sec)" value="{0.21556, 0.34721,
0.50444, 0.58864, 0.65517, 0.74116, 0.77724, 0.81336, 0.86094, 0.93168}" />
  <return id="PRsurge" description="Surge Pressure Ratio (-)" value="{1.03581, 1.10500, 1.21050,
1.28809, 1.34042, 1.40804, 1.46745, 1.52683, 1.59777, 1.69644}" />
  </table>
```



For TABLE_2D in ASCII format:

```
x 4
0.74000 0.80000 0.90000 1.00000

y 10
0.00000 0.11111 0.22222 0.33333 0.44444 0.55555 0.66666 0.7 0.8 1.00000

v
58.4751386 58.3129686 58.2048553 57.7029902 56.5690050 54.7078205 51.9223126 49.4833389
47.3870683 44.6423087
62.6509410 62.4921274 62.1264380 61.4629731 60.0702192 57.2192355 55.7118989 53.8956418
52.8212118 51.5587129
66.6014306 66.3955998 66.1897689 65.8167658 65.1240457 63.6205402 61.2686354 58.5144723
55.7686678 53.8137971
69.9783104 69.9166656 69.6279452 69.3085765 68.9376630 68.5361013 68.0582673 67.7347193
67.3167886 65.7631314
```

And for TABLE_2D in XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE eds SYSTEM "DTDFILE_FULLPATH">
<table type="2D" name="Wc_vs_NcRdes_BETA" description="Corrected mass flow versus rotational
speed and BETA parameter" versgion="1.0" cdate="02/03/2006" mdate="13/08/2007" revision="0.2">
        <interp default="LINEAR" valid="{CONSTANT, LINEAR, CUBIC, SPLINE}" />
        <extrap default="CONSTANT" valid="{FORBIDDEN, CONSTANT, LINEAR, CUBIC, SPLINE}" />
        <axis1,0.80,0.90,1.00}" />
        <axis2 id="BETA" description="BETA parameter (-)" value="{0.00,0.11,0.22,0.33,0.44,0.55,0.
66,0.7,0.8,1.00}" />
<return id="Wc" description="Corrected mass flow (kg/sec)"
value="{{58.47,58.31,58.20,57.70,56.56,54.70,51.922,49.48,47.38,44.64},
        {62.60,62.74,62.80,61.41,60.62,57.55,55.89,53.88,52.82,51.55},
{66.06,66.98,66.89,65.88,65.17,63.62,61.24,58.53,55.76,53.71},
{69.04,69.56,69.52,69.35,68.30,68.13,68.73,67.73,67.36,65.76}}" />
</table>
```

For TABLE_3D in ASCII format:

```
x 2
0.2 0.4

y 3
0.7 0.8 0.9

z 3
6.0 -2.0 -3

v
4.0 1.0 5.4
6.0 3.0 1.7
7.0 0.0 2
v
2.0 5.0 -34
3.0 7.0 -4
5.0 9.12e-3 567
```

And for TABLE_3D in XML format:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE eds SYSTEM "DTDFILE_FULLPATH">
   <table type="3D" name="mu_T" description="Dynamic Viscosity as a function of T, FARB and
WAR)">
      <interp default="LINEAR" valid="{CONSTANT, LINEAR, SPLINE}" />
      <extrap default="CONSTANT" valid="{FORBIDDEN, CONSTANT, LINEAR, SPLINE}" />
      <axis1 id="WAR" description="Water to Air Ratio (-)" value="{ 0.2,0.4 }" />
      <axis2 id="T" description="Temeperature(K)" value="{ 0.7,0.8,0.9}" />
```

```
    <axis3 id="FARB" description="Burnt Fuel to Air Ratio (-)" value="{6.0,-2.0,-3}" />
    <return id="mu" description="Dynamic Viscosity (Ns/m2)" value="{{
      {{4.0,1.0,5.4},{6.0,3.0,1.7},{7.0,0.0 ,2}},
      {{2.0,5.0,-34},{3.0,7.0,-4},{5.0,9.12e-3,567}}}" />
</table>
```

The following is an example of usage:

```
COMPONENT test
   DATA
      FILEPATH tableFileXml =   "tables/table1d.xml"
      TABLE_1D txml
   DECLS
REAL Rxml
   INIT
      readTable1D(tableFileXml,txml,2)  -- read from XML
   CONTINUOUS
      Rxml =     linearInterp1D(txml,TIME)
END COMPONENT
```

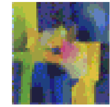
Note that we use the number 2 to specify that it is an XML format.

Other functions can be used to delete existing tables (typically used to read a new version of the tables) and to print tables, namely:

```
INTEGER clearTable1D(TABLE_1D t)
INTEGER clearTable2D(TABLE_2D t)
INTEGER clearTable3D(TABLE_3D t)
INTEGER printTable1D(TABLE_1D t,IN INTEGER format)
INTEGER printTable2D(TABLE_2D t,IN INTEGER format)
INTEGER printTable3D(TABLE_3D t,IN INTEGER format)
```

For them moment format can be only 1. The following is an example of usage:

```
COMPONENT test
   DATA
      FILEPATH tableFileXml =   "tables/table1d.xml"
      FILEPATH tableFileXml2 =  "tables/table1d.xml"
      TABLE_1D txml
   DECLS
REAL Rxml
   INIT
      readTable1D(tableFileXml,txml,2)
      printTable1D(txml,1)
   DISCRETE
WHEN ( TIME > 20 ) THEN
           clearTable1D(txml)   -- delete the existing table
     readTable1D(tableFileXml2,txml,2) -- read again
END WHEN
```



```
   CONTINUOUS
      Rxml =     linearInterp1D(txml,TIME)
END COMPONENT
```

8.4.2.1. Special function for reading tables from columns

The function readTableCols1D() enables a 1D table to be read using two columns of a file. An example is given below.

```
COMPONENT foo
DECLS
     TABLE_1D tab
     REAL x
INIT
     readTableCols1D("myTable.txt",4,8,tab)
     printTable1D (tab, 1)
CONTINUOUS
     x= linearInterp1D(tab,TIME)
END COMPONENT
```

Let's assume that there is a file named "myTable.txt" with the following contents:

```
95  1  1  0   10.2   0.005   16.3   84.0
95  1  1  1   10.9   0.005   16.4   82.6
95  1  1  2   11.0   0.005   15.8   86.3
95  1  1  3    7.7   0.004   13.9   90.0
95  1  1  4    4.9   0.004   13.3   91.6
95  1  1  5    6.0   0.004   13.5   91.7
95  1  1  6    5.7   0.004   12.4   88.4
95  1  1  7    7.4   0.006   10.6   80.9
95  1  1  8    7.8   0.089    9.6   78.9
95  1  1  9    9.2   0.302    9.8   76.2
```

The function call:

```
readTableCols1D("myTable.txt", 4, 8, tab)
```

loads the 4th column of the file in the independent variable of the table **tab1**, and the 8th column of the file in the dependent variable of the table **tab1**. The effect is as if we had declared tab1 in the following way

```
TABLE_1D tab1 = { {0,1,2,3,4,5,6,7,8,9},
                  {84.,82.6,86.3,90.,91.6,91.7,88.4,80.9,78.9,76.2}}
```

If the file path is absolute, it is important to ensure that the slash symbols in the pathname are doubled in the string argument. For example

```
readTableCols1D("C:\\DataFiles\\myTable.txt", 4, 8, tab1)
```

8.4.3. Interpolation in tables

8.4.3.1. Linear Interpolation in Tables

EL has built-in routines for linear interpolation in tables. The first argument to these routines is the table, followed by the input values to the table. They return the interpolated value. Each of these functions implements interpolation for one, two and three dimensional tables. The format is:

```
FUNCTION REAL linearInterp1D(TABLE_1D my1Dtable,
                             REAL vx )
FUNCTION REAL linearInterp2D(TABLE_2D my2Dtable,
                             REAL vx,
                             REAL vy )
FUNCTION REAL linearInterp3D(TABLE_3D my3Dtable,
                             REAL vx,
                             REAL vy,
                             REAL vz)
```

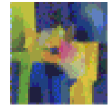
For example, using interpolation to calculate a variable:

```
x = linearInterp2D(my2Dtable, z, t)
```

where **my2Dtable** has been declared type TABLE_2D and z, y and t are numeric variables. This function returns the interpolated value at points z and t using **my2Dtable**.

The following example uses an interpolation function to calculate the value of the variable f_out.fh:

```
COMPONENT Inlet
     PORTS
          OUT fluid f_out
     DATA
          REAL D = 0.010  --Fluid outlet diameter (m)
             -- Define table for the Temperature
          TABLE_1D T = {{0., 100.}, {300., 300.}}
     CONTINUOUS
          f_out.fA = PI*D**2/4
          f_out.fh = linearInterp1D( T,TIME)
END COMPONENT
```



In very large tables it can be useful to begin the task of searching at the time it was tried the previous time. As a table may be used for several interpolations simultaneously, you can send additional position marker arguments. In this way, if a table is being interpolated in two well separated areas, you can do the task more efficiently by using two sets of position variables. The format of these functions is similar to the above, but there is an extra INTEGER argument which is used as a memory flag to start next time from a certain position. The value is changed internally by the interpolation functions and the change is visible outside (is OUT). The interpolation function should have as many integer flags as dimensions. The interfaces of these functions are:

```
FUNCTION REAL linearInterpHist1D(TABLE_1D my1Dtable,
                                  REAL x,
                                  OUT INTEGER pi)
FUNCTION REAL linearInterpHist2D(TABLE_2D  my2Dtable,
                                  REAL x,
                                  REAL y,
                                  OUT INTEGER  pi,
                                  OUT INTEGER  pj)
FUNCTION REAL linearInterpHist3D(TABLE_3D my3Dtable,
                                  REAL x,
                                  REAL y,
                                  REAL z,
                                  OUT INTEGER  pi,
                                  OUT INTEGER  pj,
                                  OUT INTEGER  pk)
```

For instance, in the above example, we can write:

```
DECLS
     INTEGER lastx = 0.
CONTINUOUS
     f_out.fh = linearInterpHist1D(my1Dtable, TIME, lastx)
```

This introduces an integer variable **"lastx"** to which the interpolation function assigns a value. When the next interpolation begins, this value can be used to begin the search close to where the previous one left off. If the table is very large (in this example it isn't), this can save a lot of interpolation time.

Interpolation with Splines in Tables

With EcosimPro the user has a tool which is also capable of interpolation with splines. The advantage of using spline interpolation instead of linear interpolation is that the final curve will be a lot smoother, because interpolation with splines will preserve the continuity of the first and second derivatives at the connecting points.

The following functions are available for this interpolation:

```
FUNCTION REAL splineInterp1D(TABLE_1D my1Dtable,
                             REAL x)
FUNCTION REAL splineInterp2D(TABLE_2D my2Dtable,
                             REAL x,
                             REAL y)
FUNCTION REAL invSplineInterp1D(TABLE_1D my1Dtable,
                             REAL val)
```

The first and second functions can be used for spline interpolation in arrays of one and two dimensions. They are equivalent to linear interpolation functions linearInterp1D() and linearInterp2D().

The invSplineInterp1D() function calculates the reverse of a one-dimensional interpolation with splines (splineInterp1D() function). The tool will sometimes transform it in an attempt to find a variable. For example, if a variable is determined as:

```
x = splineInterp1D(mtab, y)
```

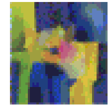
EcosimPro knows how to find “y”, which would give:

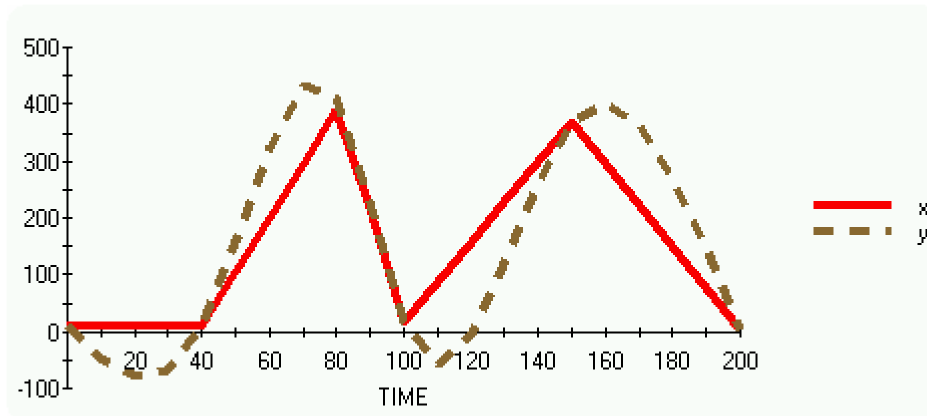
```
y = invSplineInterp1D(mtab, x)
```

Sometimes a model that uses linear interpolation does not converge owing to a sudden change in slope (first derivative) or in curvature radius (second derivative). The following component, for example, performs a linear interpolation for the “x” variable and a spline interpolation for the “y” variable. Both use the “TIME” variable as an input variable to the array:

```
COMPONENT test1
     DECLS
          REAL x,y
          TABLE_1D mtab = {{0.0, 40.0,  81.0, 100.0, 150.0,
200.},
                           {9.5,  9.5, 400.6,  16.6, 368.0,
3.}}
     CONTINUOUS
          x = linearInterp1D(mtab, TIME)
          y = splineInterp1D(mtab, TIME)
END COMPONENT
```

After compilation and integration up to 200 seconds we obtain:





The spline interpolation ("y" variable) is smoother.

Interpolations based on TIME Variable

EL also provides a family of time interpolation functions which use TIME as the input parameter in the table and avoid discontinuity problems between two intervals. The available functions are:

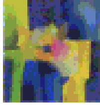
```
FUNCTION REAL timeTableStep(
                          REAL TIME,
                          TABLE_1D table)
FUNCTION REAL timeTableInterp(
                         REAL TIME,
                         TABLE_1D table)
FUNCTION REAL periodTimeTableInterp(
                         REAL TIME,
                         TABLE_1D table,
                         REAL period)
FUNCTION REAL periodTimeTableStep(
                         REAL TIME,
                         TABLE_1D table,
                         REAL period)
```

The function timeTableStep() returns the step position at time TIME from the table. It is not an interpolation but just the step value at that position. For instance given the table:

```
TABLE_1D myTable = { {0,100,200,300,400,500},{1, 3, 2, 4, 5, 1} }
```

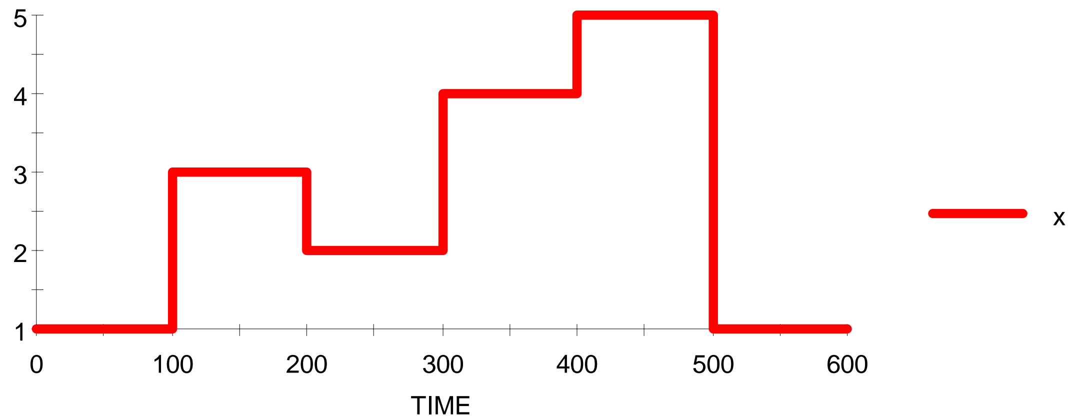
given the component:

```
COMPONENT test
DECLS
     REAL x
CONTINUOUS
```




```
    x= timeTableStep(TIME, myTable)
END COMPONENT
```

The value of x in simulation is:

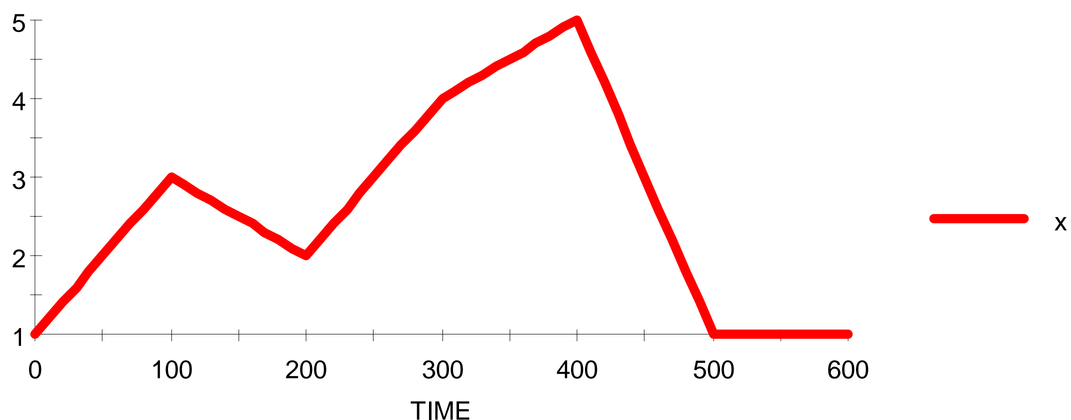


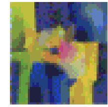
The value of x will be 2, since TIME is in interval [100,200] and the corresponding step value is 2. EcosimPro ensures that the discontinuity goes from x=2 to x=3.

The function timeTableInterp() returns the interpolated value at time TIME from the table. For instance changing the last component to use the new function:

```
x= timeTableInterp(TIME, myTable)
```

The output is:

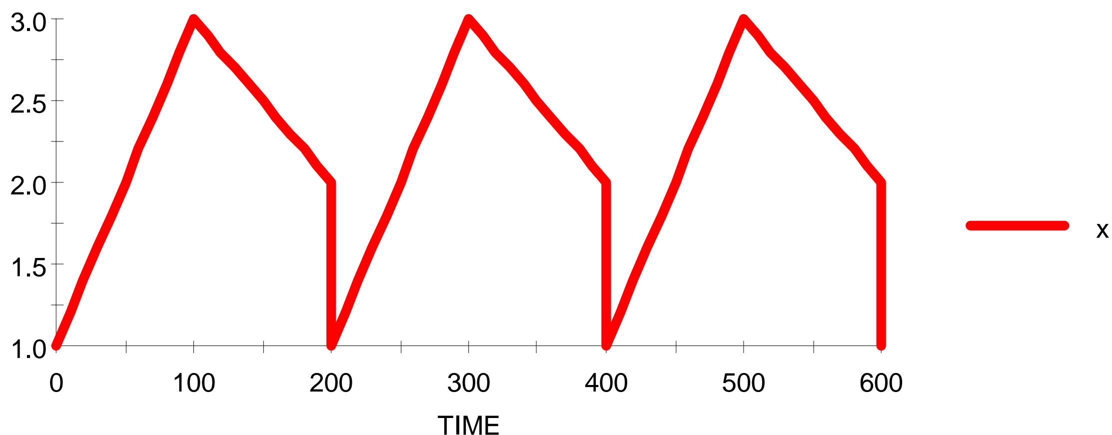




The function periodTimeTableInterp() is similar to timeTableInterp() but with a repeatable period from the table. For instance, changing the last component to use the new function:

```
x= periodTimeTableInterp(TIME, myTable,200)
```

The output is:

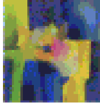


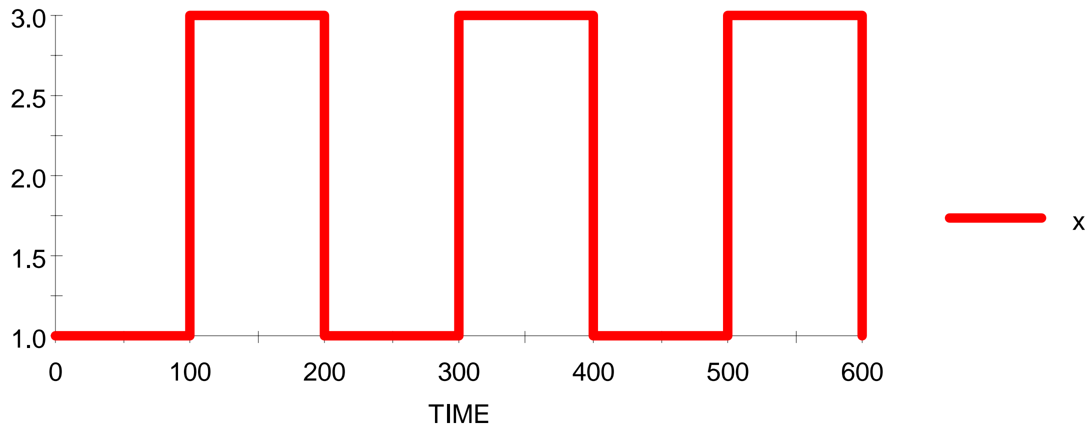
Since the period is 200, we can observe how there are three identical cycles corresponding to the values 1, 3, 2 which correspond to the first three segments of the table.

The function periodTimeTableStep() is similar to periodTimeTableInterp() but returning the step value for the table. For instance, changing the last component to use the new function:

```
x= periodTimeTableStep(TIME, myTable,200)
```

The output is:





In period [0,200], only the values 1 and 3 are returned in a periodical format.

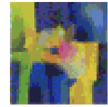
Advanced Interpolation on Table Intervals

EL provides another family of functions to detect which interval of the table a variable is in, and to interpolate in a specific interval. These functions allow the user to manually handle the discontinuities at the connection points between two intervals.

Important: the intervals begin with value 0 for the first interval, 1 for the second, and so on.

To detect what the actual interval is, the functions are:

```
FUNCTION BOOLEAN cellCrossing1D(
                              TABLE_1D tbl,
                              REAL x,
                              INTEGER pi,
                              OUT INTEGER i)
FUNCTION BOOLEAN cellCrossing2D(
                              TABLE_2D tbl,
                              REAL x,
                              REAL y,
                              INTEGER pi,
                              OUT INTEGER i,
                              INTEGER pj,
                              OUT INTEGER j)
FUNCTION BOOLEAN cellCrossing3D(
                              TABLE_3D tbl,
                              REAL x,
                              REAL y,
```



```
                              REAL z,
                              INTEGER pi,
                              OUT INTEGER i,
                              INTEGER pj,
                              OUT INTEGER j,
                              INTEGER pk,
                              OUT INTEGER k)
```

These functions return TRUE when the interval number of x is different from "pi","pj" and "pk" (depending on the dimension). The new interval is stored in "i", "j" and "k".

There is another family of functions to interpolate in a specific interval, namely:

```
FUNCTION REAL cellLinearInterp1D(
                              TABLE_1D  tbl,
                              REAL x,
                               INTEGER i )
FUNCTION REAL cellLinearInterp2D(
                              TABLE_2D  tbl,
                              REAL x,
                              REAL y,
                              IN INTEGER i,
                              INTEGER j )
FUNCTION REAL cellLinearInterp3D(
                              TABLE_3D  tbl,
                              REAL x,
                              REAL y,
                              REAL z,
                              INTEGER i,
                              INTEGER j,
                              INTEGER k )
```

These functions return the interpolated or extrapolated value of x, y and z with respect to the i-th, j-th and k-th interval (depending on dimensions) of the table tbl.

The following component is an example of how to manually handle the discontinuities in a table using both families of functions:

```
COMPONENT CompLinearInterp1D
    DATA
        TABLE_1D table
     REAL signal_in
     REAL signal_out
    DECLS
        INTEGER i, pi
    DISCRETE
        WHEN  cellCrossing1D(table, signal_in, pi, i) THEN
```

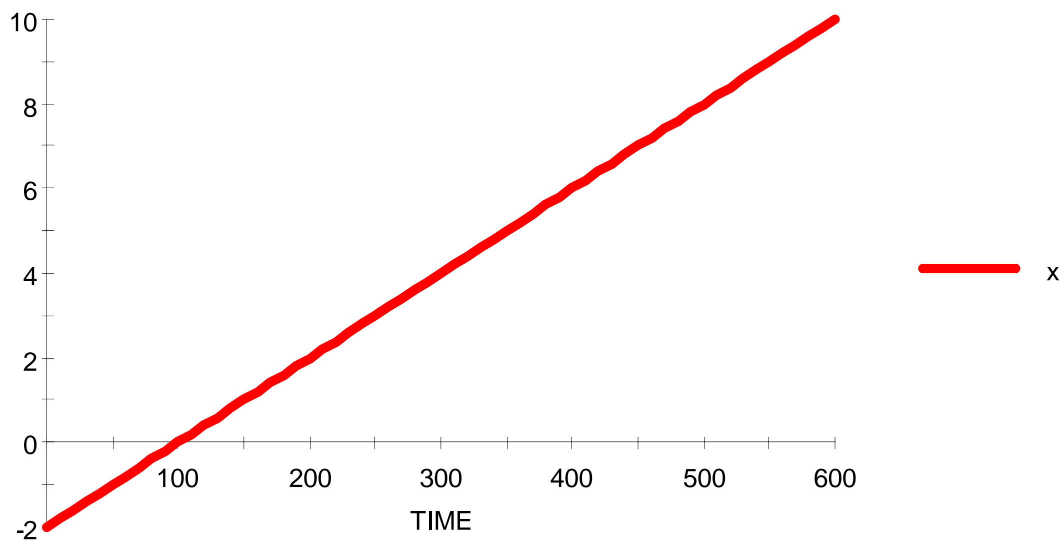
```
            pi = i
        END WHEN
    CONTINUOUS
        signal_out = cellLinearInterp1D(table, signal_in, pi)
END COMPONENT
```

The WHEN statement detects new changes of interval based on variable signal_in. When it is produced, it sets pi = i to save the current interval. This interval number is used to interpolate with cellLinearInterp1D in that specific interval pi.

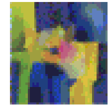
A more simple component could be:

```
COMPONENT test1
DECLS
     REAL x,y
CONTINUOUS
     x=  cellLinearInterp1D(myTable,TIME,2)
END COMPONENT
```

Simulating over 600 seconds, the output is:



Since the interpolation is made at interval 2, the range is [2,4] for TIME range [200,300]. We can see that the position at TIME= 200 is 2 and the position at TIME= 300 is 4. Other values outside of the range are simply extrapolated.



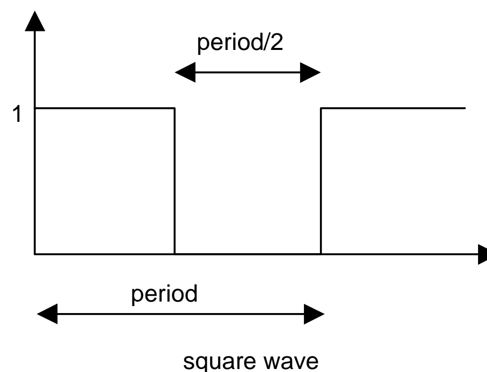
8.4.4. Waveform Functions

EcosimPro provides a set of functions normally used to generate inputs to simulation variables. These are very useful but they are often difficult to program, as they are discontinuous. Sometimes this discontinuity is not important or does not have much influence on the system of equations and can be ignored. In such functions the discontinuities are detected automatically by EcosimPro. Each discontinuity is handled properly to continue the simulation. The following functions have been implemented in the predefined EcosimPro libraries for use in models and experiments.

The first argument to all these functions is time. Generally, TIME is used, but you can also specify a phase lag over time, by using an expression like, for example, TIME+phase.

square() function

This function generates a square wave form, as shown in the following figure:



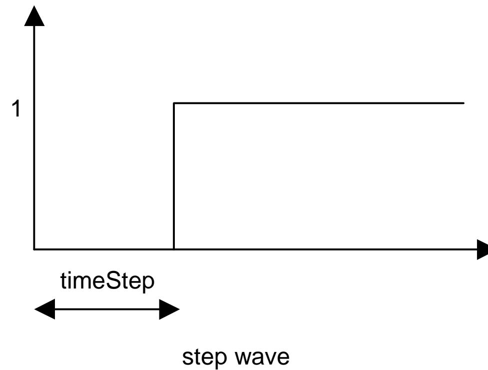
The function prototype is:

```
"C++" FUNCTION REAL square(REAL TIME,
                           REAL period)
```

Where TIME is the time of calculation and period is the desired period of change.

step() function

This function generates a step wave form, as shown in the following figure:



step wave

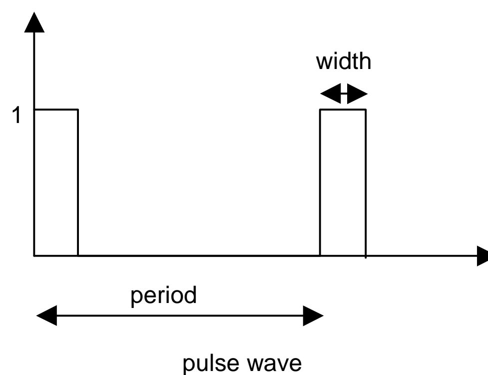
The function prototype is :

```
"C++" FUNCTION REAL step(REAL TIME,
                         REAL timeStep)
```

where TIME is the time of calculation and timeStep is the delay to generate the step.

8.4.4.2. pulse() function

This function generates a pulse wave form, as shown in the following figure:

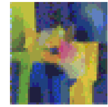


pulse wave

The function prototype is:

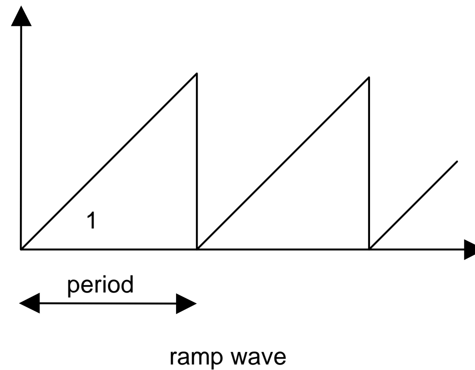
```
"C++" FUNCTION REAL pulse(REAL TIME,
                          REAL period,
                          REAL width)
```

where TIME is the time of calculation, period is the delay between pulses, and width is the size of the pulse.



8.4.4.3. ramp() function

This function generates a pulse wave form, as shown in the following figure:



ramp wave

The function prototype is:

```
"C++" FUNCTION REAL ramp(REAL TIME,
                          REAL period)
```

Where TIME is the time of calculation and period gives the width of the ramp.

8.4.5. LOG_STATUS() Function

Sometimes during the simulation we want to know the exact state of a simulation model. This can be done by calling the function LOG_STATUS with format:

```
LOG_STATUS("My Header")
```

The string "My header" is used only for heading the report. This report will be printed in the log file, but only if logging simulation is ON. All the variables, ZONE states and residues will be reported.

This function is normally used for debugging purposes when the system does not converge and the modeller wants to follow the previous state of the simulation.

8.4.6. Obtaining Array Dimensions during Runtime

There are functions in EcosimPro for calculating the dimensions of an array during the execution time. They are typically called from functions written in EL when they are passed as data array arguments. The functions available are as follows:

```
FUNCTION INTEGER dimArrayReal(OUT REAL v)
FUNCTION INTEGER sizeArrayReal(OUT REAL v,INTEGER code)
FUNCTION INTEGER dimArrayInt(OUT INTEGER v)
FUNCTION INTEGER sizeArrayInt(OUT INTEGER v,INTEGER code)
FUNCTION INTEGER dimArrayBool(OUT BOOLEAN v)
FUNCTION INTEGER sizeArrayBool(OUT BOOLEAN v,INTEGER code)
FUNCTION INTEGER dimArrayString(OUT STRING v)
FUNCTION INTEGER sizeArrayString(OUT STRING v,INTEGER code)
```

There are basically two functions:

- Function dimArrayReal(OUT REAL) returns the number of dimensions of an array. For example, with the array REAL m[23,2,5] if we call the function:

```
INTEGER dim= dimArrayReal(m[1])
```

the value returned will be 3 (dim=3) because this array has 3 dimensions. It must be established that the argument passed is “m[1]”, in other words the first element of the array must always be passed as though it were in one dimension because the number of dimensions is not yet known

- Function sizeArrayReal(OUT REAL, INTEGER code), depending on the code, returns an integer with the size of:
  - If code is 0, it returns the total size of the array
  - If code is greater than 0 and less than the number of dimensions (dim), it returns the size of the dimension given in code

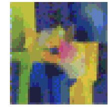
  For example, it could be written like this:

```
total= sizeArrayReal(m[1], 0) – total will be 23 * 2 * 5 = 230
seconDim = sizeArrayReal(m[1], 2)  -- seconDim will be 2
```

The other functions are the same as those described above, but using integer, boolean and string arrays.

The following is a complete example of the use of these functions:

```
FUNCTION REAL funfun(OUT REAL m[])
DECLS
   REAL x
   INTEGER size
BODY
   size= sizeArrayReal(m[1],1) -- obtain dimension  1 size
   FOR(i IN 1, size)
      m[i]= 8.34
   END FOR
   x= SUM(i IN 1,size; m[i])
   RETURN x
END FUNCTION
```



```
COMPONENT test
DECLS
   REAL g[13]
   REAL x
CONTINUOUS
   x= funfun(g)
END COMPONENT
```

This example shows how to create an EL function having an array argument and making some operations on the array calculating the size of the array.

There are another couple of special functions to obtain the size and dimensions of an array given the name of the array in string format (e.g., "v"). But they have a constraint; they can be used to find sizes of arrays of the mathematical model (the partition) and experiments. They are not valid to obtain sizes of arrays declared in functions or classes. The functions are:

```
FUNCTION INTEGER dimArrayName(IN STRING name)
FUNCTION INTEGER sizeArrayName(IN STRING name,INTEGER code)
```

The first function returns the number of dimensions of an array with name "aName" and the second the size of the total array or a dimension (like the previous functions). An example of usage:

```
COMPONENT foos
DECLS
     REAL vxm[3,4,5]
END COMPONENT


EXPERIMENT exp1 ON foos.default
DECLS
   INTEGER dim
BODY
                 -- get the number of dimensions
   dim= dimArrayName("vxm")
   WRITE("DIMENSIONS: %d\n", dim)
                -- get the total size of array
   dim= sizeArrayName("vxm",0)
   WRITE("TOTAL SIZE: %d\n", dim)
                -- get the size of the third dimension
   dim= sizeArrayName("vxm",3)
   WRITE("SIZE OF SECOND DIMENSION: %d\n", dim)
END EXPERIMENT
```

This experiment produces the following output:

```
DIMENSIONS: 3
TOTAL SIZE: 60
SIZE OF SECOND DIMENSION: 5
- END OF EXPERIMENT -
```

8.4.7. NEW_BRANCH() Function

During simulation it is sometimes necessary to show different curves for a single variable in the same graph.

The function NEW_BRANCH() enables us to indicate at what moment of the simulation each branch commences and to assign it a specific identity.

The syntax of this function is:

```
NEW_BRANCH(STRING label)
```

where “label” represents the text which is added to the name of each variable in the legend of each graph. This label can also include numeric values from the simulation, using for example the syntax:

```
NEW_BRANCH(“VarName = $VarName”)
```

where “$VarName” is the value taken by the variable “VarName” the moment the new branch is created.

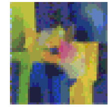
Some typical examples of the use of branches in a simulation are given below.

8.4.7.1. A simple Example

A simplest example is to represent two different branches of a simulation marking each one with a label.

We will use the following component:

```
COMPONENT equation
   DATA
      REAL tau = 0.6                "Delay time (s)"
   DECLS
      REAL x                        "Main variable (-)"
      REAL y                        "Delayed variable (-)"
   CONTINUOUS
      y' = (x - y) / tau
```



```
END COMPONENT
```

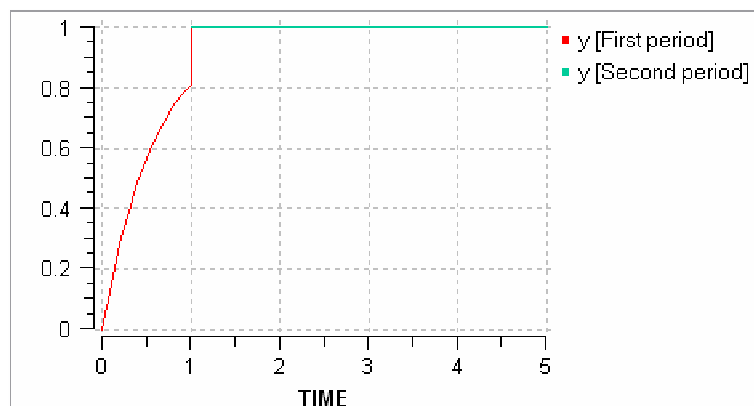
which represents the behaviour of a model whose representative variable “y” responds to an input variable “x” with a delay characterised by the parameter “tau”.

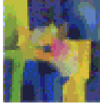
We could then generate a default partition and an experiment of the following type:

```
EXPERIMENT exp1 ON equation.default
   INIT
      y = 0.
   BOUNDS
      x = 1.
   BODY
      TIME = 0.
      NEW_BRANCH("First period")
      TSTOP = 1.
      CINT = 0.1
      INTEG()
      STEADY()
      NEW_BRANCH("Second period")
      TSTOP = 5.
      CINT = 1.
      INTEG()
END EXPERIMENT
```

Two periods can be differentiated in this experiment. In the first, the model is integrated between 0s and 1s and a steady state is calculated; in the second, integration continues up to 5s.

The label “First period” is assigned to the first and “Second period” is assigned to the second, as illustrated in the graph resulting from the simulation:





8.4.7.2. Parametric Study (One Parameter) Example

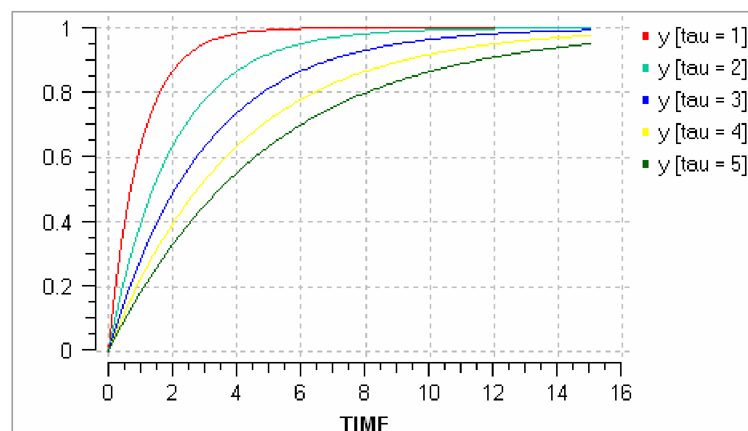
In this example we want study the change in the transient response of a determined system when one of its characteristic parameters varies.

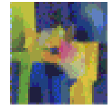
To do this, we can create another experiment using the same model (component and partition) from the previous experiment, expressed as follows:

```
EXPERIMENT exp2 ON equation.default
   INIT
      y = 0.
   BOUNDS
      x = 1.
   BODY
      FOR (i IN 1,5)
         TIME = 0.
         tau = i
         NEW_BRANCH("tau = $tau")
         TSTOP = 15.
         CINT = 0.1
         INTEG()
         RESET()
      END FOR
END EXPERIMENT
```

where a loop is constructed using the statement FOR with a variable “i” which takes values between 1 and 5, and which is used to make the parameter “tau” vary. Each of the different values is integrated into the model between 0s and 15s.

In this example, with the statement NEW_BRANCH we can create a different curve (different input in the graph legend, different colour, etc) for each value of “tau” after simulating this experiment in the monitor, as illustrated below:





We can also see in the legend that each branch contains its own input identified with the label defined and where “$tau” has been replaced with its value.

8.4.7.3. Parametric Study (Two Parameter) Example

We can do just the same for a model which depends on two characteristic parameters, nesting loops appropriately.

Let’s say that a certain system depends on two parameters “k1” and “k2”, and we want to analyse how a determined system reference variable “y” varies for a determined range of values for these parameters.

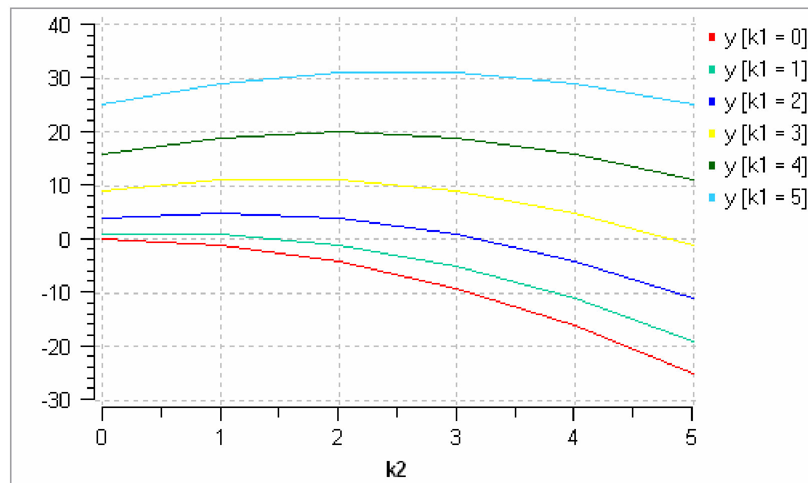
An elementary component would be:

```
COMPONENT quadratic
   DATA
      REAL k1 = 0.
      REAL k2 = 0.
   DECLS
      REAL y
   CONTINUOUS
      y = k1**2 + k1 * k2 - k2**2
END COMPONENT
```

For which we can create a default partition and the experiment could be expressed as follows:

```
EXPERIMENT exp ON quadratic.default
   BODY
      TIME = 0.
      FOR (j IN 0,5)
         k1 = j
         NEW_BRANCH("k1 = $k1")
         FOR (i IN 0,5)
            k2 = i
            STEADY()
         END FOR
      END FOR
END EXPERIMENT
```

The result of this simulation would be like this:



where we can see how "y" varies with the parameter "k2" for each of the values of parameter "k1" shown in the legend.

8.4.8. Function to obtain Units of Variables

The unitsOfVariable() function can be used to obtain the units of a variable as follows:

```
EXPERIMENT exp2 ON turbojet.default
DECLS
      STRING unitsv
   BODY
      unitsv= unitsOfVariable("Bleed.g_branch[2].WF")
      WRITE("unitsv= %s\n",unitsv)
END EXPERIMENT
```

The output of this experiment could be:

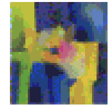
```
unitsv= kg/s
```

If the variable did not have any units or if it did not exist, an empty string is returned. If the user wants to check first whether the variable exists, the existsVariable() function can be used.

8.4.9. Function to obtain the Release Report

The versionReport() function returns a string with the identification of the version of the program including the following:

- Names of experiment, library, component and partition



- User who generated the experiment
- Date and time of creation
- Libraries used and their versions
- Name and version of program

An example of the use of this function is as follows:

```
EXPERIMENT exp2 ON turbojet.default
DECLS
      STRING vreport
   BODY
      vreport= versionReport()
      WRITE("vreport= %s\n",vreport)
END EXPERIMENT
```

It will be printed out:

```
Experiment: exp2 Library: GAS_TURBINE_EXAMPLES Component: turbojet Partition:
default
User: pce Date: 06/28/07 Time: 16:25:17
Library dependency list: CONTROL V0.0 GAS_TURBINE V0.0 GAS_TURBINE_EXAMPLES
V0.0 MATH V0.0
Programme & Version: EcosimPro V4.1.0
```

8.4.10. Function to obtain Internal Residues

If the user wants to know the value of an internal residue of the Jacobian, the getResidueValue() function can be used with the following format:

```
residue1 = getResidueValue(1)
```

This function returns a REAL with the value of the internal residue 1. The internal residues are associated with the dynamic and algebraic variables of the Jacobian equations system.

To know the Jacobian variables, display the partition information file. In this file you will find a table with the Jacobian variables and their associated closure equations. The function allows the user to obtain any of the N residues at any time.

If the residue number you pass to the function is lower than zero or greater than the number of internal residues (or Jacobian variables), the function returns 0.0 and displays an error message.

8.4.11. Deactivate the solving of internal algebraic boxes.

By default EcosimPro will try to solve the internal algebraic boxes every time the user wants to make steady or transient studies. There is a function to deactivate/activate this calculation (by default it is activated):

```
"C++" FUNCTION NO_TYPE solveAlgebraicBoxes(BOOLEAN flag)
```

When this function is called in an experiment, it will activate (default) or deactivate the calculations. Example:

```
EXPERIMENT exp1 ON test.default
BODY
     solveAlgebraicBoxes( FALSE )
     STEADY()
END EXPERIMENT
```

8.4.12. Function to issue errors from EL

The issueError() function is used to generate error messages from the EL code. Its syntax is as follows:

```
NO_TYPE issueError(IN INTEGER errorId, …)
```

where "errorId" is the error number to be displayed. The three dots mean that it can take a variable number of arguments. Normally these arguments are used to pass the values of variables to be printed in the error message (in a similar way to the WRITE() function)

The user can create a file with error codes for each library in the following way:

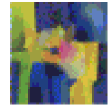
- Create a text file called LIBNAME.errors.txt and place it just below the directory of the library. For example, if the library is ELECTRICAL, the file will be called ELECTRICAL.errors.txt and will be in the ELECTRICAL directory.

- The file format must include a header with the filed names separated by tabs. Below, in each row is each error code. For example:

```
CODE SEV  CAT  SOURCE    MESSAGE
255  9    24   98        The beta(BETA=%g) is too high
256  99   25   98        The compressor is out out limits (%g,%g)
```

where:

- CODE is the number of the error code. Codes 212, 213,214, 215 and 216 are reserved for the different types of assertions, and so should not be used.



- SEV is the severity, as follows:

Note: 0-6 Warning: 7 Error: 8 Killpoint: 9 Fatal: 99

- CAT is the category, which can be any positive number

- SOURCE is the source of the error, represented by a positive integer. Usually a different number is used for each library, so that the source of the error can be identified rapidly.

- MESSAGE is the message to be displayed in the event of error and is free format. If an argument is required one of the following may be used:

```
%d for integers
%g for reals
%s for strings
```

In this way when the function is called arguments can be passed to it. Similar to the EL WRITE() statement.

For example, to create a new error, with code 312, type KILLPOINT, category 22 and source 98. The message has to inform that a value is outside the limits, and has to print both the value and the limit, which are passed to it as arguments.

To do this, make a new entry in the error file, as follows:

```
CODE SEV  CAT  SOURCE   MESSAGE
312  9    22   98   The variable %s with value %g is out of
limits(%g,%g)
```

If the function is then called:

```
issueError(312,"speed",4.54, 5.5, 10.5)
```

and it is executed, the following message is printed:

```
*** KILLPOINT level 2 (code 9:312:22:98) path: .) in file
"exp1.exp" at line 14 ***
The variable speed with value 4.54 is out of limits(5.5,10.5)
```

The current calculation will halt and the program hill continue with the next one.

9. Partitions

The partition in EcosimPro is an intermediate step between the component and the experiment. A partition is a mathematical model of the component.

When we want to simulate a component, we must first define a partition and then create experiments for that partition. EcosimPro always provides a default partition for each component.

The partition is identified by a name, the default partition is called "default". Users will be required to give new names to new partitions introduced by them. Normally, in the end, the modeller has components with one or more partitions and each partition has one or more associated experiments.

The development environment of EcosimPro will always give the user the option of defining a default partition. If this is the case, it is the system that selects the default options which will be correct in some cases and not in others. It is recommended that this option be used when first learning to model in EcosimPro as it speeds up the job.

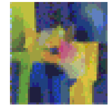
As users becomes more experienced, it is recommended that the partition be generated step by step because they can still use the default options proposed by EcosimPro.

There will be physical problems that can only converge if a correct variable is chosen to solve a high index problem or a break in an algebraic loop. In such cases there is little EcosimPro can do with its mathematical knowledge of the problem. What it needs is physical knowledge of the problem which only the modeller can provide.

EcosimPro will also offer the option of editing a partition and observing each of the conditions that have been imposed for a partition. Two partitions will be identical if all the conditions they have are identical; these conditions are described below. It must be borne in mind that the mathematical model will very often change, depending on which partition characteristics have been selected. In this respect, it is recommended that users pay special attention when it comes to generating the partitions.

EcosimPro can make different partitions depending on the following factors:

- Remove derivatives from the model
- Change DATA to unknown variable
- Select boundary conditions
- Optimal tearing of algebraic loops



- Resolve high index problems

Let's take a look at the different cases, one by one.

9.1. Remove derivatives

The modeller can decide to make all derivatives of the model equal to zero. This can be done typically when he wants to remove all dynamic aspects of the mathematical model.

This option will replace any derivative in the equations by a zero. For instance in an equation such as:

```
x' + 4*y = sin(TIME) + data1
```

will be replaced by:

```
4*y = sin(TIME) + data1
```

9.2. Change DATA to unknown variables

Users will sometimes want to change some DATA in the modelling for unknown variables. This is typically used to carry out design studies and to give EcosimPro very powerful flexibility. The following is an example with the modelling of a resistor:

```
COMPONENT Resistor IS_A Two_pins
     DATA
          REAL R = 1000    "Resistance (Ohms)"
     CONTINUOUS
          v = e_p.i * R
END COMPONENT
```

Normally the variable "R" will be considered as known and constant. EcosimPro will not assume that it should look for an equation to calculate it. However, we may wish to carry out a parametric study of this variable, imposing other typical unknown variables: the voltage ("v") or the current ("e_p.i"). If we choose to make a design partition, the first question EcosimPro will ask is if we want to change any DATA to unknown variables. We can select the datum R and EcosimPro will subsequently follow all the steps of the normal partitions. At some point we will be asked about boundary conditions and we can enter the voltage or the current. In this case, EcosimPro can handle Ohm's law symbolically so that it is left with the equation:

```
R = v / e_p.i
```

As we can see, there is tremendous flexibility with this capability. It means that even having modelled a problem assuming the DATA and the variables, we can still change it at the last moment and in the end we can define which are variables and which are not.

9.3. Boundary Conditions

The boundary conditions impose conditions on the system of equations to make the number of equations coincide with the unknown variables. Sometimes the system contains more variables than equations. In these cases the modeller must decide which variables will be considered boundary conditions.

For example, let's assume we have the following simple component:

```
COMPONENT foo
      DECLS
            REAL x, y
      CONTINUOUS
            x + 3 = sin(y) * 8
END COMPONENT
```

This component has two unknown variables (x and y) and a unique equation. One variable should be considered as a boundary condition and it is the modeller who decides which one. In EcosimPro terms there are two potential partitions:

- Partition 1: "y" is a boundary. EcosimPro will transform the equation to

```
x= (sin(y) * 8) - 3
```

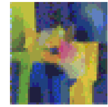
- Partition 2: "x" is a boundary. EcosimPro will transform the equation to

```
y= asin( (x+3) /8)
```

EcosimPro always suggests a solution, but it is the modeller who decides the final set of boundaries. EcosimPro will then check that the selection is correct, otherwise it will issue an error and the user would need to change the selection.

9.4. Tearing Algebraic Loops

EcosimPro helps to solve the non-linear algebraic loops of equations. In this case the modeller must decide which variable(s) should be chosen as algebraics.



EcosimPro will suggest a set of algebraic variables, but it is the the modeller who decides the final set of variables. In anycase, EcosimPro will check that the set is correct mathematically speaking but it can not be checked that is correct physically speaking. It is the modeller's task to ensure that the set of algebraic variables he selects is correct.

9.5. High Index Problems

Users will also have the option of solving high index problems (see Mathematical Algorithms document) by selecting the dynamic variables which they do not want the solver to calculate.

In this case some state variable depends on another or it can be transformed symbolically and EcosimPro will help the modeller by selecting a reasonable set of state variables to become regular unknown variables. For instance given the component:

```
COMPONENT NewtonLaw
DECLS
     REAL F  "Force"
     REAL m  "mass"
     REAL x  "lenght"
CONTINUOUS
     F = m * x''
     x= sin(TIME)
END COMPONENT
```

EcosimPro will detect that the term x´´ can be calculated from the last equation and will produce the following mathematical model:

```
x= sin(TIME)
x´= cos(TIME)
x´´ = -sin(TIME)
F= m * x´´
```

In other words, x´and x´´ are no longer treated as state variable since we have the symbolic expressions to calculate them and they are treated as explicit variables (like F).

10. Experiments

Once a model has been developed, we can run experiments on it. EL has an experiment language which is very similar to the language used for functions and components as described above. Its purpose is to enable us to initialize data, to calculate steady states or to integrate the model in a timeframe through the use of sequential instructions (ie, assignment, IF, FOR, etc).

The BNF notation is as follows:

```
experiment_def   ::= experiment_item_s
              EXPERIMENT IDENTIFIER ON scoped_id
             ( DECLS var_decl_s )?
          ( OBJECTS class_instace_stm_s  )?
             ( INIT expt_assign_stm_s )?
             ( BOUNDS expt_assign_stm_s )?
             ( BODY seq_stm_s )?
              END EXPERIMENT
experiment_item_s::=  ( use_stm | extern_func_decl |
                        function_def ) *
```

“experiment_itms_s” can be one of three types of declaration:

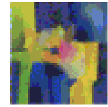
- USE statement
- Predeclaration of external functions in C/C++ and FORTRAN
- Definition of any function in EL used in the experiment. Its format is identical to any other EL function, with some added capabilities like call integration functions and direct use of TIME, TSTOP and CINT variables

We declare a new experiment with a name (IDENTIFIER) and the model and partition on which the experiment is to be carried out. For example, to declare the experiment testBridge on the component/partition diodeBridge.partition1, we would declare the following:

```
EXPERIMENT testBridge ON diodeBridge.partition1
```

Within the experiment, the following 4 optional blocks can be included:

- DECLS. Declarations of local variables to the experiment
- OBJECTS. Declaration of external objects defined with classes
- INIT. Initial values (ie, starting values) assigned to the dynamic variables of the system



- BOUNDS. Math expressions assigned to the boundary variables
- BODY. This block contains the sequential instructions which define the experiment. All sequential EL instructions such as IF-THEN-ELSE, FOR, WHILE, etc, are valid in this block. You can also use special functions, which are valid only in this block, in order to integrate a model, calculate steady states, etc. These functions are described in the following paragraphs

10.1. DECLS Block

The DECLS block contains all the local variables used in the current experiment. They can reuse any of the fundamental types of EL like REAL, INTEGER, TABLE_1D, etc.

Example:

```
DECLS
     REAL x
     REAL v[3,4]
     STRING st = "test"
```

10.2. OBJECTS Block

An experiment can use instances of C++ classes in the OBJECTS block. In this block the objects are simply instantiated and then used with the format: object.(method/attribute). For example, given the following external class:

```
EXTERN CLASS mmath
  DECLS
           REAL value
  METHODS
           EXTERN METHOD REAL add(IN REAL a, IN REAL b)
           EXTERN METHOD REAL subs(IN REAL a, IN REAL b)
END CLASS INCLUDE "<mmath.h>" IN "testCpp.lib"
```

it can be used from an experiment and its variables and methods made use of; for example:

```
EXPERIMENT exp1 ON myTest.default
OBJECTS
     mmath m1
BODY
   m1.value= 4.3
```

```
   x = m1.add(2,2)
   STEADY()
END COMPONENT
```

This means it allows the use of objects in any part of the function. To access its variables, simply write the following:

```
   ObjectName.variableName
```

and to call its methods:

```
   ObjectName.methodName( arguments )
```

To read more about this subject go to the section on EL connection with external classes.

10.3. INIT Block

The dynamic variables, those whose derivative appears in the formulation, and the algebraic unknowns have to be initialized prior to integration.

The dynamic variables are fixed in the model, but the algebraic unknowns are not fixed in the model. The processed model might contain implicit algebraic equations, and the number of algebraic unknowns must be equal to the number of algebraic implicit equations. The unknowns of the algebraic equations could be any of the non-explicit variables.

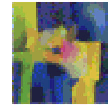
For example, let's consider the following algebraic system of two equations:

$$f\ (x, y, z) = 0$$

$$g\ (x, y, z) = 0$$

In this equation system, we can provided **z** as a data or boundary condition and we calculate the **x** and **y** values. But, it is also possible to fix the value of **x** as a datum, and to calculate **y**, and **z**. Of course, we need to enter starting values for the implicit unknowns, because the solution procedure is iterative.

The way to specify which non-explicit variables are the algebraic unknowns of the equation system is by assigning them an initial value in the INIT block. The initial guessed value of the algebraic unknowns should not be very far from the actual solution, because the convergence to the correct solution is not guaranteed.



In summary, all the dynamic variables have to be initialized in the INIT block with the exceptions mentioned later. The algebraic unknowns of the equation system also have to be specified and initialized in the INIT block.

In general, all the dynamic variables, those whose derivative appears in the component formulation, require an initial value to start the solution process. The default-generated experiment automatically lists all the algebraic and dynamic variables. The user just needs to assign initial values to them.

10.3.1. Self-Initializing Components

There are some components that provide self-initialization of the component dynamic variables as a function of some component data. This initialization is written into the INIT block of the component, for instance see the following component called "Channel":

```
ABSTRACT COMPONENT Channel
      PORTS
            IN    fluid f_in        "input port"
            OUT   fluid f_out       "output port"
      DATA
            ENUM FluidNames Name_Fluid = Xenon   "Fluid name"
            REAL D = 0.010                 "Fluid diameter (m)"
            REAL l = 1.0                   "Pipe length (m)"
      DECLS
            EXPL REAL A             "Fluid area in component (m^2)"
            EXPL REAL V             "Fluid volume in component (m^3)"
      INIT
            f_in.fluidname = Name_Fluid
            f_out.fluidname = Name_Fluid
            A = 0.25 * PI * D**2    -- init the area
            V = A * l                     -- init the volume
      CONTINUOUS
            f_in.bA = A
            f_out.fA = A
END COMPONENT
```

In this case we have made the initialization locally because we use other variables to calculate the initial values of some variables.

In general, the dynamic and algebraic variables are taken from this block in the component and if we try to initialize them in the experiment, it will have no effect. There are no compilation errors but the initial values are not taken from the initialization in the experiment. In fact, it is calling first the experiment initialization and then the component initialization; ie, it overrides the first values. We must be aware that for the boundaries it is the opposite, and the valid initialization is taken from the BOUNDS block in the experiment (see the next section).

10.4. BOUNDS Block

The BOUNDS block sets the values for every boundary of the model. The values can be complex expressions depending on time or a single value. It is called every time the model tries to evaluate the boundary variables (it overrides any other boundary initialization).

During the experiment we can also modify any algebraic, dynamic or boundary variable. EcosimPro will detect automatically when to reinitialize the model. In other words, it is up to the user when and how to modify any variable but for tracing purposes it is recommended that they be written into the corresponding block.

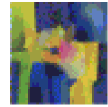
Using the interface, the system will automatically fill in the blocks INIT and BOUNDS. The user only has to write in the correct initial values or expressions.

The following is an example of an experiment:

```
EXPERIMENT exp1 ON myModel.partition1
     DECLS
          REAL m
     INIT
          z1.x = 4
          z2.x = 5
     BOUNDS -- set expressions for boundary variables
          b1.t = sin(TIME / 2)
          b2.t = cos(TIME / 2)
     BODY
          REPORT("myReport","*")
          STEADY()
          INTEG_TO(10, 0.1)
          z2.x= 6
          b2.t= sin(TIME/2)
          INTEG_TO(200,15)
          RESET()
          b2.t= cos(TIME / 2)
          INTEG_TO(20,0.1)
END EXPERIMENT
```

This experiment, called exp1, will simulate the myModel.partition1 model. It will take the following actions:

- Initialize the dynamic variables z1.x and z2.x
- Set the corresponding expressions to calculate the boundaries
- Create a report called "myReport" to print all variables



- Calculate a steady state (STEADY function)
- Integrate from 0 to 10 seconds with a communication interval of 0.1 seconds
- Change the values of dynamic variable z2.x and boundary b2.t
- Integrate from 10 to 200 seconds with a communication interval of 15 seconds
- Reset model to original initial values. Also TIME returns to zero
- Change boundary b2.t to initial expression again
- Integrate from 0 to 20 seconds with a communication interval of 0.1 seconds

10.5. BODY Block

10.5.1. Experiment Functions and Variables

EL provides a set of predefined functions for experiments. They are fundamental functions used to integrate models, calculate steady states, generate reports, etc. You can create your own function library and call functions from it for use in the experiment. Appendix B contains a description of all the available experiment functions.

10.5.1.1. Variables and Data Types in Experiments

During the experiment, some EL global variables are in scope at all times (some of them like TIME, TSTOP and CINT have already been explained). Below are the most important.

For the transient setup:

- **TIME**: The current integration time. It is used to set up and/or change an initial integration time. At the beginning of an experiment body, this variable is always zero, so if you wish to execute an integration from zero, it is not necessary to initialize it the first time
- **TSTOP**: The end time up to which the integration is carried out. This variable can be modified dynamically
- **CINT**: The communication interval of the integration; ie, the frequency with which data and plotter charts are made available to you. It also indicates the detection of intermediate events (discontinuities) which cause the model to reinitialize
- **ABS_ERROR**: The maximum absolute error of the transient integrator. This error is taken into account by the differential-algebraic solver. By default it is 1E-6

- **REL_ERROR**. The maximum relative error of the transient integrator solving differential-algebraic systems. By default it is 1E-6
- **INIT_INTEG_STEP**. The size of the first integration step. This is only used in special cases when the integrator is to use a smaller step and then the default step. Normally the value is –1; this means that the integrator will use its default value
- **IMETHOD**. This experiment variable allows us to switch from one numerical integration method to other. EcosimPro provides three integration methods: DASSL, DASSL_SPARSE and RK4 (Runge-Kutta 44). By default DASSL is used if the sparsity factor is lower that 93%, otherwise DASSL_SPARSE is used

For the steady solver:

- **TOLTYPE**. The type of tolerance used in the steady solver (set in TOLERANCE). It can be either relative (RELTOL) or absolute (ABSTOL). By default it is set to RELTOL
- **TOLERANCE**. The relative or absolute maximum error required for the steady solver. The default value is 1E-6
- **SMETHOD**. It defines the steady algorithm to be used. At the moment two solvers are available: the EcosimPro solver based on the improved Newton-Raphson (NR) or the Minpack algorithm (MINPACK). By default the first one is used and it is recommended since the debugging information provided by this method is very complete (e.g., Jacobian evolution) whereas the second does not have this kind of debugging information

For both types of solvers:

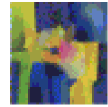
- **DEBUG_LEVEL**. Used to set the level of debugging (default is 1). It is recommend to change this variable using the setDebugLevel().

Some integration functions use a data type (**statusIntegration**) to indicate at which instant of the integration the variables are. It is an enumeration type which has the following values:

- **INTEG_BEGIN** indicates that it is at the initial state of integration
- **IS_CINT** indicates that integration has been completed up to a communication interval or up to an internal integration step
- **IS_STEP** indicates that integration has been completed up to the next internal integration step
- **INTEG_END** indicates that integration has been completed

RESET Function

```
RESET()
```



Used to reinitialize the mathematical models. The following are changed:

- Resets all model variables to default except the TIME, TSTOP and CINT
- Executes the INIT blocks of the components (if any)
- Executes the INIT block of the experiment (if any)
- Checks all events (time events, WHENs) to reach a stable situation (if any)

RESET_EVENTS()

Used to reset all events to reach a point where no more events are produced. Typically it is used by the tool to run all events at the beginning.

RESET_VARIABLES()

Resets all variables to the initial situation. It does not changed the variables TIME, TSTOP and CINT.

10.5.1.2. Integration Functions

The modeller can calculate transient states easily in EcosimPro by using the functions explained in this section.

The basic mathematical problem solved by the EcosimPro transient solver can be stated if intermediate explicit equations are disregarded, such as:

$$F\left(t, y, \dot{y}\right) = 0$$

The EcosimPro transient solver is based on the GEAR mathematical method. The underlying idea of Gear for solving differential-algebraic systems (DAS) is to replace the derivative in (1) with a difference approximation, and then to solve the resulting equation for the solution at current time tn using Newton's method (see the EcosimPro math document for more details).

Very frequently, transients are initiated from known equilibrium points, with null derivatives. Rare would be the cases in which the initial values estimated by the user for the dynamic variables provide a steady state as a model condition.

Before explaining EcosimPro's different integration functions, we will explain the concepts of initial time of integration, integration step, communication interval step and final integration time:

- The initial time of integration is defined by the variable TIME. It is the starting time for the integration process. This variable can be changed at any time and new integration should reinitialize the system again. It is not recommended that the value of TIME be changed in the middle of an integration process
- Integration step is an internal step made by the integration routine. Normally the user is not interested in the output of this step because it is very small
- Communications interval defined by the variable CINT. The user wants to simulate a model and obtain data values at certain times, normally using a fixed period of time, for instance every 0.1 seconds. The integration will stop at this time to report values. There will also be a communication interval when an event occurs. Again the integration stops and all the variables will be refreshed
- Final integration time is defined by the variable TSTOP. It is the stop time to be integrated. When this time is reached the integration stops

There are several routines to integrate, some of them carry out the process in one go, ie, from TIME to TSTOP; others integrate a communication interval and return control to the user; the last one integrates an internal step and returns control to the user. Experience shows that those used most are the first ones, but EcosimPro allows the user to create sophisticated integration processes with full control of the integration.

INTEG()

This function will integrate the model assuming that you have already assigned the initial time (TIME), end time (TSTOP) and the communications interval (CINT) global variables. For example:

```
BODY
     TIME = 0
     TSTOP = 10
     CINT = 0.1
     INTEG()
```

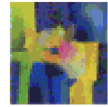
This block of code integrates a model from 0 to 10 seconds with a communication interval of 0.1 second.

INTEG_TO (REAL tstop, REAL cint)

This is a simplified version of the previous function. It is used to integrate a model by specifying the end time and the communications interval. For example:

```
BODY
     INTEG_TO(10,0.1)
```

This function has the same effect as the previous one.



INTEG_CINT()

This function integrates up to the next communications interval and returns a code. It is useful for checking on an interval-by-interval basis. The function returns a **statusIntegration** type code which indicates the status. It normally returns the **IS_CINT** code and at the end it returns **INTEG_END**. The function could be used as follows:

```
BODY
     TIME= 0
     TSTOP= 10
     CINT= 0.1
     WHILE ( INTEG_CINT() != INTEG_END )
          IF ( c1.speed > 9 )
               reached= TRUE
          END IF
     END WHILE
```

This code integrates each integration step and hands control back to the user for any purpose, in this case to check if a variable meets a condition.

INTEG_STEP ()

This function integrates an integration step and returns a code. It normally returns the **IS_STEP** code and at the end it returns **INTEG_END**.

The granularity of this function is smaller than the previous one; at each communications interval there can be multiple internal integration steps.

It can be used for a more detailed study of intervals in which you can check the evolution of the model on a step-by-step basis.

The following integrates the previous example step by step:

```
WHILE ( INTEG_STEP() != INTEG_END )
     IF ( c1.speed > 9 )
          reached= TRUE
     END IF
END WHILE
```

In this case you can integrate a step and check the variable speed.

CONTINUE()

EcosimPro has another unique call to continue the previous integration without having to reinitialize any parameter or satisfy residuals. It is called CONTINUE() and can be used to continue the last integration at a TIME with a bigger value. For example:

```
TIME = 0
TSTOP = 10
CINT = 1
INTEG()  -- integrates up to 10 seconds and stops
      -- make some calculations with the results
TSTOP= 20
CONTINUE()  -- continues last integration up to 20 seconds
```

This function is optimised to continue a previous integration with a minimum overhead.

10.5.2. Steady States

Visit chapter entitled "Steady State Calculations" to know how to solve steady states of mathematical models.

10.5.3. Integration Methods

EcosimPro has several numerical solvers for the differential-algebraic systems. By default, EcosimPro uses the dense DASSL method if the sparsity of the final Jacobian matrix is lower than 93%; otherwise, it uses a sparse version of DASSL which is named DASSL_SPARSE in EcosimPro. This leads to a lot of time being saved in the integration process.

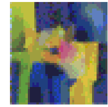
The user can view the default selected method in the Mathematical View of the Partition.

```
Size of jacobian matrix: 32x32
Sparsity factor in jacobian matrix: 25.67%
Default integration method: DASSL
```

EcosimPro also provides a classical integration method based on the fourth order Runge-Kutta method (in EcosimPro it is called RK4). This is an explicit method with fixed order and step. It makes four derivative evaluations per step. It works very well if the problem is not too complex and also when there are a lot of discrete events in the system.

The user can change the method during the same experiment. There is a global variable called IMETHOD which can be used to change it during the simulation, for example

```
BODY
        -- Integrates using dense DASSL method
     IMETHOD= DASSL
     INTEG_TO(10,0.1)
        -- Integrates using sparase DASSL method
```



```
            IMETHOD= DASSL_SPARSE
            INTEG_TO(100,5)
               -- Integrates using Runge-Kutta 44
            IMETHOD= RK4
            INTEG_TO(101,0.001)
END EXPERIMENT
```

All the complexity in changing the method internally is hidden to the user.

10.5.4. Integration at Time Zero

With EcosimPro, integration can be performed at 0 time. It simply calls the component INIT blocks and satisfies residuals with the equations given in the CONTINUOUS blocks. For example, given the component:

```
COMPONENT test
DECLS
     REAL x
     REAL y
INIT
     x= 3
     y= 4
CONTINUOUS
     x' = sin(TIME)
     y= x + 2
END COMPONENT
```

If the following is done in the experiment:

```
TIME = 0
TSTOP = 0
CINT = 0
INTEG()
PRINT("x= $x y= $y")
```

The value that is printed will be:

```
x= 3 y= 5
```

We can check that this will be the initial value of the variables just before integration begins.

10.5.5. Detecting Type of Ongoing Calculation

There are two boolean variables which inform whether the current calculation is transient or steady. They are declared as:

```
BOOLEAN IN_TRANSIENT
BOOLEAN IN_STEADY
```

The variable IN_TRANSIENT is TRUE when EcosimPro is calculating a transient, otherwise it is FALSE.

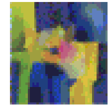
Variable IN_STEADY is TRUE when EcosimPro is calculating a steady state, otherwise it is FALSE.

These two variables can be used in any component, port, experiment, etc. For example, to calculate an initial value of a variable based on whether the experiment is calculating a transient, a steady or any other situation:

```
COMPONENT foo
DECLS
     REAL x
INIT
     IF( IN_TRANSIENT = TRUE) THEN
               x= 3.0
     ELSE IF (IN_STEADY == TRUE) THEN
              x= 4.0
     ELSE
        x=5.0
...
END COMPONENT
```

10.5.6. Report Generation

In EL, all external reports and plots that trace the simulation variables are called views. EL allows users to generate reports and plots by means of functions. The views are updated automatically and users do not need to be concerned about them. They are normally updated at every communications interval. Users can update reports manually at any time using the function REPORT_REFRESH(), and they can also activate or deactivate report generation. The following paragraphs explain the functions for handling reports.



10.5.6.1. REPORT_TABLE(fileName, variables)

This function creates a file called fileName which consists of a report on the variables specified. Both arguments are character strings. The variables can consist of the complete names of variables or of simple regular expressions, separated by the white space symbol. The format of regular expressions is explained in Appendix A.

Examples of report generation:

```
   -- report "rpt" with all the variables
REPORT_TABLE("rpt","*")
   -- report with all the variables ended in 'v'
REPORT_TABLE("rpt","*v")
   -- report with all the variables ended in 'v' and
      -- starting by 'valve'
REPORT_TABLE("rpt","valve*v")
-- report with all the variables ended in 'v' and
   -- any other starting by 'valve'
```

10.5.6.2. REPORT_TABLE("rpt","*v valve*")

This type of report generates a table where the first row contains the names of the variables, the first column always contains the time, and the subsequent columns contain the values of the variables at each instant of time. Each row terminates with a newline character and fields are separated by a tab. The table generated is available for your immediate use in EXCEL. The example below shows a table for 3 variables: 2 reals, (x and y), and a Boolean (z), It was created by the following function call: REPORT(“rpt”,”x & y & z”)

TIME	x	y	Z
0.0	0.23445	78.90	TRUE
0.1	0.33340	90.90	TRUE
0.2	0.45550	98.08	FALSE
0.3	0.54210	100.23	FALSE

10.5.6.3. REPORT_LIST(name, variables)

This function is a variant of the previous one, the only difference being in the format of the final report which is generated. Instead of a table it outputs a list with the following format:

```
variableName = value
```

This format is normally used to generate reports after calculating a steady state. The format to obtain a report similar to the last one is:

```
REPORT_LIST("rpt","x  y  z")
```

The file created would be as follows:

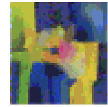
```
     TIME= 0
     x = 0.23445
     y= 78.9
     z= TRUE
--------------------------
     TIME = 0.1
     x= 0.3334
     y = 90.9
     z = TRUE
--------------------------
     TIME = 0.2
     x = 0.4555
     y = 98.08
     z = FALSE
--------------------------
     TIME = 0.3
     x = 0.5421
     y = 100.23
     z = FALSE
--------------------------
```

10.5.6.4. SET_REPORT_ACTIVE (string name, BOOLEAN newState)

This function is used to activate or deactivate reports. You simply type the name of the report and the status it is to have. If a report is deactivated it means that it will not be automatically updated. When reports are created, they are activated by default. For example:

```
EXPERIMENT exp ON test.default
     BODY
          REPORT("rpt"," x & y ")
          INTEG_TO(5,0.1)
          SET_REPORT_ACTIVE("rpt",FALSE)
          INTEG_TO(10,0.1)
END EXPERIMENT
```

In this example, report “rpt” is activated for the first integration but not for the second.



10.5.6.5. *REPORT_REFRESH ()*

The function REPORT_REFRESH() is used to refresh all reports manually. All views are updated each time it is called. It is useful when data is to be reported at specific times. For example, in the following experiment, the views are updated at the normal frequency (0.1 seconds) up to TIME 10 seconds, and from 10 seconds onwards there is an extra manual update at every integration step (added to the automatic refresh every communication interval).

For example:

```
EXPERIMENT exp ON test.default
      BODY
            REPORT("rpt"," x & y ")
            TIME= 0
            TSTOP= 20
            CINT= 0.1
            WHILE ( INTEG_STEP() != INTEG_END )
                  IF ( TIME >= 10 ) REPORT_REFRESH () -- manual refresh
            END WHILE
END EXPERIMENT
```

10.6. Functions In Experiments

The experiment allows us to use functions in the experiments. The functions can be reused and they can be defined in the library or they can be defined in the experiment file.

Basically both types of functions are identical, but there are two main differences:

- The experiment functions will not be visible in the library
- The experiment functions can directly access the partition variables and they can also integrate the model (INTEG functions) and steady states (STEADY())

The modeller can also use an experiment function to be passed as an argument to EL and external functions. This is a very powerful feature and classically used for optimisation of parameters. An example is given below. This case shows how to pass an experiment function as an argument of an external FORTRAN function named makeOptimisation. This function will call the EL function optimiseFun() directly to calculate the variable c2.v after a transient calculation by modifying the parameter c2.i:

```
-- predeclaration of a external function coded in FORTRAN
"FORTRAN" FUNCTION NO_TYPE makeOptimisation( FUNC_PTR FUN)
```

```
-- EL function for optimisation
FUNCTION NO_TYPE optimiseFun( IN REAL p1,OUT REAL x )
     BODY
          c2.i = p1
          TIME= 0
          TSTOP= 20
          CINT= 0.1
          INTEG()
          x = c2.v
     RETURN
END FUNCTION

EXPERIMENT exp1 ON test.default
     BODY
        -- to be called from the FORTRAN routine
       makeOptimisation(optimiseFun)
END EXPERIMENT
```

This is an example that gives us an idea of the flexibility provided by EcosimPro experiments to be connected to any external software.

10.7. Initialization Issues

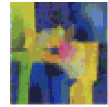
We have explained that when using EL there are several places where the user can initialize variables. This section will outline the priorities of these initializations.

In general, the order of priorities from greater to less is:

- The INIT block of the components (called by default at the beginning of INTEG() and STEADY() functions). It can be called explicitly with function EXEC_INIT()
- Set a datum in the experiment BODY
- A call RESTORE_STATE()
- The INIT and BOUND section of the EXPERIMENT
- The default value assigned when declared

For example:

```
COMPONENT foo
     DECLS
          REAL Var= 1
     INIT
          Var= 2
```



```
      CONTINUOUS
            Var' = sin(TIME)
END COMPONENT

EXPERIMENT exp1 ON foo.default
      INIT
            Var= 3
      BODY
            Var= 4
            INTEG_TO(10,0.1)
END EXPERIMENT
```

When EcosimPro reaches the integration method INTEG_TO, the next sequence of initializations has been produced:

1. Var= 1 (from declaration)
2. Var= 3 (INIT block of the experiment)
3. Var= 4 (experiment BODY)
4. Var= 2 (at the beginning of the INTEG_TO() function, it automatically calls the INIT block of components)

Hence, the good value of Var at the beginning of integration will be 2. This shows how important the usage of INIT block is in components, since this block has maximum priority to initialize variables. EXEC_INIT() function can be used to force the execution of the INIT block of components "manually".

10.7.1. EXEC_INIT() Function

The EXEC_INIT() function can be used from any EcosimPro experiment to explicitly call the INIT blocks of components. This call will inhibit the default call from INTEG() or STEADY() family of functions afterwards. For example, given the following component and experiment:

```
COMPONENT foo
DECLS
     REAL x= 0.0
INIT
     x= 1.0
CONTINUOUS
     x' = sin(TIME)
END COMPONENT

EXPERIMENT exp1 ON foo.default0
INIT
```

```
      x= 2
BODY
      EXEC_INIT() -- execute the INIT block of components
      x= 5.0
      STEADY()
END EXPERIMENT
```

This experiment shows how to call for initialization of the components using EXEC_INIT(), then changes a value (x=5.0) and then calculates a steady state. When calculating the steady, EcosimPro will not try to execute the INIT block of the components again, since it was already called from the EXEC_INIT() function. The sequence of values of x is according to the following table:

1. Is a hidden call made to the INIT block of the component (x= 1.0)?
2. Is a hidden call made to the INIT block of the experiment (x= 2)?
3. Is the default value set from declaration time (REAL x= 0.0)?
4. Change the SET_INIT_ACTIVE to... (if blank, there is no change)

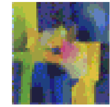
Statement	Value of x after execution	Comments	(1)	(2)	(3)	(4)
BODY	2.0	From INIT of experiment		YES	YES	TRUE
EXEC_INIT()	1.0	From INIT of component	YES			FALSE
x= 5.0	5.0	Set the value manually				
STEADY()	1.0	After calculation of STEADY				FALSE
END EXPERIMENT	1.0	After calculation of STEADY				

If we want a new execution of the INIT block of the component at the beginning of the STEADY() function, we have to use the function SET_INIT_ACTIVE() to change the flag.

10.7.2. SET_INIT_ACTIVE() Function

The SET_INIT_ACTIVE() function allows users to activate or deactivate the flag to call the component INIT blocks. For example, if we want to activate the flag, in the last example after calling the EXEC_INIT() function we can write:

```
EXPERIMENT exp1 ON foo.default0
INIT
      x= 2
BODY
```



```
     EXEC_INIT() -- execute the INIT block of components
                 -- and deactivate flag
     x= 5.0
     SET_INIT_ACTIVE(TRUE) -- activate flag again
     STEADY()
END EXPERIMENT
```

In this example the value x will be at the different points:

Statement	Value of x after execution	Comments	(1)	(2)	(3)	(4)
BODY	2.0	From INIT of experiment		YES	YES	TRUE
EXEC_INIT()	1.0	From INIT of component	YES			FALSE
x= 5.0	5.0	Set the value manually				
SET_INIT_ACTIVE(TRUE)	5.0					TRUE
STEADY()	1.0	After calculation of STEADY	YES			FALSE
END EXPERIMENT	1.0	After calculation of STEADY				

It is important to note the behaviour of some functions with respect to this flag:

- INTEG family of functions: if the flag is TRUE they call by default the INIT blocks and set the flag to FALSE
- EXEC_INIT(): if the flag is TRUE it explicitly calls the INIT blocks and sets the flag to FALSE
- RESTORE_STATE(): it sets the values of variables from a file and it sets the flag to FALSE
- RESET(): it calls the initialization made by users in the declarations, executes the INIT block of experiments and sets the flag to TRUE

The following is a more sophisticated example with some of these functions:

```
EXPERIMENT exp1 ON foo.default
DECLS
INIT
   -- State variables
  x= 2.0
BOUNDS   -- set expressions for boundary variables: v = f(t,...)
BODY
     EXEC_INIT() -- execute the INIT block of components
     x= 5.0
     SET_INIT_ACTIVE(TRUE) -- activate flag again
     STEADY()
     RESET()
```

```
      STEADY()
      RESTORE_STATE("miFile.rpt") -- asume it contains x= 9.0
      STEADY()
END EXPERIMENT
```

The value of x at the different states will be:

Statement	Value of x after execution	Comments	(1)	(2)	(3)	(4)
BODY	2.0	From INIT of experiment		YES	YES	TRUE
EXEC_INIT()	1.0	From INIT of component	YES			FALSE
x= 5.0	5.0	Set the value manually				
SET_INIT_ACTIVE(TRUE)	5.0					TRUE
STEADY()	1.0	After calculation of STEADY	YES			FALSE
RESET()	2.0	From INIT of experiment		YES	YES	TRUE
STEADY()	1.0	After calculation of STEADY	YES			FALSE
RESTORE_STATE("miFile.rpt")	9.0	Read from file				FALSE
STEADY()	1.0	After calculation of STEADY				FALSE
END EXPERIMENT	1.0	After calculation of STEADY				

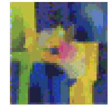
10.7.3. EXEC_EVENTS() Function

The function EXEC_EVENTS() executes every event and propagates until no more WHEN or Timed Assignment are active.

10.8. Changing the Precision of the Output Data

At any point in the experiment, EcosimPro users can decide how many significant numbers they require for all the output results. Accordingly, they must use the overall variable RDIGITS. For example:

```
EXPERIMENT exp1 ON compo1.default
     BODY
          REPORT_TABLE("rpt"," * ")
             -- Use 5 digits of precision
          RDIGITS = 5
          INTEG_TO(10,0.1)
             -- use 13 digits of precision
          RDIGITS= 13
```



```
        INTEG_TO(20,0.1)
END EXPERIMENT
```

By default RDIGITS is equal to 9, which under normal conditions is sufficient precision for the majority of engineering problems. The range of normal values for RDIGITS varies between 5 and 15 significant numbers.

10.9. Changing the Moment to Refresh Data

There are times when users want to change the exact moment at which the integration results of a model are obtained. By default EcosimPro will provide results to each communication interval and to each discrete event which causes the model to vary.

If we want to change the refresh moment, we can change a global experiment variable called "REPORT_MODE". The allowable values of the variable are:

- **IS_EVENT**: this is the default mode. Results are refreshed for each communication interval and discrete event
- **IS_CINT**: in this mode, the results will only be given at precise communication intervals
- **IS_STEP**: in this mode, all the results obtained will be given: communication intervals, discrete events and internal integration steps

For example, let's assume we are integrating a model from 0 to 0.5 seconds with a communication interval of 0.1 seconds. And let's suppose that at some moment an event is detected at 0.18 and 0.34 seconds which causes a ZONE to change or WHEN to appear. Depending on the value assigned to REPORT_MODE, EcosimPro will provide:

```
REPORT_MODE = IS_EVENT (default mode)
```

TIME	EVENT	RESULT
0.00	NO	0.234
0.10	NO	0.345
0.18	YES	0.456
0.20	NO	0.498
0.30	NO	0.497
0.34	YES	0.456
0.40	NO	0.478
0.50	NO	0.488

```
REPORT_MODE = IS_CINT
```

TIME	EVENT	RESULT
0.0	NO	0.234
0.1	NO	0.345
0.2	NO	0.498
0.3	NO	0.497
0.4	NO	0.478
0.5	NO	0.488

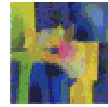
```
REPORT_MODE = IS_STEP
```

TIME	EVENT	RESULT
0.00	NO	0.234
0.000025	NO	0.235
0.000050	NO	0.236
0.000700	NO	0.249
0.001500	NO	0.270
0.05	NO	0.290
0.50	NO	0.310
0.10	NO	0.345
0.18	YES	0.456
0.20	NO	0.498
0.30	NO	0.497
0.34	YES	0.456
0.40	NO	0.478
0.50	NO	0.488

Using IS_STEP mode all the values are obtained. This option could be of great interest if we wish to analyse a system that does not converge or if we wish to investigate why the solver changes the integration time.

The way to use it is shown in the following example:

```
EXPERIMENT exp1 ON comp.default
      BODY
            REPORT_TABLE("rpt ", " * ")
            REPORT_MODE = IS_STEP
```



```
            INTEG_TO (0.5, 0.1)
            REPORT_MODE = IS_EVENT
            INTEG_TO (1.0, 0.1)
END EXPERIMENT
```

This example shows how the value of the variable can be changed, even during the experiment. In this example, integration up to 0.5 seconds will give all the integration results, including the internal steps; but then, integration from 0.5 to 1 second will only give us the results of each communication interval and of significant events.

10.10. NSTEPS: A Way to Reduce the Integration Step

The modeller will sometimes want to work with a specified integration step which does not coincide with the communications interval (CINT). This situation typically arises when fixed step numeric algorithms are used such as the explicit 4th order Runge-Kutta method.

In such cases, EL supplies a variable called NSTEPS which allows the communications interval (CINT) to be divided into a series of internal steps. For example, let us take an experiment like the following:

```
EXPERIMENT exp1 ON mod.default
      BODY
            IMETHOD = RK4  -- use fixed step integration Runge-Kutta
            TIME = 0
            TSTOP = 3
            CINT = 1
            INTEG()
END EXPERIMENT
```

In this case the intermediate integration values will be reported every second of integration and integration will also be performed every second.

Since the algorithm is blind when it comes to selecting a suitable integration step, convergence sometimes does not take place and we have to reduce the integration step. However, at the same time, we do not want to obtain all the intermediate values but we still want to obtain the results every specified number of communication intervals.

For this purpose we can divide the CINT into a number of intermediate steps using NSTEPS. The final integration interval will be:

```
InternalIntegrationStepSize = CINT / NSTEPS
```

If in the last example we write:

```
…
CINT = 1
NSTEPS = 10
INTEG()
```

this means that the real integration step will 0.1 seconds, but results will only be reported every second.

There is no sense in using the variable NSTEPS when using variable step integration methods like DASSL, since the actual algorithms dynamically determine the most appropriate internal step size. This is why it is recommended that NSTEPS be used only when explicit methods like RK4 are used.

10.11. Save and Restore Snapshot of Simulations

In the experiment the modeller can use two functions to save a snapshot of the variables (SAVE_STATE) and then to recover the same state (RESTORE_STATE).

10.11.1.1. SAVE_STATE (“fileName") Function

This function creates a file with the name "fileName" and writes the status of all the variables of a mathematical model in accordance with the format

```
Variable = value
```

When the file is read, the value of each of the variables at the time of simulation can be viewed. If required, the user can modify values of variables for subsequent reuse.

Care must be taken to ensure that only suitable data types are introduced.

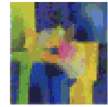
10.11.1.2. SAVE_VARS() Function

The function SAVE_VARS() can be used to save to a file a set of variables whose name matches one or several regular expressions. The syntax is:

```
FUNCTION NO_TYPE SAVE_VARS(IN STRING fileName, -- file name
     IN STRING pattern) -- regular expressions
```

where:

- fileName is the name of the file to which it is to be saved
- pattern is the set of regular expressions to filter the variables



It can be used like any other sequential function. For example:

```
EXPERIMENT exp ON foo.default
BODY
     SAVE_VARS("fileOutput.rpt"," pressure v*] x*")
END EXPERIMENT
```

This call will save in file "fileOutput.rpt" all variables with names:

- containing "pressure"
- beginning with 'x' and ending with a ']' character
- beginning with 'x'

The output produced could be:

```
pressure= 34.234e6
v[1]= 2234.35
v[2]= 3435.343
x1= 3.43435
x2= 4.4545
```

If at any moment we want to restore the variables to these values, we can use the function RESTORE_STATE() from this file.

10.11.1.3. RESTORE_STATE () Function

This function is the opposite of the previous one. When run on a "fileName" type file, it restores all the variables read to the value contained in the file. If a variable name is incorrect or a value is not accepted, the system generates an error message.

This feature allows intermediate statuses of integration to be saved and used later as a basis for subsequent integration.

It can also be used to initialize variables textually in a way which is easier to trace.

Examples:

- Experiment 1: It calculates a steady state and then saves it in file "mySteady"

```
EXPERIMENT exp1 ON compo.default
     BODY
          STEADY()
          SAVE_STATE("mySteady")  -- save state
END EXPERIMENT
```

- Experiment 2: It restores the previous state saved in file “mySteady” and then it integrates the model

```
EXPERIMENT exp1 ON compo.default
     BODY
             -- restore a previous state
          RESTORE_STATE("mySteady")
          TIME = 0
          TSTOP = 100
          CINT = 1
          INTEG()
END EXPERIMENT
```

The file “mySteady” can be something like:

```
Xspe = 4.3343445
Xspi = 5.26554332
Xspo = 2345.3434E4
```

NOTE: These functions read and write all variables (EXPLICIT, BOUNDARY, DERIVATIVE, DYNAMIC, DISCRETE, etc) with the exception of constants.

10.12. Synchronising Time with Real Time

With EcosimPro, integration is possible in real time but only in models in which the calculation time is less than the real time, because if the calculations take longer they would never be able to catch up with the real time.

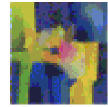
For example, with an integration from 0 to 100 seconds and a calculation time of 10 seconds we can instruct EcosimPro to integrate the model, waiting always for the EcosimPro time clock to coincide with the integration time clock. In this case integration would also take 100 seconds, which means that EcosimPro would be on standby for 90 seconds.

If the calculation time were 150 seconds, integration in real time would be impossible because it is the minimum time (150 sec) necessary for the calculations.

EcosimPro has a FOLLOW_RT variable which is normally set to FALSE and means that the calculation time clock does not follow the integration time clock. If it is set to TRUE, it will make the calculation time at least the same as the integration time.

It is easy to use; in the experiment, before initializing integration, the status of the variable is changed:

```
EXPERIMENT exp1 ON model.part1
```



```
     BODY
          FOLLOW_RT = TRUE  -- follow the real time clock
          INTEG_TO(100, 1)
END EXPERIMENT
```

10.13. Disabling the Root Finder Mechanism

Sometimes users do not want EcosimPro to calculate the exact times at which the events occur; instead, they may want to take as the valid time the right-hand extreme of the interval just integrated.

Why would users want to disable it? Mainly because in real time systems too much time is spent searching for the exact cut-off times and an approximation to the final interval time would be more than sufficient to continue with the integration.

In EcosimPro the root finder mechanism can be disabled with the experiment variable FIND_EVENTS.

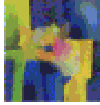
The default setting of this variable is TRUE, which means that a search is made for the roots of the exact times at which the events occur. To disable it, simply set it to FALSE.

It is used from the experiment, for example:

```
EXPERIMENT exp1 ON model.part1
     BODY
          FIND_EVENTS = FALSE
          INTEG_TO(100, 1)
END EXPERIMENT
```

Do not confuse this variable with the REPORT_MODE variable set to the value IS_CINT. Using the latter instructs EcosimPro to report only the values in the communication intervals, while internally it is calculating all the roots of the exact times at which the events occur.

The variable FIND_EVENTS is normally used when integration is performed by fixed step algorithms, such as the Runge-Kutta method (eg, RK4).



10.14. Getting Statistics of Jacobian Evaluations and Residues Functions

During the experiment, EcosimPro users can ascertain the number of Jacobian evaluations of status variables that have been performed with the functions getJacobianEvals. The signature is simple:

```
INTEGER getJacobianEvals()
```

They can also determine the actual number of times the residues function has been evaluated with the function getResiduesEvals() with signature:

```
INTEGER getResiduesEvals()
```

The following is a simple example of its use:

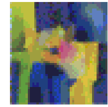
```
EXPERIMENT exp1 ON model.part1
     DECLS
          INTEGER nJacob
          INTEGER nResidues
     BODY
          INTEG_TO( 100, 1)
             --get some statistics
          nJacob = getJacobianEvals()
          nResidues = getResiduesEvals()
             --print statistics
          PRINT("Jacobian evaluations = $nJacob,
               Residues evaluations = $nResidues")
END EXPERIMENT
```

In addition, EcosimPro has other symmetrical functions to obtain the values from calls for Jacobian evaluations and calls for residues evaluations, but they obtain the incremental value from the last call. These functions are:

```
INTEGER getJacobianEvalsIncr()
INTEGER getResiduesEvalsIncr()
```

These functions would be typically used to calculate the number of times calls are made for Jacobian evaluations and residues evaluations in each communication interval. For example:

```
EXPERIMENT exp1 ON model.part1
     DECLS
          INTEGER nJacob
          INTEGER nResidues
     BODY
```



```
          WHILE (INTEG_CINT()!= INTEG_END)
               nJacob = getJacobianEvalsIncr()
               nResidues = getResiduesEvalsIncr()
                  -- print statistics
               PRINT("Jacobian evaluations = $nJacob,
                    Residues evaluations = $nResidues")
          END WHILE
END EXPERIMENT
```

10.15. Change the level of debugging

There is a function to change the debug level of the experiment when running:

```
"C++" FUNCTION NO_TYPE setDebugLevel(INTEGER level)
```

The function setDebugLevel() can be called at any time in the experiment to change the level of debugging. This level can be changed directly by changing the DEGUG_LEVEL variable; however, it is recommended that the function be called because it reports a change in the debug level during the experiment (if you use the variable directly you will get no messages).

Usage of the function is simple; for example:

```
EXPERIMENT exp1 ON test.default
BODY
     setDebugLevel( 3 )
     STEADY()
END EXPERIMENT
```

There are four levels of debugging. At each level, different information is provided in order to debug a model:

- ☐ LEVEL 0: No trace at all
- ☐ LEVEL 1 (Default):
  - Trace the reading of the symbols table
  - Trace the initial status of ZONE and IF statements
  - Trace the active WHENs
  - Trace the evolution of transient calculations (steps)
  - Trace the basic transient information: header and footer with big statistics: time consumed and number of calls to the residue functions

- Trace the basic steady information: header and footer with big statistics: time consumed, Jacobians, iterations, maximum error and number of calls to the residue functions
- WRITE and PRINT statements
- Messages about debug level, solving internal algebraic boxes, etc
- Non-convergence messages

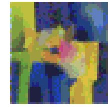
- LEVEL 2:
  - All LEVEL 1 info
  - Trace all non-convergent steady calculations. It will produce a full report in the log file (HTML format) with the evolution of the Jacobian, residues, Broyden evaluations, etc
  - Trace the evolution of ZONE and IF statements
  - Trace the begin and end execution of INIT blocks of components
- LEVEL 3:
  - All LEVEL 2 info
  - Trace the convergent steady calculations will produce a full report (similar to the non-convergent case)
  - Print all variable values when the problem does not converge
  - Print warnings about non-repeatability of the residue function and non-linearity of the Jacobian

10.16. Debugging mathematical models

There are three ways to check mathematical model robustness when integrating with DASSL. They are:

- Check any step reduction when integrating the equations
- Check that the residues function is repeatable for the same inputs
- Check the nonlinearities of the Jacobian matrix during the execution

These points are explained in the following paragraphs.



10.16.1. Check Integration Step Reduction

EcosimPro can print out a warning each time the DASSL integrator reduces the integration step. This gives an idea of a potential problem when integrating the equations system.

This check is only carried out when the DEBUG_LEVEL is set to 3 or greater and the variable WARN_STEP_REDUCT is TRUE. By default the variable is set to FALSE. Use of the variable is only required to activate/deactivate the messages.

For example, given the model:

```
FUNCTION REAL FDIS (IN REAL x)
   DECLS
       REAL y
   BODY
       IF (x < 5.333) THEN
           y = 1
       ELSE
           y = 2000 --introduces a big discontinuity
       END IF
       RETURN y
END FUNCTION

COMPONENT test
DECLS
     REAL x
CONTINUOUS
     x' = FDIS(x)
END COMPONENT
```

With the experiment:

```
EXPERIMENT exp1 ON test.default
DECLS
BODY
     REPORT_TABLE("rep","*")
     DEBUG_LEVEL= 2
     TIME= 0
     CINT= 0.1
     TSTOP= 15
     INTEG()
END EXPERIMENT
```

Running this experiment in EcosimPro, the following warning is generated at TIME 5.23:

```
**********  WARNING level 0 (code 1124)  **********
```

```
At time 5.23 integration timestep divided by 4. Worst variable: x
```

To deactivate message set WARN_STEP_REDUCT to FALSE

This means that the integrator encountered problems integrating the model and it had to reduce the integration step 4 times. Moreover, the variable causing the problems is x. From now on, the modeller has to solve the problem.

In you want to deactivate the message, you can write:

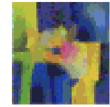
```
BODY
      DEBUG_LEVEL= 2
      WARN_STEP_REDUCT= FALSE
...
```

10.16.2. Check Residues Function Repeatability

EcosimPro can detect whether the residues function is a true independent function returning always the same outputs given the same inputs. This test gives the modeller an important clue to avoid this situation since a non-repeatable residues function can be a severe problem when it comes to making the system of equations converge.

There is a global variable to activate/deactivate this warning; it is CJVITG. By default this variable is set to FALSE but DEBUG_LEVEL must be 3. For example given the model:

```
REAL GG= 1.0  -- global variable

FUNCTION REAL mmm(IN REAL v)
BODY
      GG = GG + 1.0          -- increase global variable
      RETURN v + cos(v) + GG  -- this is wrong, there is a side effect adding
GG
END FUNCTION

COMPONENT test

DECLS
      REAL x,y,z
CONTINUOUS
      y'= sin(TIME)
      x'= cos(TIME) + mmm(y) -- this equation will have problems!
      z* x= z' - y
END COMPONENT
with the experiment:
EXPERIMENT exp1 ON test.default

BODY
      DEBUG_LEVEL= 2
```



```
      TIME= 0
      CINT= 0.0001
      TSTOP= 0.0001
      INTEG()
END EXPERIMENT
This model produces a warning at time 1E-7:
**********  WARNING level 0 (code 1123)  **********
At time 1e-007 the residues function is not repeatable in equation
[2] x' = cos(TIME)  + DEFAULT_LIB__mmm(y)
```

Possible causes:

- Use of global data in a function
- A boundary condition is a function of a model variable

To deactivate message set CJVITG to FALSE

This means that the equation [2] generates a nonrepeatable value which is true since the function mmm() uses a global variable to produce the return value. This means that this function is not independent of the input data. The integrator encounters convergence problems.

This error is typically a severe warning and you should check your model to guarantee that the residues function becomes a true independent function.

The check is carried out by calling the residues function twice with the same inputs and checking that the residues obtained are correct. The integrator cannot necessarily solve the problem when this warning is produced since the stiff problem is fairly forgiving of the accuracy of the Jacobian.

If the user wants to deactivate the message, he can write:

```
BODY
      DEBUG_LEVEL= 2
      CJVITG= FALSE
...
```

10.16.3. Check the Nonlinearity of the Jacobian

EcosimPro can check how nonlinear the Jacobian matrix of the model is when integrating with the DASSL method (dense or sparse version). This test helps the modeller to understand in some cases why the model does not converge since it gives a clue to the nonlinearity of the model and some hints about the equation causing the problem.

There is a global variable to activate/deactivate this warning; it is TSMITG. By default this variable is set to FALSE but to force EcosimPro to print this warning we need to set DEBUG_LEVEL to 3. For example, given the model:

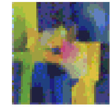
```
FUNCTION REAL FDIS (IN REAL x)
   DECLS
       REAL y
   BODY
       IF (x < 5.333) THEN
           y = 1
       ELSE
           y = 2000
       END IF
       RETURN y
END FUNCTION

COMPONENT test
DECLS
     REAL x
CONTINUOUS
     x' = FDIS(x)
END COMPONENT
with the experiment:
EXPERIMENT exp1 ON test.default
DECLS
BODY
     DEBUG_LEVEL= 2
     IMETHOD= DASSL_SPARSE
     TIME= 0
     CINT= 0.1
     TSTOP= 15
     INTEG()
END EXPERIMENT
```

This model produces the warning:

```
**********  WARNING level 0 (code 1122)  **********
At time 5.333 Jacobian nonlinear measure is 0.5
Variable name: x'  Variable type: derivative
DAE equation with nonlinearity problems:
 [1] x' = SW_TESTS_FDIS(x)
```

To deactivate message set TSMITG to FALSE or set threshold TJNITG to a big value

It means that at time 5.333 EcosimPro has detected a nonlinearity in the model to the value of 0.5 when perturbing variable x' and the affected equation is [1]. In this example the reason is easy to understand: the function FDIS is producing much discontinuity at



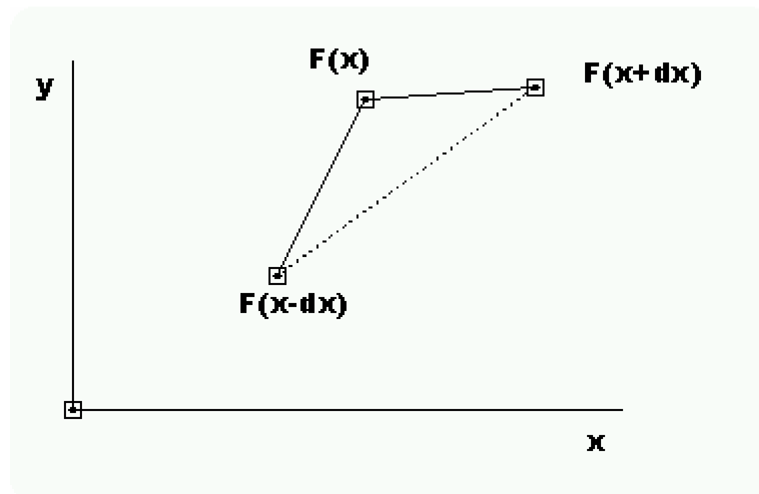
this time; in other real mathematical models it will not be so easy, but at least it is a clue as to the whereabouts of the root of the problem.

If we want to deactivate the message, we can write:

```
BODY
     DEBUG_LEVEL= 2
     TSMITG= FALSE
```

To detect the nonlinearity EcosimPro perturbs the Jacobian, first positively (F(x+dx)) and then negatively (F(x-dx)), computing the derivative vector for each value. With this information the middle point of the derivative (F(x)) is interpolated to see whether it lies in line with the positive and negative derivatives.

The following figure shows the state F(x) with a value in the middle of the box where the perturbation results in a smaller slope in the positive direction than in the negative direction to F(x-dx). If the system were linear, then the middle point would lie in the middle of the other points; ie, (F(x+dx)+F(x-dx)) / 2. Any deviation of F(x) from this midpoint implies a nonlinearity.



The nonlinear measurement is calculated with the expression:

$$\frac{|F(x) - 0.5\,(F(x+dx) + F(x-dx))|}{|F(x+dx) - F(x-dx)|}$$

If the expression is 0, it means that it is linear; other values show the nonlinearity of the model. EcosimPro prints the warning by default only when the nonlinear measurement is greater than 0.1. The user can change this threshold by changing the global variable TJNITG to other values. For example:

```
BODY
     DEBUG_LEVEL= 2
     TJNITG= 0.2
...
```

10.17. Helping DASSL_SPARSE method

When solving DAE systems in EcosimPro using DASSL_SPARSE, there is an analysis of the Jacobian matrix by decomposing an LU.

EcosimPro uses an algorithm to reuse the LU decomposition calculated with the first call. In successive calls it reuses the same pivots. EcosimPro also detects that the pivots are valid every time; if not, the LU decomposition is calculated again. This avoids a lot of calculations on each iteration to decompose the matrix.

The user can change the factor which determines the threshold to decide whether or not to recalculate the LU decomposition. This factor is called DASSL_SPARSE_U. It determines the balance between pivoting for sparsity and for stability, values near zero emphasizing sparsity and values near one emphasizing stability. The default value is 0.25. The value can vary from 0 to 1. For instance:

```
BODY
     DASSL_SPARSE_U= 0.1
```

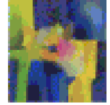
Anyway, it should not be common to have to change this parameter, which is considered an advanced feature of EcosimPro's mathematical kernel.

10.18. Residues Function Calls

We can directly call the residues function from the experiment. Sometimes it is in our interest to execute this function just once at the beginning so that we know the values of some variables.

We can do this in EcosimPro by calling the function FRES() from the experiment body. For instance:

```
EXPERIMENT exp1 ON model.part1
     BODY
          FRES()
          PRINT("The value of x is $x")
END EXPERIMENT
```



This experiment executes the residues function just once and then prints the value of the variable x.

11. CLASSES

11.1. 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.

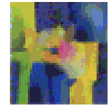
Classes are normally used in EL to support the modelling of complex systems where the use of functions is sometimes improved if all the functions referring to the same utility are grouped together and share memory by means of common variables.

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
```

The class is named by means of an identifier. A class can contain the following information:

- **IS_A**. This reserved word is used to indicate that this class inherits from one or more other classes. When multiple inheritance occurs, the parents are separated by commas
- **DESCRIPTION**. A string can be used to describe the class. If it takes up more than one line, use the separator \ to indicate that the string continues onto the next line



- **DECLS**. This is a block where class variables can be declared. All the basic EL types can be used (eg, REAL)
- **OBJECTS**. This is a block where objects from other classes can be encapsulated
- **METHODS**. This block describes the internal methods of the 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()".

11.2. 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.

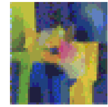
```
CLASS statusClass   "a status class"
   DECLS
          PRIVATE BOOLEAN status
   METHODS
         METHOD NO_TYPE setStatus(IN BOOLEAN sta)
         BODY
           status= sta
         END METHOD
      METHOD BOOLEAN getStatus()
         BODY
           RETURN status
         END METHOD
END CLASS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

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.



11.3. 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?
```

An explanation for each element is given below:

- **PRIVATE**. To keep the variable from being visible to others, the qualifier PRIVATE is put before it. Otherwise, it would be public and its value could be both read and changed directly. In object-oriented modelling, it is very common not to make variables directly visible, but to create "get" and "set" methods to initialize or obtain their value instead. However, this can sometimes be awkward, so it is best to give direct access to the variable (in that case, do not write PRIVATE)
- **CONST**. If it is a constant whose value should never change, write CONST before the variable type
- **data_type**. It can be any of the basic EL types (eg, REAL)
- **names**. You can declare one variable or several in the same declaration. If there are more than one, they must be separated by commas
- **init_expression**. If they are to be given an initial value, it will be given in this section
- **description**. The variable can be described with a string typically containing the units

Examples of declarations are:

```
PRIVATE REAL x = 1
```

The variable x has no outside accessibility and its initial value is 1.

```
PRIVATE CONST REAL x = 1
```

In this case, x is a private constant and its value can never change (moreover, the compiler will require us to give it a value in the declaration).

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

This gives a description of the public variable speed and its units.

11.4. 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?
```

The meaning of each key is:

- **PRIVATE**: An object can be declared as private (by default they are not) and they are not visible in the object (but they can be used locally)
- **Names**: many objects can be defined in the same line. Array objects are allowed (eg, MyClass object[2,4,7])
- **STRING_VALUE**: a description of the object can be added

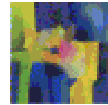
It is quite similar to declaring a variable in a component or port, but with some differences:

- The objects cannot be initialized with an initial value (like variables)
- The creation of constant objects is not allowed

These restrictions have been imposed to simplify using classes (especially for initializing complex objects that required creating special constructors)

Valid declarations are:

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



11.5. 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 )?
```

Its syntax is similar to that of functions (see the chapter on functions for details on each block) with the following special features:

A method starts and ends with the word METHOD instead of FUNCTION

A method can be kept private from the class by using the word PRIVATE, whereas a function is always public and visible in the library. By default they are public

A method can make free use of local variables, variables of the class they are in and global variables. A function can only access local variables and global variables in the library

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
```

11.6. 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
```

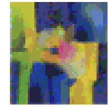
The first case is an example of calling the public method setValue and giving it an entire argument. The second directly assigns a value of 4 to the object variable object1 called speed.

It may be that an object in turn calls another internal public object. The call would be linked by once again using the point (.) operator, for example.

```
object1.foo.setValue(4)
```

This rule can be applied successively for public objects contained in the object. 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.incr(2)  -- call incr() method
            RETURN v
```



```
    END METHOD
END CLASS
```

This would produce an error in the compiler.

```
Compiling
******** Error 1 (code 780) in file
"D:\EcosimPro\USER_LIBS_ALL_RAO\ECLSS2_MODELS\sources\tclass.el" at line 47,3
********
ex.incr(2)
^
Trying to access private attribute/method "incr"
```

Given that the method incr() is private, it cannot be accessed from outside. However, it would be correct to write:

```
ex.popValue(2)
```

11.7. 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 Options->Miscellaneous. 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
```

Care must be taken when using them within the CONTINUOUS block of components, as this tool arranges the equations and thus changes the order of execution of the calls to object methods (as it does when using functions). Normally, operations with these types of objects should go in the INIT, DISCRETE blocks of components and BODY blocks of functions and experiments.

11.7.1. 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 the list of available methods:

- Return the number of variables

```
INTEGER getNumberVars ()
```

- Return the variable name associated with index i

```
STRING getVarName (IN INTEGER i)
```

- Check whether a variable exists

```
BOOLEAN existsVariable(IN STRING name)
```

- Return the variable type (in string format such as “REAL”, “INTEGER”, “BOOLEAN”, etc)

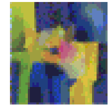
```
STRING getVarTypeStr (IN STRING name)
```

- Return the variable category (in string format such as “DYNAMIC”, “ALGEBRAIC”, “DATA”,”DERIVATIVE”, etc)

```
STRING  getVarCategoryStr (IN STRING name)
```

- Return the actual value of a REAL variable

```
REAL    getValueReal  (IN STRING name)
```



- Set the REAL value to value v (returns FALSE if not found)

```
BOOLEAN setValueReal  (IN STRING name, IN REAL v)
```

- Return the actual value of an INTEGER variable

```
INTEGER    getValueInt  (IN STRING name)
```

- Set the INTEGER value to value v (returns FALSE if not found)

```
BOOLEAN setValueInt  (IN STRING name, IN INTEGER v)
```

- Return the actual value of a BOOLEAN variable

```
BOOLEAN    getValueBool  (IN STRING name)
```

- Set the BOOLEAN value to value v (returns FALSE if not found)

```
BOOLEAN setValueBool  (IN STRING name, IN BOOLEAN v)
```

- Return the actual value of a STRING variable

```
STRING    getValueString  (IN STRING name)
```

- Set the STRING value to value v (returns FALSE if not found)

```
BOOLEAN setValueString  (IN STRING name, IN STRING v)
```

- Return the actual value (in string format) of an Enumerative variable

```
STRING    getValueEnum  (IN STRING name)
```

- Set the Enumerative value to value v (returns FALSE if not found)

```
BOOLEAN setValueEnum  (IN STRING name, IN STRING v)
```

- Return the actual value (in string format) of any variable of any type

```
STRING    getValueAsString(IN STRING name)
```

The following code gives an example of valid calls for the partition “transient” applied to component “aircraft”: After producing the partition, a class named “aircraft_transient” is generated automatically and it is possible to use it from any OBJECTS block. The following example uses it from another component.

```
COMPONENT foo
DECLS
      STRING tvar
   STRING tcategory
OBJECTS
      aircraft_transient obj
INIT
      -- obtain the total number of variable of this model
```

```
      nvars = obj.getNumberVars()

      -- print all variable names
         FOR(i IN 1,nvars)
            WRITE("variable %d= %s\n", i , obj.getVarName(i))
         END FOR

      -- set a value for variable c1.c if exits
      IF ( obj.existsVariable("c1.c") )
             obj.setValueReal("c1.c", 3.45)

      -- get the variable type (e.g. "REAL", "INTEGER", etc.)
      tvar = obj.getVarTypeStr("c1.p")

      -- get the variable category (e.g. "EXPLICIT", "BOUNDARY", etc.)
      tcategory= obj.getVarCategoryStr("c1.p")
END COMPONENT
```

For arrays, you can use a set of specific functions that make them easier to use by letting the arrays be managed globally instead of element by element.

- Get a 1D array in v[] passing the dimension and return false if not found

```
BOOLEAN getArray1D(IN STRING name, OUT REAL v[],IN INTEGER dim1)
```

- Set a 1D array

```
BOOLEAN setArray1D(IN STRING name, IN REAL v[],IN INTEGER dim1)
```

- Get a 2D array in v[] passing the two dimensions

```
BOOLEAN getArray2D(IN STRING name, OUT REAL v[],IN INTEGER dim1, ,IN INTEGER
dim2)
```

- Set a 2D array passing the two dimensions

```
BOOLEAN setArray2D(IN STRING name, IN REAL v[],IN INTEGER dim1)
```

- Get a 3D array in v[] passing the 3 dimensions and return false if not found

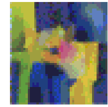
```
BOOLEAN getArray3D(IN STRING name, OUT REAL v[],IN INTEGER dim1, IN INTEGER
dim2,IN INTEGER dim3)
```

- Set a 3D array

```
BOOLEAN setArray3D(IN STRING name, IN REAL v[],IN INTEGER dim1, IN INTEGER
dim2,IN INTEGER dim3)
```

These methods help save on code when initializing arrays and copying them. For example: given the component:

```
COMPONENT test
DECLS
      REAL v[3]
```



```
END COMPONENT
```

Let's imagine that we make a mathematical partition, thereby automatically creating the internal class test_default. The way to initialize the array manually would be, for example:

```
COMPONENT other
DECLS
     REAL initial[3] = { 1.2, 8.23, 9.24 }
          REAL result[3]
OBJECTS
     test_default test
INIT
     -- init the array v
     test.setValueReal("v[1]",initial[1])
     test.setValueReal("v[2]",initial[2])
     test.setValueReal("v[3]",initial[3])

         -- do some calculations here

     -- get the array
     result [1]= test.getValueReal("v[1]")
     result [2]= test.getValueReal("v[2]")
     result [3]= test.getValueReal("v[3]")

END COMPONENT
```

By using the array get/set functions, we could write something a bit simpler:

```
COMPONENT other
DECLS
     REAL initial[3] = { 1.2, 8.23, 9.24 }
     REAL result[3]
OBJECTS
     test1_default test
INIT
     -- init the array v passing the initial array and the
dimension
     test.setArray1D("v",initial,3)

         -- do some calculations here

     -- get the array into result passing the dimension
     test.getArray1D("v",result,3))

END COMPONENT
```

If the array is multi-dimensional, you only have to define the arrays of the right dimensions and use the specific functions for those dimensions. For example:

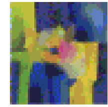
```
COMPONENT other2
DECLS
      REAL initial[3,2] = { {1.2, 8.23}, {3.2, 45.2}, {4.2, 97.2} }
      REAL result[3,2]
OBJECTS
      test2_default test
INIT
      -- init the array v
      test.setArray2D("v",initial,3,2)

         -- do some calculations here

      -- get the array
      test.getArray2D("v",result,3,2)

END COMPONENT
```

11.7.2. 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. Thus, all the functions explained in the chapter on experiments can be used with these classes as well.

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.

11.7.3. 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:

- Print simulation messages when running

```
NO_TYPE setTraceProgramme(BOOLEAN status)
```

- Return the actual status of flag for tracing the simulation

```
BOOLEAN traceProgramme()
```

□ Print simulation messages in the log file

```
NO_TYPE setLogProgramme(BOOLEAN status)
```

□ Return the actual status of flag for logging the simulation

```
BOOLEAN logProgramme()
```

□ Check assertions when running

```
NO_TYPE setWarnProgramme(BOOLEAN status)
```

□ Return the actual status of flag for checking assertions

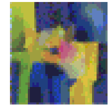
```
BOOLEAN warnProgramme()
```

11.7.4. 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, we have 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.

12. USING C++ CLASSES FROM EL

12.1. Introduction

EL allows C++ external classes to be reused in EL. To do this, a class that is to be used as an interface for the C++ classes has to be predeclared in EL. The general format to define a CLASS in EL is as follows:

```
class_def    : EXTERN? CLASS IDENTIFIER EOL*
 ( IS_A scoped_id_s EOL*)?
       STRING_VALUE?  EOL*
       ( DECLS     var_object_decl_s )?
       ( OBJECTS  class_instace_stm_s )?
          ( METHODS  extern_method_def_s )?
           END CLASS EOL*
      (INCLUDE EOL* STRING_VALUE)? EOL* (IN EOL* STRING_VALUE+ EOL)?

extern_method_header : EXTERN METHOD data_type IDENTIFIER
         '(' EOL* func_arg_decl_s ')' STRING_VALUE?
```

It is necessary to use the word EXTERN before the CLASS word and before the METHOD declarations. The method must be a pure declaration without body.

Finally, the header file must be stated (using INCLUDE directive) and the library where the C++ class can be located.

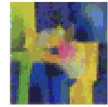
By default, the include files should be located in the “include” directory of the library and the libraries in the “lib” library. If they are not found there they will be found in the directory external/include and external/lib.

For example, if the following C++ class is used:

```
class mmath
{
public:
      double add(double a, double b);
      double subs(double a,double b);
};
```

The following EL class interface must be created:

```
EXTERN CLASS mmath
  DECLS
```



```
            REAL m_val
  METHODS
            EXTERN METHOD REAL add(IN REAL a, IN REAL b)
            EXTERN METHOD REAL subs(IN REAL a, IN REAL b)
END CLASS INCLUDE "mmath.h" IN "testCpp.lib"
```

The following elements can be distinguished in this interface:

- The word EXTERN indicates that it is an externally-defined class. The word EXTERN must also be used for each of the methods
- The class has a REAL class variable called “m_val”
- It has two public methods that may be accessed, and each of these has two real input arguments and one REAL-type return argument
- The ‘include’ file that defines the class header is in mmath.h (normally located in LIB_DIR/include)
- The library where the binary of the class is kept is in testCpp.lib (normally located in LIB_DIR/lib)

This class could then be used from any component, method, function or experiment of EcosimPro. For this purpose, the above definition would have to be compiled, and then the mmath class would appear as an additional item of the library. An EL component could be created as follows:

```
COMPONENT foo
DECLS
     REAL x
OBJECTS
     mmath m1
INIT
     x= m1.add(2,2)
END COMPONENT
```

Object m1 must be created in the OBJECTS section. To use the object later we only need to type the name of the object, a period and the name of the method, as in the case of a function. For example:

```
  m1.add(2,2)
```

This example is very simple, but the same process could be applied to more complex classes.

12.2. Valid data types

All C++ classes may be reused, with the proviso that the types of data exchanged must be those normally used by EcosimPro. The following table shows the types allowed and how they are transformed in C++:

EL Language	C++ Language
INTEGER	int
REAL	double
STRING	std::string (1)
FILEPATH	std::string (1)
BOOL	int (TRUE is 1, FALSE is 0)
ENUM	Int (each element of the enumeration has an associated number, beginning with 1)
FUNC_PTR	void (*ptr)()
OBJ_PTR	void*
NO_TYPE	void

(1) The std::string class is the ANSI C++ class for strings from the STL(Standard Template Library)

EL also allows the transfer of IN type or OUT type arguments such as:

```
EXTERN METHOD NO_TYPE add(IN REAL a, IN REAL b, OUT REAL result)
```

In this case, in C++ there must be a method like the one set out below:

```
void add(double x, double y, double *z)
```

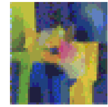
The OUT variable must be translated with a pointer-type argument in C++, whereas the IN-type variable will be a regular variable.

Arrays can also be transferred as method arguments. In this case they must be translated into C++ as OUT-type variables; ie, with a pointer-type variable (for the interface method, for example):

```
EXTERN METHOD REAL average(OUT REAL v[], IN INTEGER size)
```

There must be a C++ method such as:

```
double average(double *array, int dimension)
```



The array has been translated from a simple pointer to a double. This is the standard C++ way to transfer an array in its methods. Therefore, in this case we have also transferred an integer that provides information on the size of the array transferred.

12.3. Use of Boolean Variables

Boolean-type variables of C++ are handled as int variables in C++, so great care should be taken when dealing with them. EcosimPro will set a 1 if it is TRUE and a 0 if it is FALSE. As an example, we will define a C++ method to handle a boolean variable:

```
double calculateA (int flag);  // flag is a boolean condition
```

the corresponding EL code that needs to be used as a method would be:

```
EXTERN METHOD double calculateAverage(IN BOOLEAN flag)
```

12.4. Use of Enumerative Variables

Enumerative variables are also handled as int types in C++. The enumeration types will have to be reimplemented in C++ for complete correspondence with the enumeration types defined in EL. For example, with the following EL enumeration

```
ENUM colors= { red, blue, green }
```

This same enumeration type can be reimplemented in C++ as follows:

```
enum colors { red=1, blue=2, green=3 }
```

It must be noted that in EL the numbering starts at 1 instead of 0, as used by default in C++. It is indispensable to associate the unique number with each option because the values of the enumerated elements would otherwise not be correct.

If this new enumeration type has been defined in C++, a C++ method such as the following can be defined:

```
int selectColor(colors col);
```

In EL it would be as follows:

```
METHOD INTEGER selectColor(ENUM colors col)
```

A possible implementation of the method in C++ would therefore be as follows:

```
int selectColor(colors col)
{
     if (col == red )
          return 1;
        else if( col == blue )
          return  2;
     else if (col == green )
          return 3;
     else
          return 0;
}
```

Another option would be to define an input/output enumeration argument with the OUT option. For instance, the same method could be used to change an EcosimPro enumeration variable from C++. In the case of the following C++ method:

```
void  selectColor(enum colors *col)
{
     *col= blue; // changes color to blue
}
```

In EL, the interface would be defined as:

```
METHOD NO_TYPE selectColor(OUT ENUM colors col)
```

12.5. Use of String Variables

To define an interface as a string variable we have to use the std::string class provided in the standard C++ installation. This class is an ANSI standard contained in the Standard Template Library (STL) of the C++ compiler

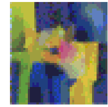
The header file for compiling is <string>

For example, for method:

```
BOOLEAN setName( IN STRING name)
```

the interface in C++ is

```
#include <string>
int setName( std::string name)
```



12.6. Use of Object Arrays

Arrays of any size can be used to define objects of external C++ classes. Their use is similar to that of the port or component arrays. For example, the following could be typed:

```
COMPONENT foo
DECLS
     REAL x
OBJECTS
     mmath m1[3,2]
INIT
     x= m1[2,2].add(2,2)
END COMPONENT
```

The only point that changes is that every time the array is used there has to be an indication of the referenced array element.

12.7. Include Files

The C++ external class must be supplied with a header or .h file that will be used by EcosimPro to compile the final model. The word INCLUDE must be used to tell EcosimPro where this file is located.

There are two possible formats to type the name of the file that indicates where EcosimPro has to look for it:

- Relative path. If the name does not include directories (eg, “<math.h>”), EcosimPro will first search for file.h in the include directory of the library (eg, “c:\EcosimPro\USER_LIBS\ELECTRIC\include”), then in the EcosimPro global include directory (eg, “C:\EcosimPro\external\include”) and finally in all the paths that contain the variable of the INCLUDE environment. This option is recommended
- Absolute path. If the name of the file appears with an absolute path (eg, “c:\myDir\file.h”) it means that EcosimPro will try to open the file from that position

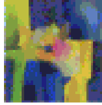
12.8. Object Libraries

As in the case of header files, EcosimPro will need to know which libraries to use when linking the final model with the C++ class used. The format is IN “lib1.lib”, where lib1.lib

is supposed to be a library compiled in C++ which contains the implementation of the class in use.

In this case, the name of the library does not admit <> symbols which were allowed by include, so only the following options are allowed:

- Relative path: If the name of the library is of the “lib1.lib” type, EcosimPro will first look in the lib directory of the library (eg, “c:\EcosimPro\USER_LIBS\ELECTRIC\lib”), then in the EcosimPro global directory in EcosimPro\external\lib (eg, “C:\EcosimPro\external\lib”), and finally in all the paths containing the variable of the LIBS environment and, failing that, it will look for it in the current directory
- Absolute path: If the name of the library appears with a disk unit, as in “c:\lib\lib1.lib”, EcosimPro will search for it in the specific directory set out in the path



13. STEADY STATE CALCULATIONS

EcosimPro provides sophisticated methods for solving steady states in a set of non-linear equations. They have been designed for easy use from the EcosimPro experiments.

This section presents the main functions and variables used to solve steady calculations.

13.1.1.1. STEADY()

This function calculates the steady state of a model (the derivatives are cancelled) and all the internal variables until dynamic equilibrium is reached.

If EcosimPro cannot calculate the steady state, the STEADY() function will return the code STEADY_NOK; otherwise, it returns STEADY_OK. For example:

```
EXPERIMENT exp1 ON model.default
BODY
      TOLTYPE= RELTOL
      TOLERANCE = 1e-4
      IF (STEADY() == STEADY_NOK)
            WRITE("EcosimPro fails to calculate the steady")
      ELSE
            WRITE("EcosimPro succeeds the steady")
      END IF
END EXPERIMENT
```

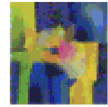
EcosimPro is capable of calculating the steady state of math models by solving the non-linear algebraic equation system that results from automatically declaring the time derivatives of all the dynamic variables to be equal to zero:

Explicit algebraic equations:

$$xi = Gi\ (t, y1, y2, ..., yn, x1, x2, ..., xi\text{-}1) \qquad i = 1, 2, ..., m$$

Implicit differential-algebraic equations:

$$Fj\ (t, y1, y2, ..., yn, 0, 0, ..., 0, x1, x2, ..., xm) = 0 \qquad j = 1, 2, ..., n$$



The steady state algorithm will compute the solution using the Newton-Raphson method:

$$\Delta y \frac{m}{Newton} = \left(\frac{\partial F}{\partial y}\right)^{-1} F\left(t, y^{m-1}, 0\right)$$

In some cases we could use the FREEZE() function to deactivate the calculation of some dynamic variables and achieve steady state.

13.1.1.2. NUMERICAL METHODS

EcosimPro provides two methods for solving non-linear equation systems: NR and MINPACK. They are both very similar, but one (NR) has a series of characters that give it more flexibility and more debugging information. Either method may be used in EcosimPro experiments.

Both methods have been developed based on the original routine hybrd() in the Minpack library (1). It tries to find the zero of a system of n nonlinear functions in n variables by modifying the Powell hybrid method. The Jacobian is then calculated by a forward-difference approximation. EcosimPro properly prepares the function residues to be used by this method.

(1) Minpack Copyright Notice (1999) University of Chicago. All rights reserved

Two of its main characteristics are the choice of correction as a convex combination of the Newton and scaled gradient directions, and the updating of the Jacobian by the Broyden method. The choice of correction guarantees (under reasonable conditions) global convergence for starting points far from the solution and a fast rate of convergence. The Jacobian is approximated by forward differences at the starting point, but forward differences are not used again until the Broyden method fails to produce satisfactory progress or a limit is reached. This means that it does not produce a Jacobian for each iteration, but only when the Broyden method fails.

EcosimPro has modified different parts of the algorithm, giving it greater robustness and capacity to report iterations, Jacobians, etc.

13.1.1.3. SMETHOD

The global variable SMETHOD is used to specify the chosen method that the following values may have.

- **NR (default)**: It is the default method. It is the preferred method since it provides a complete debug report in case of non-convergence problems. This method allows

utilisation of RELTOL, ABSTOL and FRACTOL global tolerance schemas. It enables the minimum and maximum perturbations (absolute and relative) to be changed for each algebraic variable and the minimum and maximum allowed steps (absolute and relative) for each iteration. It also provides a complete debug file about the evolution of the resolution in HTML format.

- **MINPACK**: It is directly based on ¿¿¿the function hybrib() method??? the original routine hybrd()??? from the Minpack package with a few modifications to obtain better convergence. It only allows the use of RELTOL as global tolerance criteria. It does not produce a debug report at all.
- Users can select any of the methods by changing the SMETHOD variables in the experiment; for instance:

```
SMETHOD= MINPACK
```

13.1.1.4. LOCAL RESIDUE EVALUATION

Each residue is calculated either in absolute (ABSTOL) or in fractional mode (FRACTOL). By default all residues are calculated in absolute mode. The user has the option to use ABSTOL or FRACTOL for local residues. RELTOL is valid only for global convergence criteria.

To understand both concepts (ABSTOL and FRACTOL), an explanation is given of how the residues are changed. Given an algebraic or closure equation i:

```
lhs_eqt(i) = rhs_eqt(i)
```

the residue equation is defined as:

```
FRES(i) = lhs_eqt(i) - rhs_eqt(i)
```

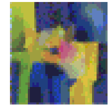
This residue is calculated in absolute mode (ABSTOL). Sometimes a user wants to normalise a residue. This can be done by means of the fractional mode (FRACTOL). In this case the residue expression is divided by a "vref" term that ideally should be close to one of the equation terms (lhs_eqt or rhs_eqt).

```
FRES(i) = ( lhs_eqt(i) - rhs_eqt(i) ) / vref
```

This returns a normalised residue. The user can provide a vref value (using the functions evalNormResidue() or setResidueInfo()). If this value has not been set, EcosimPro will use rhs_eqt(i) as vref term; in other words:

```
FRES(i) = ( lhs_eqt(i) - rhs_eqt(i) ) / rhs_eqt(i)
```

To avoid problems with the denominator (eg, it becomes zero), EcosimPro would use the lhs_eqt(i) term only when:



```
| lhs_eqt(i) | > 1e-6  AND | rhs_eqt(i) | > 1e-6
```

Otherwise, the denominator is 1.0 (as in ABSTOL mode).

To change local residue information, use the following function:

```
BOOLEAN setResidueInfo(INTEGER residue,
                       REAL    tolerance,
                       INTEGER tolType,
                       REAL    vref= 1.0)
```

where **residue** is the residue equation number, **tolerance** is the allowed tolerance in this residue, **tolType** is the tolerance type — either ABSTOL or FRACTOL, and vref is the reference value to be used when FRACTOL is selected (by default it is 1.0). The following is an example of usage:

```
setResidueInfo( 2, 1e-3, FRACTOL, 4568.5)
```

This example changes the second residue equation parameters as follows: the local tolerance is 1E-3, the residue will be evaluated in fractional mode and the reference value for the denominator is 4568.5. If, instead, it is written thus:

```
setResidueInfo( 2, 1e-3, FRACTOL)
```

EcosimPro will estimate the vref on every residues function call based on the criteria explained before.

Another function, called evalNormResidue(), is used to express a closure equation and enable the residue to be calculated in ABSTOL or FRACTOL mode without having to change its expression. The function signature is as follow:

```
REAL evalNormResidue(IN INTEGER index,
                     IN REAL lhs,
                     IN REAL rhs,
                     IN REAL vRef= 1.0);
```

where index is the residue number, lhs is the left-hand side of the equation, rhs is the right-hand side of the equation, and vRef is the reference value to be used when FRACTOL is selected. The latter is set by default to 1.0 and its use is not mandatory.

The modeller should use this function only for external closure equations (internals are managed automatically by EcosimPro). For example, a residue can be written as:

```
 res[3] = evalNormResidue(3, x+ 4*y, 5923.4, 4568.5)
```

Residue 3 (res[3]) calls this function to calculate the residue. If it uses the FRACTOL format, the function returns:

```
res[3] = ((x + 4*y) - 5923.4 ) / 4568.5
```

the last argument 4568.5 as a reference value for normalising the residue. If it uses the ABSTOL format the function returns:

```
res[3] = (x + 4*y) - 5923.
```

If the user provides a vRef value in both functions setResidueInfo() and evalNormResidues(), the latter has priority.

13.1.1.5. GLOBAL CONVERGENCE CRITERIA

In the previous section we saw how a residue is calculated. This section addresses the global convergence criteria of the equation system. Strictly speaking, RELTOL is only global convergence method, since the convergence check in ABSTOL and FRACTOL is carried out individually for each residue.

The global variable TOLTYPE is used to specify the type of global tolerance; for example:

```
TOLTYPE =RELTOL
```

The global variable TOLERANCE is used to indicate the global tolerance required by the algorithm. If the user assigns a local tolerance (see function setResidueInfo), it will always prevail unless the global convergence criterion RELTOL is used which prevails over the global tolerance.

Using the variable TOLERANCE is simple. For example:

```
   TOLERANCE = 1e-6
```

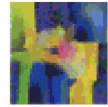
The global convergence criteria of the algorithm in use can be controlled. Global convergence comes in 3 types:

13.1.1.6. RELTOL

This is the relative convergence criterion which is always used in the Minpack routine and the default one in NR. It is not the original criterion of the Minpack package, but rather a variant.

If this (the default) method is selected, the individual residues will be calculated in absolute (ABSTOL) unless any have been individually changed to FRACTOL using the function setResidueInfo(), in which case it is calculated in fractional mode.

Convergence test:



If FRES is the residues vector, NXSOL is the normalised solutions vector, and ENORM(Z) denotes the Euclidean norm of a vector Z, then this convergence test attempts to guarantee that:

```
   ENORM(FRES) < TOLERANCE   AND  ENORM(FRES) <
TOLERANCE*ENORM(NXSOL)
```

If these conditions are satisfied, the algorithm succeeds.

If TOLERANCE is less than machine precision, then the method only attempts to satisfy the maximum tolerance that the latter allows. Further progress is not usually possible.

13.1.1.7. ABSTOL

When this method is selected, all the local convergence criteria are calculated in absolute mode (ABSTOL) unless explicitly defined for a FRACTOL residue (using the function setResidueInfo()), in which case that established by the function prevails.

The tolerance to be applied can be either the value of the TOLERANCE variable or the local tolerance set in the function setResidueInfo(). If no specific tolerance is assigned, the global variable TOLERANCE is taken.

Convergence test:

If FRES is the residues vector, N is the number of residues and TOL is the tolerances vector, then this convergence test attempts to guarantee that:

```
     success= TRUE
     FOR( i IN 1,N)
  IF ( FRES(i) > TOL(i) )
     success= FALSE
END FOR
```

If any local residue fails, then the global convergence fails; otherwise, the algorithm succeeds.

13.1.1.8. FRACTOL

When this method is selected, all the local convergence criteria are calculated in fractional mode (FRACTOL) unless explicitly defined for an ABSTOL residue (using the function setResidueInfo()), in which case that established by the function prevails.

The tolerance to be applied can be either the value of the TOLERANCE variable or the local tolerance set in the function setResidueInfo().If no specific tolerance is assigned, the global variable TOLERANCE is taken.

Convergence test:

If FRES is the residues vector and N is the number of residues, then this convergence test attempts to guarantee that

```
success= TRUE
FOR( i IN 1,N)
  IF ( FRES(i) > TOLERANCE )
     success= FALSE
END FOR
```

If any local residue fails, then the global convergence fails; otherwise, the algorithm succeeds.

13.2. Changing Parameters for the Steady Solver

EL allows certain default values to be changed when steady states are calculated. The table below shows the parameters that can be modified along with their default values and the associated EL variable or function which allows them to be modified:

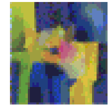
Parameter	EL variable	Values	Default value
Solver method	SMETHOD	NR, MINPACK	NR
Tolerance type	TOLTYPE	RELTOL, ABSTOL, FRACTOL	RELTOL
Tolerance error	TOLERANCE	> 0	1.0E-6
Maximum number of iterations	MAXITER	> 0	10000
Maximum number of residue function evaluations	MAXFEVAL	> 0	40000
Maximum number of Jacobian evaluations	MAXJACEVAL	> 0	60
Maximum number of consecutive Jacobian estimations	MAXBROYDEN	> 0	60

A steady state global tolerance error can be changed with the following statements:

```
TOLTYPE = ABSTOL
TOLERANCE = 1.0e-3
```

The maximum number of iterations carried out by the steady solver can be changed with the following:

```
MAXITER = 15
```



And the maximum number of residue function calls can be changed with the following:

```
MAXFEVAL = 200
```

It is also possible to independently modify some parameters per variable or residue function. Parameters that can be modified are shown below with their default values and associated function.

Parameter	EL function	Default value
Unknown variable minimum absolute perturbation for Jacobian calculation	setAlgVariableInfo()	1.5E-8
Unknown variable minimum relative perturbation for Jacobian calculation	setAlgVariableInfo()	1.5E-8
Unknown variable maximum absolute step for each iteration	setAlgVariableInfo()	1.0E+18
Unknown variable maximum relative step for each iteration	setAlgVariableInfo()	1.0E+18
Residue tolerance type	setResidueInfo()	ABSTOL
Residue tolerance error	setResidueInfo()	1.0E-6
Residue reference value to be used as denominator for producing a fractional residue (used only in FRACTOL local tolerance)	setResidueInfo()	1.0

The syntax for the algebraic parameter modification function is:

```
BOOLEAN setAlgVariableInfo(STRING name,
                          REAL dxAbs,
                                REAL dxRel,
                                REAL maxStepAbs,
                                REAL maxStepRel)
```

where **name** is the name of the unknown variable, **dxAbs** and **dxRel** are the minimum perturbations (absolute and relative) for Jacobian calculation, and **maxStepAbs** and **maxStepRel** are the maximum allowed steps (absolute and relative) for each iteration. If a default value is not to be modified then -1 should be used. For example:

```
setAlgVariableInfo("hpc.N", 1.0e-3, -1, 1.0e-1, -1)
```

This call would change the minimum absolute perturbation for the Jacobian evaluation default value and the maximum absolute step for the “hpc.N” variable of the model. The rest will be left at their default values.

And the syntax for the residue parameter modification function is:

```
BOOLEAN setResidueInfo(INTEGER residue,
                       REAL    localTolerance,
```

```
INTEGER tolType,
DOUBLE  vref = 1.0)
```

where **residue** is the residue equation number, **tolerance** is the allowed tolerance error, **tolType** is the tolerance type, and vref is the term to be used as denominator to calculate the fractional residue (used only in FRACTOL mode), relative or absolute. For example:

```
setResidueInfo( 2, 1.0e-3, ABSTOL)
setResidueInfo( 4, 1.0e-5, FRACTOL, 1452.3)
```

This example would change the second residue equation parameters as follows: the tolerance error type to absolute and its value to 1.0E-3. The fourth residue will use fractional tolerance, with a tolerance of 1E-5 and the term 1452.3 to be used to estimate a fractional residue (see previous section).

13.3. Freezing Variables

13.3.1.1. FREEZE(variable)

This function freezes any dynamic variable while calculating a steady state. During the steady calculation this dynamic variable will not make the derivative zero, it will be ignored by the Newton-Raphson solver. For instance before calling the steady function you can add two freeze statements:

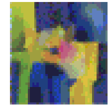
```
FREEZE( pump.f_out.h)
FREEZE( pump.f_in.h )
STEADY()
```

At the end of the calculation the variables pump.f_out.h' and pump.f_in.h' can have any value (in the regular case they should be 0).

13.3.1.2. UNFREEZE(variable)

This function unfreezes any frozen dynamic variable and allows it to be used in the calculation of a steady state. It is the opposite of the previous function. For instance

```
FREEZE( pump.f_out.h)
FREEZE( pump.f_in.h )
STEADY()
          -- returns freeze variables to original state
UNFREEZE( pump.f_out.h)
UNFREEZE( pump.f_in.h )
```



13.3.1.3. *ISFROZEN(variable)*

This function returns TRUE if the variable is already frozen, otherwise it returns FALSE.

13.4. Steady Solver Debugging Information

When a steady state fails to find a solution and you use debug level 2, the steady solver generates an internal debugging report which is written in the log file (if you have debug level 3 it also writes the convergent steady states).

Normally, it only makes sense to request this report when a steady state is non-convergent, but it can be produced anyway. The debug report is located in the log file associated with each experiment execution.

Using debug level 1, the user can display some information about the success or lack of success of the method. Only a brief error message is printed. However, for debug levels 2 and 3, more information is given detailing the solver configuration and iteration process: variable and residue values, Jacobian matrixes, etc.

The following is an example for a component test:

```
COMPONENT test
   DECLS
      REAL x
      REAL y
   CONTINUOUS
      x' = 1 + y**2
      y' = x - y
END COMPONENT
```

A default partition and a simple experiment which attempts to calculate the steady state could be generated for it. The experiment should look like this:

```
EXPERIMENT exp1 ON test.default
 INIT
      -- State variables
      x = 0
      y = 0
 BODY
      setDebugLevel(3)
      IF (STEADY() == STEADY_NOK) THEN
         WRITE("Convergence not reached\n")
      ELSE
         WRITE("Convergence reached\n")
```

```
        END IF
END EXPERIMENT
```

When running this experiment the output produced is:

```
Experiment: exp1 Library: SCLIB_ENGINE_LIBRARY Component: test1 Partition: default
User: pce Date: 11/16/07 Time: 09:45:57
Library dependency list: DEFAULT_LIB V0.0
Programme & Version: ECOSIMPRO V1.4.0
TRACE: TRUE LOG: TRUE WARN: TRUE POST-PROCESS: TRUE
Begin of reading symbols table...
File: C:/EcosimPro/USER_LIBS\autocode\DEFAULT_LIB.test1.default.stab
Variables: 6
Equations: 2
End of reading symbols table
Changed DEBUG_LEVEL to "MAXIMUM(3)"
=====================================
Begin of steady-1 (Method: NR, TOLTYPE= RELTOL, TOLERANCE= 1e-006)
Begin of components INIT block execution...
End of components INIT block execution
*** NOTE level 0 (code 9:204:32:98) ***

Cannot find steady state

Reason: Convergence not reached: Iteration is no making good progress as measured by
        the   improvement from the last 20 iterations trials.

End of steady-1 (Status: NOK, Max-Error:1, Runtime:0s, Jacob.:2, Iter.:0, Residues:25)
=====================================
Convergence not reached
- END OF EXPERIMENT -
Total steady calls OK: 0 NOK: 1
Total processor time: 0.001 seconds
```

If we open the log file associated with the experiment, we obtain more information about the resolution:

(GLOBAL ITERATION)

Convergence not reached: Iteration is no making good progress as measured by the improvement from the last 20 iterations trials.
Tracing:

Number of iterations	0
Number of residue evaluations	25
Number of Jacobian evaluations	2
Number of Broyden estimations	18

Solver parameters:
Global settings:

Global convergence criteria	RELTOL
Global tolerance	1E-006
Maximum number of iterations	10000
Maximum number of residue evaluations	600
Maximum number of Jacobian evaluations	60
Maximum number of consecutive Broyden updates	60

Residue equation parameters:

	Tolerance Type	Tolerance	Residue expression
Residue 1:	ABSTOL	1E-006	[1] x' = 1. + y ** 2.
Residue 2:	ABSTOL	1E-006	[2] y' = x - y

Unknown variable parameters:

	Initial guess	Absolute perturbation	Relative perturbation	Absolute maximum step	Relative maximum step
x	0	1.5E-008	1.5E-008	1E+018	1E+018
y	0	1.5E-008	1.5E-008	1E+018	1E+018

Iteration process results:
Unknowns:

Iteration	x	y
0	0	0

Residues:

Iteration	Residue 1	Residue 2	Global Error	Convergence Ratio
0	-1	0	1	1

Convergence process:

Iteration	Residue1	Residue2	Global convergence
0	NO	YES	NO

```
*** NOTE level 0 (code 9:204:32:98) ***
Cannot find steady state Reason: Convergence not reached: Iteration is not making good progress
as measured by the improvement from the last 20 iteration trials.
End of steady-1 (Status: NOK, Max-Error:1, Runtime:0s, Jacob.:2, Iter.:0, Residues:25)
=====================================
Convergence not reached
- END OF EXPERIMENT -
Total steady calls OK: 0 NOK: 1
Final statistics:
```

CALCULATION	STATUS	ESI CODE	MAX-ERROR	RUNTIME(s)	ITERATIONS	JACOBIANS	RESIDUES	TRACE
steady-1	STEADY_NOK	9:204:32:98	1	0	0	2	25	debug

Total processor time: 0.001 seconds

This simple example shows how the first algebraic variable can not be calculated properly.

13.5. Steady State Calculation Example

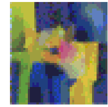
Examples of usage of some customisation functions for steady calculations are shown in this section.

The following experiment changes some parameters of the steady solver:

```
EXPERIMENT exp1 ON test.default
   INIT
      -- State variables (initial values)
      x = 0
      y = 0
   BODY
      -- Change the tolerance error
      TOLTYPE = RELTOL
      TOLERANCE= 1e-8

      -- Change the maximum number of iterations
      MAXITER = 80

      -- Change the maximum numbers of residues function calls
      MAXFEVAL = 200

      -- Change the parameters related to variable "x"
      setAlgVariableInfo("x", 0.005, 0.005, 0.1, 0.1)

      -- Change the parameters related to second residue
      setResidueInfo( 2, 0.0001, FRACTOL,34.5)

      -- Calculates the steady state
      IF (STEADY() == STEADY_NOK) THEN
         WRITE("Convergence not reached\n")
      END IF

END EXPERIMENT
```

This example customizes the algebraic system by using ABSTOL as the global convergence criteria (all individual residues will use ABSTOL), changes the global tolerance to 1E-8, changes the maximum number of iterations to 80, and the maximum number of residue function calls to 200. For variable "x", set the minimum absolute and relative perturbation for Jacobian calculation to 0.005, and set the maximum absolute and relative step for each iteration to 0.1. In this case it changes residue 2 to use fractional tolerance with a local tolerance of 0.0001 and assigns a reference value to be used to calculate this value (see previous section) of 34.5.

Finally, it calls the STEADY() functions and checks whether it succeeds or not.



13.6. External Newton-Raphson Functions

The user can directly call the nonlinear equation solvers with the functions nlsolver() and nldsolver(). These routines code the same algorithms used in the tool (an improved Newton-Raphson numerical method for solving a set of algebraic equations). The user can directly call these functions to solve a set of algebraic equations. There are two functions:

```
NO_TYPE nlsolver (      FUNC_PTR fcn,
               IN INTEGER n,
               OUT REAL dyn[],
               OUT REAL fres[],
               OUT INTEGER info)

NO_TYPE nldsolver (       FUNC_PTR fcn,
                  IN INTEGER n,
                  OUT STRING names[],
                  OUT REAL dyn[],
                  OUT REAL fres[],
                  OUT INTEGER info,
                  IN STRING equations[],
                  IN STRING names[])
```

Both are identical; the only difference is that the second uses two extra arguments passing arrays of strings with the residue equations and names of the unknown variables (used for reporting more meaningful reports).

The arguments of the functions are:

- fcn: Function pointer. This function is called from the Newton-Raphson method (see below)
- n: Number of unknown variables
- dyn: Unknowns vector
- fres: Residues vector
- info: Returned code from the solver. It can take the following values:
  - info = 1 Ok, convergence succeeds
  - info = 2 Number of calls to fcn has reached or exceeded maxfev
  - info = 3 tol is too small. No further improvement in the approximate solution x is possible
  - info = 4 Iteration is not making good progress, as measured by the improvement from the last five Jacobian evaluations

- info = 5 Iteration is not making good progress, as measured by the improvement from the last ten iterations
- info = 6 A numerical exception happened during the calculation

- equations: Array with the residue equation strings
- names: Array with the unknown variable names

The fcn function must have the following signature:

```
FUNCTION NO_TYPE fcn(OUT INTEGER n,   -- number of algebraics
                 OUT REAL dyn[],  -- algebraics vector
                 OUT REAL fres[], -- residues vector
                 OUT INTEGER iflag) – debug flag
```

This function is used by the Newton-Raphson as an interface to evaluate the residues. It provides the expression of the residues or closure equations to be imposed. The following example shows a simple usage of this function:

Equations to solve:

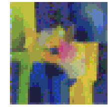
```
1.645 = x + y * 1e-6
2.3445e-6 * x + 3e-6 * y = 3.454
```

This can be expressed in a function using the following function:

```
  FUNCTION NO_TYPE fcn(INTEGER n,
                       REAL z[],
                       OUT REAL residues[],
                       INTEGER iflag)
  BODY
     residues[1]=  1.645 -  (z[1] + z[2]*1e-6 )
     residues[2]= (2.3445e-6*z[1] + 3e-6*z[2]) -  3.454
  END FUNCTION
```

It can also be expressed by using the evalNormResidue() function (only in case we want a residue in fractional mode):

```
  FUNCTION NO_TYPE fcn(INTEGER n,
                       REAL z[],
                       OUT REAL residues[],
                       INTEGER iflag)
  BODY
residues[1]=evalNormResidue(1, 1.645,    z[1] + z[2]*1e-6 )
residues[2]=evalNormResidue(2, 2.3445e-6*z[1] + 3e-6*z[2],
3.454)
  END FUNCTION
```



Both are equivalent; the only difference is that the second takes into account whether the residue is calculated using ABSTOL or FRACTOL tolerance.

Now we can solve this equation system from any experiment; we just need to:

- declare an array of unknown variables
- declare an array of residues
- declare an integer which gives the calculation status
- set the level of debugging wanted
- init the unknowns
- set the global tolerance criteria (or use the default one RELTOL)
- call the nlsolver() or nldsolver() functions
- check the status and print results

For example:

```
EXPERIMENT exp2 ON test.default
   DECLS
     REAL z[2]            -- unknowns array
     REAL residues[2]     -- residues array
     INTEGER info         -- calculation status
   BODY
     setDebugLevel(3)
          -- init the unknowns
     z[1] = 1e6
     z[2] = 0.0
          -- set the global tolerance criteria
     TOLTYPE= RELTOL    -- global tolerance criteria

          -- call the external newton-Raphson method
     nlsolver(fcn,2,z,residues,info)
     IF( info != 1 ) THEN
        WRITE("It does not converge\n")
     ELSE
        WRITE("It converges !!!\n")
        WRITE("SOLUTIONS: z[1]= %g,z[2]= %g ",z[1],z[2])
     END IF
END EXPERIMENT
```

When running this experiment the output is:

```
Changed DEBUG_LEVEL to "MAXIMUM(3)"
=====================================
```

```
Begin of external-steady-1 (Method: NR, TOLTYPE= RELTOL,
TOLERANCE= 1e-006)
End of external-steady-1 (Status: OK, Max-Error:5.39e-009,
Runtime:0s, Jacob.:1, Iter.:6, Residues:9)
=====================================
It converges !!!
SOLUTIONS: z[1]= 0.493668,z[2]= 1.15133e+006
```

The solver has found a solution for this problem using 6 iterations. We can deduce from the debug report that the residue 1 has very big values:

Residues:

Iteration	Residue 1	Residue 2	Global Error	Convergence Ratio
0	-999998	-1.1095	999998	1
1	-899998	-1.34395	899998	0.9
2	-699998	-1.81285	699998	0.777777
3	-299998	-2.75065	299998	0.42857
4	-0.741623	-1.22913	1.43554	4.78516E-006
5	-0.0576413	0.000564845	0.057644	0.0401549
6	2.01472E-008	-1.97429E-010	2.01482E-008	3.49528E-007

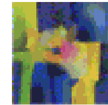
We can cause this residue to be calculated using FRACTOL local criteria with a reference value of 500000, and we can do it using the setResidueInfo() function to specify that this residue must be calculated in fractional mode with this reference value:

```
    -- set the global tolerance criteria
TOLTYPE= RELTOL   -- global tolerance criteria
setResidueInfo(1,1e-5,FRACTOL,500000) -- set local residue mode
```

We can run the experiment again. If we examine the log file, the residue value is lower but the iterations remain identical.

Residues:

Iteration	Residue 1	Residue 2	Global Error	Convergence Ratio
0	-2	-1.1095	2.28713	1
1	-1.9223	-0.924149	2.1329	0.932566



2	-1.71482	-0.654386	1.83544	0.860535
3	-1.15864	-0.443812	1.24074	0.675989
4	-0.0502092	-0.0124	0.0517178	0.0416832
5	1.2334E-009	-6.06228E-006	6.06228E-006	0.000117219
6	-1.09667E-012	5.39021E-009	5.39021E-009	0.00088914

We can use the other function nldsolver() to obtain more information in the debug reports. We have to pass two arrays of strings with the equations and the unknown names. For example:

```
STRING eqts[2]  = { "1.645 = x + y * 1e-6", \
                   "2.3445e-6 * x + 3e-6 * y = 3.454" }
STRING names[2] = { "x", "y"}
```

Now, instead of passing the new arguments, the nldsolver() function is used:

```
nldsolver(fcn,1,names,z,residues,info,eqts,names)
```

When reading the debug report, all equations and unknowns are referred to more meaningful names: "1.645 = x + y * 1E-6" instead of "Residue equation1" and "x" instead of "Var 1". For example:

Residue equation parameters:

	Tolerance Type	Tolerance	Residue expression
Residue 1:	FRACTOL	0.01	1.645 = x + y * 1E-6
Residue 2:	ABSTOL	1E-006	2.3445E-6 * x + 3E-6 * y = 3.454

Unknown variable parameters:

	Initial guess	Absolute perturbation	Relative perturbation	Absolute maximum step	Relative maximum step
x	1E+006	1.5E-008	1.5E-008	1E+018	1E+018
y	0	1.5E-008	1.5E-008	1E+018	1E+018

14. LIBRARY OF PREDEFINED CLASSES

14.1. TABLE CLASS

Apart from the EL type TABLE_1D, TABLE_2D and TABLE_3D, the modeller can use the TABLE class to represent any of the tables (in one, two or three dimensions).

Use of the TABLE class is similar to that of other class types in EL but it has some constraints:

- It cannot be defined as a datum (in the DATA block of the component)
- It cannot be initialised in the declaration
- To be used, it has to be read from an external file (either in XML or ASCII format)

However, it also has some advantages:

- It is not necessary to specify the dimension when declaring; this is determined when the table is read from an external file
- The TABLE objects has methods that makes easier to work with them (instead of individual funations). The user does not need to pass the table as an argument everytime he wants to interpolate the table.

An example of use of this class is the following:

```
COMPONENT foo
DECLS
          REAL x
OBJECTS
     TABLE table1
INIT
     table1.read("tables/table1d.xml",2)
CONTINUOUS
     x= table1.interpd1D(TIME)
END COMPONENT
```

This example declares an object named "table1" of TABLE. It reads the table from a file and it interpolates it with variable TIME as the argument. This code is the equivalent of writing:

```
COMPONENT foo
DECLS
            TABLE_1D table1
      REAL      x
OBJECTS
      TABLE table1
INIT
      readTable1D ("tables/table1d.xml",table1,2   )
CONTINUOUS
      x= linearInterp1D (table1,TIME)
END COMPONENT
```

The advantage of the first approach is that the object "table1" is more intelligent than the second case since it behaves like a real object with an interface.

The TABLE class description with all available methods is as follows:

```
EXTERN CLASS TABLE IS_A INTEG_topClass
   METHODS
      -- Read table from file
      EXTERN METHOD BOOLEAN read(IN STRING fileName,
                                 IN INTEGER format=2,
                                 IN INTEGER dim=1)

      -- Free memory from tables and clear object entities
      EXTERN METHOD BOOLEAN clear()

      -- Return dimension
      EXTERN METHOD INTEGER dimension()

      -- Printing functions
      EXTERN METHOD BOOLEAN print(IN INTEGER format)

      --Interpolation functions providing inter. and extrap. method
      EXTERN METHOD REAL interp1D(IN ENUM T_INTERP tin, IN ENUM T_INTERP tex,
                       IN REAL x)
      EXTERN METHOD REAL interp2D(IN ENUM T_INTERP tin, IN ENUM T_INTERP tex,
                       IN REAL x, IN REAL y)
      EXTERN METHOD REAL interp3D(IN ENUM T_INTERP tin,
                       IN ENUM T_INTERP tex, IN REAL x,IN REAL y,IN REAL z)

      -- Interpolate using the default methods
      EXTERN METHOD REAL interpd1D(IN REAL x)
      EXTERN METHOD REAL interpd2D(IN REAL x, IN REAL y)
      EXTERN METHOD REAL interpd3D(IN REAL x, IN REAL y, IN REAL z)

      -- Interpolation with history functions for tables
      EXTERN METHOD REAL interpHist1D(IN ENUM T_INTERP tin,
```

```
                 IN ENUM T_INTERP tex, IN REAL x, OUT INTEGER i)
EXTERN METHOD REAL interpHist2D(IN ENUM T_INTERP tin,
                 IN ENUM T_INTERP tex, IN REAL x,IN REAL y,
                 OUT INTEGER i,OUT INTEGER j)
EXTERN METHOD REAL interpHist3D(IN ENUM T_INTERP tin,
                IN ENUM T_INTERP tex, IN REAL x, IN REAL y,
                IN REAL z, OUT INTEGER i, OUT INTEGER j, OUT INTEGER k)

-- Default interpolation with history functions
EXTERN METHOD REAL interpHistd1D(IN REAL x, OUT INTEGER i)
EXTERN METHOD REAL interpHistd2D(IN REAL x,IN REAL y,OUT INTEGER i,
                       OUT INTEGER j)
EXTERN METHOD REAL interpHistd3D(IN REAL x,IN REAL y, IN REAL z,
                       OUT INTEGER i, OUT INTEGER j, OUT INTEGER k)

-- Inverse interpolation functions for tables
EXTERN METHOD REAL invInterp1D(IN ENUM T_INTERP tin,
            IN ENUM T_INTERP tex,  IN REAL val)
EXTERN METHOD REAL invInterpd1D(IN REAL val)

END CLASS INCLUDE "INTEG_simula.h" IN "INTEG.lib"
```

14.1.1. Table reading

The read() function is used to read tables. The signature is as follows:

```
NO_TYPE read(IN STRING filePath, IN INTEGER format=2)
```

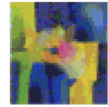
This method reads the table from a file either in XML (format=2) or ASCII (format=1). If the modeller does not pass this argument, EcosimPro sets format to 2. For example:

```
table1.read("table1.xml")
```

This call reads the file table1.xml in XML. The XML format is described in a latter appendix and the ASCII format is described in the chapter dedicated to family basic types TABLE_1D (2D or 3D).

To facilitate the use of file paths, it is recommended that FILEPATH variables be used to enable relative or absolute file paths to be entered when working in the object editor. For example:

```
DECLS
     FILEPATH pathTable ="@MECHANICAL@/tables/table34.xml"
OBJECTS
          TABLE table1
INIT
     table1.read( pathTable)
```



14.1.2. Reset table

The method clear() method is used to clear the current table from memory. Its signature is:

```
BOOLEAN clear()
```

It returns FALSE if any problem is found during the resetting.

14.1.3. Obtaining the dimensions

There is a method to obtain the dimensions of a table. Its signature is:

```
INTEGER dimension()
```

It returns either 1 or 2 or 3.

14.1.4. Interpolation in TABLE class

There are a number of methods available for interpolation in tables, offering very powerful use of the tables.

14.1.4.1. Interpolation methods

The TABLE class allows interpolation and extrapolation with a number of predefined methods. There is an enumeration in EL with all the identifiers of the methods, which are as follows:

```
ENUM T_INTERP = {DEFAULT,
                 FORBIDDEN,
                 CONSTANT,
                 LINEAR,
                 QUADRATIC,
                 CUBIC,
                 SPLINE}
```

The meaning of each one is as follows:

- **DEFAULT:** Takes the default interpolation/extrapolation from the .xml file
- **FORBIDDEN:** Forbidden interpolation or extrapolation. If tried, an error message is displayed

- **CONSTANT:** Constant values are used for each interval
- **LINEAR:** Linear interpolation between two points
- **QUADRATIC:** Interpolation by 2nd order polynomials between 3 points
- **CUBIC:** Cubic interpolation by 3rd order polynomials between 4 points
- **SPLINE:** Cubic interpolation by 3rd order polynomials between 2 points, maintaining continuity of first and second derivative between points.

Each table has a default interpolation method and a set of valid interpolation/extrapolation methods. If the user tries to use a method that is not permitted, an error message will be displayed. If the user wants to use the default method, the DEFAULT indicator must be used. For instance the following can be written in XML:

```
<interp default="SPLINE" valid="{CONSTANT, LINEAR, SPLINE}" />
<extrap default="LINEAR" valid="{FORBIDDEN, CONSTANT, LINEAR, SPLINE}" />
```

These two lines say that the default method for interpolation is SPLINE and the valid methods are CONSTANT, LINEAR and SPLINE. For extrapolation, the default method is LINEAR and the valid methods are FORBIDDEN, CONSTANT, LINEAR and SPLINE.

14.1.4.2. Default interpolation methods

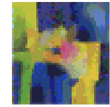
There are a number of methods for 1-, 2- or 3-dimensional tables where only the name of the table and the input values need to be passed.The interpolation and extrapolation methods are the default ones read from the .xml file:

```
REAL interpd1D(IN REAL x) -- input value x
REAL interpd2D(IN REAL x, -- input value x
         IN REAL y) -- input value y
REAL interpd3D(IN REAL x, -- input value x
         IN REAL y, -- input value y
         IN REAL z) -- input value z
```

14.1.4.3. Explicit interpolation methods

There are a number of methods for 1-, 2- or 3-dimensional tables which require that the interpolation and extrapolation methods be passed. Only numerical methods from a valid list read from the .xml file can be used:

```
REAL interp1D(
     IN ENUM T_INTERP tin, -- interpolation method
     IN ENUM T_INTERP tex, -- extrapolation method
     IN REAL x) -- input value x
```



```
REAL interp2D(
      IN ENUM T_INTERP tin, -- interpolation method
      IN ENUM T_INTERP tex, -- extrapolation method
      IN REAL x, -- input value x
      IN REAL y) -- input value y
REAL interp3D(
      IN ENUM T_INTERP tin, -- interpolation method
      IN ENUM T_INTERP tex, -- extrapolation method
      IN REAL x, -- input value x
      IN REAL y, -- input value y
      IN REAL z) -- input value z
```

14.1.4.4. Default interpolation methods with history

There are a number of methods for 1-, 2- or 3-dimensional tables with default interpolation and extrapolation methods and with one or more extra arguments which allow the value of the previous position in the table to be passed and, therefore, to be more efficient in the search for the corrective interval.

In very large tables it can be useful to begin the task of searching at the time it was last tried. As a table may be used for several interpolations simultaneously, you can send additional position marker arguments. In this way, if a table is being interpolated in two well-separated areas, you can perform the task more efficiently by using two sets of position variables. The format of these functions is similar to that described above, but there are extra INTEGER arguments which are used as a memory flag to start next time from a certain position. The value is changed internally by the interpolation functions and the change is visible outside (they are OUT variables). The interpolation function should have as many integer flags as dimensions.

```
REAL interpdHist1D(
    IN REAL x,        -- input value x
    OUT INTEGER i)  -- previous position for x
REAL interpdHist2D(
    IN REAL x,      -- input value x
    IN REAL y,      -- input value y
    OUT INTEGER i, -- previous position for x
    OUT INTEGER j) -- previous position for y
REAL interpdHist3D(
    IN REAL x,      -- input value x
    IN REAL y,      -- input value y
    IN REAL z,      -- input value z
    OUT INTEGER i, -- previous position for x
    OUT INTEGER j, -- previous position for y
    OUT INTEGER k) -- previous position for z
```

For example:

```
DECLS
     INTEGER lastx = 0.
OBJECTS
     TABLE table1
CONTINUOUS
     f_out.fh = table1.interpdHist1D(TIME, lastx)
```

The lastx index is updated with each call, so that in the next call the search in the table is started at the last lastx index.

14.1.4.5. Explicit interpolation methods with history

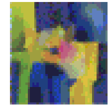
There are a number of methods for 1-, 2- or 3-dimensional tables with passing of historical values which allow explicit definition of the interpolation and extrapolation methods:

```
REAL interpHist1D(
    IN ENUM T_INTERP tin, -- interpolation method
    IN ENUM T_INTERP tex, -- extrapolation method
    IN REAL x,      -- input value x
    OUT INTEGER i) -- previous position for x
REAL interpHist2D(
    IN ENUM T_INTERP tin, -- interpolation method
    IN ENUM T_INTERP tex, -- extrapolation method
    IN REAL x,     -- input value x
    IN REAL y,     -- input value y
    OUT INTEGER i, -- previous position for x
    OUT INTEGER j) -- previous position for y
REAL interpHist3D(
    IN ENUM T_INTERP tin, -- interpolation method
    IN ENUM T_INTERP tex, -- extrapolation method
    IN REAL x,     -- input value x
    IN REAL y,     -- input value y
    IN REAL z,     -- input value z
    OUT INTEGER i, -- previous position for x
    OUT INTEGER j, -- previous position for y
    OUT INTEGER k) -- previous position for z
```

14.1.4.6. Inverse interpolation methods in 1D tables

There are a number of methods for inverse interpolation in 1D tables. There are two versions: one passes the interpolation and extrapolation methods, and the other uses the default methods:

```
REAL invInterp1D(
    IN ENUM T_INTERP tin, -- interpolation method
```



```
        IN ENUM T_INTERP tex, -- extrapolation method
        IN REAL val)  -- output value, now it is the input

   REAL invInterpd1D(
        IN REAL val) -- output value, now it is the input
```

14.1.5. Print

The print() method is used to print a table. The signature of this method is as follows:

```
BOOLEAN print(IN INTEGER format)
```

At the moment, the format must be equal to 1. In future more formats for printing tables will be created.

14.2. SYSTEM_IOREADER CLASS

ECOSIMPRO has a SYSTEM_IOREADER class in the system that facilitates the use external and general information written in XML format (see the XML format for SYSTEM_IOREADER files) during the execution of experiments. For example:

```
EXPERIMENT exp1 ON test_io.default
   DECLS
        STRING var_exp_string
   OBJECTS
        SYSTEM_IOREADER io
   BODY
        //read the external file
      io.read("file.io.xml")

      //get the value associated to a variable entry
      datum_real = io.getVariableReal("area","","m2")
      WRITE("value of datum_real: %f\n",var_real)

      //get the value associated to a report entry
        var_exp_string = io.getReportFilename("report1")
        WRITE("value of var_exp_string: %s\n",var_exp_string)

      //get the value associated to a generic entry
      var_exp_string = io.getGenericAsString("nsteps")
      WRITE("value of var_exp_string: %s\n",var_exp_string)

      //calculate
      STEADY()
END EXPERIMENT
```

From this point on, the user can use the "io" object like any other object in ECOSIMPRO. The definition of the CLASS with all the available methods in EL is:

```
EXTERN CLASS SYSTEM_IOREADER IS_A INTEG_topClass
   METHODS
```

```
    -- Read io information from file
    EXTERN METHOD BOOLEAN read(IN STRING fileName)

    -- Gets values variables values
    EXTERN METHOD BOOLEAN getVariableBoolean(
            IN STRING varName,IN STRING varPosition)
    EXTERN METHOD INTEGER getVariableInteger(
     IN STRING varName,IN STRING varPosition,IN STRING units)
    EXTERN METHOD REAL    getVariableReal(
     IN STRING varName,IN STRING varPosition,IN STRING units)
    EXTERN METHOD STRING  getReportFormat(IN STRING id)
    EXTERN METHOD STRING  getReportFilename(IN STRING id)
    EXTERN METHOD STRING  getReportVariables(IN STRING id)
    EXTERN METHOD STRING  getGenericAsString(IN STRING id)

END CLASS INCLUDE "INTEG_ioreader.h" IN "kerint.lib"
```

There are a number of functions that can be used to extract the value of the constants that have been inserted in the SYSTEM_IOREADER file.

14.2.1. SYSTEM_IOREADER reading

The read() function is used to read input files, as follows:

```
BOOLEAN read(IN STRING fileName)
```

This method reads the information from a file in .xml format. For example:

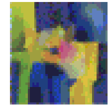
```
io.read("file.io.xml")
```

14.2.2. Functions for extracting variables

This group of functions allows the value of REAL, INTEGER and BOOLEAN variables to be obtained. The functions are:

```
EXTERN METHOD BOOLEAN getVariableBoolean(IN STRING varName,IN STRING varPosition)
EXTERN METHOD INTEGER getVariableInteger(IN STRING varName,IN STRING varPosition,IN STRING units)
EXTERN METHOD REAL    getVariableReal(IN STRING varName,IN STRING varPosition,IN STRING units)
```

where varName is the name of the variable to be extracted and varPosition is an identifier used for grouping variables. In the case of REAL and INTEGER variables the parameter units allow checking whether the unit specified corresponds to the unit in the file. If the variable is not in the file or is not of the expected type, an error message will be displayed.



The use is simple. For example:

```
valueX= io.getVariableReal("X","mygroup","m2")
```

It returns the value of variable "X" extracted from the file.

14.2.3. Functions for extracting report related information

This group of functions allows the value of report-related information to be obtained. A report entry in the file is of id-format-filename-set of variables type. The functions are:

```
EXTERN METHOD STRING  getReportFormat(IN STRING id)
EXTERN METHOD STRING  getReportFilename(IN STRING id)
EXTERN METHOD STRING  getReportVariables(IN STRING id)
```

where id is the identifier for a given report. If it is not in the file or is not of the expected type, an error message will be displayed.

The use is simple. For example:

```
var_exp_string = io.getReportFilename("report1")
```

14.2.4. Functions for extracting generic information

This group of functions allows the value of generic entries in the file to be obtained. These entries are of the identifier-value type. The functions are:

```
EXTERN METHOD STRING  getGenericAsString(IN STRING id)
```

where id is the identifier for a given generic element. If it is not in the file or is not of the expected type, an error message will be displayed.

The use is simple. For example:

```
var_exp_string = io.getGenericAsString("nsteps")
```

15. Appendix A: Regular Expressions

A regular expression or pattern is a text string that describes some (mathematical) sets of strings. A pattern R "matches" a string S if S is in the set of strings described by R.

The main usage of regular expressions in EL is for the selection of variables in reports and plots. The user can introduce a regular expression to refer to a set of variables. For instance a regular expression to match all symbols ending with "voltage" would be "*voltage".

The metacharacters used are:

- Match-zero-or-more characters: *
- Match one character: ?

The following table shows several typical examples of use:

Regular Expression	It will match ...
m	Any name containing the letter m
mus	Any name containing the word “mus”
m*	Any name beginning with m
mus*	Any name beginning with “mus”
*m	Any name ending with m
m*v	Any name starting with m and ending with v
m*g*v	Any name starting with m, ending with v and with a letter g at least in the middle
??m?	Any word with 4 letters and the third is an m
*.var	Any word ending with “.var”
*1.?	Any word ending with “.1” and any letter
??.?	Any word with 4 letters and the third is a dot
Fr[*	Any word starting with “Fr[“

16. Appendix B: The SYSTEM Libraries

The internal functions, global variables, enumerative types and classes defined in the system library are defined below:

```
--------------------------------------------------------------------------------
-- Interpolation and extrapolation methods
--------------------------------------------------------------------------------
ENUM T_INTERP = {DEFAULT, FORBIDDEN, CONSTANT, LINEAR, QUADRATIC, CUBIC, SPLINE}


-- Global variables
REAL TIME                           "Current time"
REAL TSTOP                          "Final simulation time"

BOOLEAN IN_STEADY                   "Steady calculation status flag (TRUE/FALSE)"
BOOLEAN IN_TRANSIENT                "Transient calculation status flag (TRUE/FALSE)"


-- Functions
--------------------------------------------------------------------------------
-- Mathematical functions
--------------------------------------------------------------------------------
"C" FUNCTION REAL sin(REAL x)
"C" FUNCTION REAL cos(REAL x)
"C" FUNCTION REAL tan(REAL x)
"C" FUNCTION REAL asin(REAL x)
"C" FUNCTION REAL acos(REAL x)
"C" FUNCTION REAL atan(REAL x)
"C" FUNCTION REAL atan2(REAL x, REAL y)
"C" FUNCTION REAL sinh(REAL x)
"C" FUNCTION REAL cosh(REAL x)
"C" FUNCTION REAL tanh(REAL x)
"C" FUNCTION REAL sqrt(REAL x)
"C" FUNCTION REAL exp(REAL x)
"C" FUNCTION REAL log(REAL x)
"C" FUNCTION REAL log10(REAL x)
-- Returns the absolute value of x
"C++" FUNCTION REAL abs(REAL x)
-- Returns the sqrt(x) if x positive, if not returns the -sqrt(-x)
"C++" FUNCTION REAL ssqrt(REAL x)
-- Special power function, if x positive returns x**x, if not returns -x*x.
"C++" FUNCTION REAL spow2(REAL x)
-- Special functions for fluids
"C++" FUNCTION REAL fsqrt(REAL x, REAL xlam)
"C++" FUNCTION REAL fpow2(REAL sqrtx, REAL xlam)
"C++" FUNCTION REAL dfsqrt(REAL x, REAL xlam)
"C++" FUNCTION REAL dfpow2(REAL sqrtx, REAL xlam)

--------------------------------------------------------------------------------
-- Returns the value of a variable with a certain delay (only valid from components)
--------------------------------------------------------------------------------
"C++" FUNCTION REAL delay(REAL var, REAL time)

--------------------------------------------------------------------------------
-- General functions
--------------------------------------------------------------------------------
"C++" FUNCTION NO_TYPE PRINT(STRING str)
```

```
"C++" FUNCTION NO_TYPE PRINTF(STRING file, STRING str)
"C++" FUNCTION ENUM GeneralEnumType setofElem(SET_OF(GeneralEnumType) setof, INTEGER i)
"C++" FUNCTION INTEGER setofPos(SET_OF(GeneralEnumType) setof, ENUM GeneralEnumType elem)
"C++" FUNCTION INTEGER setofSize(SET_OF(GeneralEnumType) setof)

--------------------------------------------------------------------------------
-- Functions returning variables names and values for the WRITE() function
--------------------------------------------------------------------------------
"C++" FUNCTION STRING gnameReal(REAL x)
"C++" FUNCTION STRING gnameInt(INTEGER x)
"C++" FUNCTION STRING gnameBool(BOOLEAN x)
"C++" FUNCTION STRING gnameString(STRING x)
"C++" FUNCTION STRING gnameEnum(ENUM GeneralEnumType x)
"C++" FUNCTION STRING gvalBool(BOOLEAN x)
"C++" FUNCTION STRING gvalEnum(ENUM GeneralEnumType x)
"C++" FUNCTION STRING gvalEnum2(STRING enumType, ENUM GeneralEnumType x)

--------------------------------------------------------------------------------
-- Functions working with arrays
--------------------------------------------------------------------------------
"C++" FUNCTION INTEGER sizeArrayName(STRING name, INTEGER code)
"C++" FUNCTION INTEGER dimArrayName(STRING name)
"C++" FUNCTION INTEGER dimArrayReal(OUT REAL v)
"C++" FUNCTION INTEGER sizeArrayReal(OUT REAL v, INTEGER code)
"C++" FUNCTION INTEGER dimArrayInt(OUT INTEGER v)
"C++" FUNCTION INTEGER sizeArrayInt(OUT INTEGER v, INTEGER code)
"C++" FUNCTION INTEGER dimArrayBool(OUT BOOLEAN v)
"C++" FUNCTION INTEGER sizeArrayBool(OUT BOOLEAN v, INTEGER code)
"C++" FUNCTION INTEGER dimArrayString(OUT STRING v)
"C++" FUNCTION INTEGER sizeArrayString(OUT STRING v, INTEGER code)

--------------------------------------------------------------------------------
-- Reading table
--------------------------------------------------------------------------------
"C++" FUNCTION INTEGER readTable1D(STRING filename, TABLE_1D t, INTEGER format)
"C++" FUNCTION INTEGER readTable2D(STRING filename, TABLE_2D t, INTEGER format)
"C++" FUNCTION INTEGER readTable3D(STRING filename, TABLE_3D t, INTEGER format)
"C++" FUNCTION INTEGER printTable1D(TABLE_1D t, INTEGER format)
"C++" FUNCTION INTEGER printTable2D(TABLE_2D t, INTEGER format)
"C++" FUNCTION INTEGER printTable3D(TABLE_3D t, INTEGER format)
"C++" FUNCTION INTEGER clearTable1D(TABLE_1D t)
"C++" FUNCTION INTEGER clearTable2D(TABLE_2D t)
"C++" FUNCTION INTEGER clearTable3D(TABLE_3D t)
"C++" FUNCTION INTEGER readTableCols1D(STRING filename,INTEGER c1,INTEGER c2,TABLE_1D t)

--------------------------------------------------------------------------------
-- Interpolation in tables
--------------------------------------------------------------------------------
"C++" FUNCTION BOOLEAN cellCrossing1D(TABLE_1D tbl, REAL x, INTEGER pi, OUT INTEGER i)
"C++" FUNCTION BOOLEAN cellCrossing2D(TABLE_2D tbl, REAL x, REAL y, INTEGER pi, OUT INTEGER i,
INTEGER pj, OUT INTEGER j)
"C++" FUNCTION BOOLEAN cellCrossing3D(TABLE_3D tbl, REAL x, REAL y, REAL z, INTEGER pi, OUT
INTEGER i, INTEGER pj, OUT INTEGER j, INTEGER pk, OUT INTEGER k)
"C++" FUNCTION REAL    cellLinearInterp1D(TABLE_1D tbl, REAL x, INTEGER i)
"C++" FUNCTION REAL    cellLinearInterp2D(TABLE_2D tbl, REAL x, REAL y, IN INTEGER i, INTEGER j)
"C++" FUNCTION REAL    cellLinearInterp3D(TABLE_3D tbl, REAL x, REAL y, REAL z, INTEGER i,
INTEGER j, INTEGER k)
"C++" FUNCTION REAL    linearInterp1D(TABLE_1D table, REAL val)
"C++" FUNCTION REAL    linearInterp2D(TABLE_2D table, REAL vx, REAL vy)
"C++" FUNCTION REAL    linearInterp3D(TABLE_3D table, REAL vx, REAL vy, REAL vz)
"C++" FUNCTION REAL    linearInterpHist1D(TABLE_1D tbl, REAL x, OUT INTEGER pi)
"C++" FUNCTION REAL    linearInterpHist2D(TABLE_2D tbl, REAL x, REAL y, OUT INTEGER pi, OUT
INTEGER pj)
"C++" FUNCTION REAL    linearInterpHist3D(TABLE_3D tbl, REAL x, REAL y, REAL z, OUT INTEGER pi,
OUT INTEGER pj, OUT INTEGER pk)
"C++" FUNCTION REAL    splineInterp1D(TABLE_1D table, REAL x)
"C++" FUNCTION REAL    splineInterp2D(TABLE_2D table, REAL x, REAL y)
"C++" FUNCTION REAL    invSplineInterp1D(TABLE_1D table, REAL val)

--------------------------------------------------------------------------------
```

```
-- Periodic functions
--------------------------------------------------------------------------------
"C++" FUNCTION REAL ramp(REAL TIME, REAL period, INTEGER waveInd)
"C++" FUNCTION REAL square(REAL TIME, REAL period, INTEGER waveInd)
"C++" FUNCTION REAL step(REAL TIME, REAL timeStep, INTEGER waveInd)
"C++" FUNCTION REAL pulse(REAL TIME, REAL period, REAL width, INTEGER waveInd)
"C++" FUNCTION REAL timeTableInterp(REAL TIME, TABLE_1D table, INTEGER waveInd)
"C++" FUNCTION REAL timeTableStep(REAL TIME, TABLE_1D table, INTEGER waveInd)
"C++" FUNCTION REAL periodTimeTableInterp(REAL TIME, TABLE_1D table, REAL period, INTEGER
waveInd)
"C++" FUNCTION REAL periodTimeTableStep(REAL TIME, TABLE_1D table, REAL period, INTEGER waveInd)

--------------------------------------------------------------------------------
-- Accessing variables with functions
--------------------------------------------------------------------------------
"C++" FUNCTION INTEGER getValueInt   (IN STRING name)
"C++" FUNCTION STRING  getValueEnum  (IN STRING name)
"C++" FUNCTION BOOLEAN getValueBool  (IN STRING name)
"C++" FUNCTION STRING  getValueString(IN STRING name)
"C++" FUNCTION REAL    getValueReal  (IN STRING name)
"C++" FUNCTION STRING  getValueAsString(IN STRING name)
"C++" FUNCTION BOOLEAN setValueInt   (IN STRING name, IN INTEGER v)
"C++" FUNCTION BOOLEAN setValueEnum  (IN STRING name, IN STRING v)
"C++" FUNCTION BOOLEAN setValueBool  (IN STRING name, IN BOOLEAN v)
"C++" FUNCTION BOOLEAN setValueString(IN STRING name, IN STRING v)
"C++" FUNCTION BOOLEAN setValueReal  (IN STRING name, IN REAL v)

-------------------------------------------------------------
-- FUNCTIONS TO ACCESS THE VARIABLES INFO IN THE MODEL
-------------------------------------------------------------
"C++" FUNCTION INTEGER getNumberVars ()
"C++" FUNCTION BOOLEAN existsVariable(IN STRING name)
"C++" FUNCTION STRING  getVarName (IN INTEGER v)
"C++" FUNCTION INTEGER getVarType (IN STRING name)
"C++" FUNCTION STRING  getVarTypeStr (IN STRING name)
"C++" FUNCTION INTEGER getVarCategory (IN STRING name)
"C++" FUNCTION STRING  getVarCategoryStr (IN STRING name)

-------------------------------------------------------------
-- FUNCTIONS TO GET/SET 1D, 2D and 3D ARRAYS
-------------------------------------------------------------
"C++" FUNCTION BOOLEAN setArray1D(IN STRING name, IN REAL v[],IN INTEGER dim1)
"C++" FUNCTION BOOLEAN setArray2D(IN STRING name, IN REAL v[],IN INTEGER dim1,IN INTEGER dim2)
"C++" FUNCTION BOOLEAN setArray3D(IN STRING name, IN REAL v[],IN INTEGER dim1,IN INTEGER dim2,IN
INTEGER dim3)
"C++" FUNCTION BOOLEAN getArray1D(IN STRING name, OUT REAL v[],IN INTEGER dim1)
"C++" FUNCTION BOOLEAN getArray2D(IN STRING name, OUT REAL v[],IN INTEGER dim1,IN INTEGER dim2)
"C++" FUNCTION BOOLEAN getArray3D(IN STRING name, OUT REAL v[],IN INTEGER dim1,IN INTEGER dim2,IN
INTEGER dim3)

-------------------------------------------------------------
-- OBTAIN RESIDUES FROM THE MATH MODEL
-------------------------------------------------------------
"C++" FUNCTION REAL getResidueValue(IN INTEGER index, IN INTEGER box= -1)

-------------------------------------------------------------
-- WORKING WITH FILE PATHS
-------------------------------------------------------------
"C++" FUNCTION STRING  expandFilePath(IN STRING filePath)
"C++" FUNCTION STRING  checkFileExists(IN STRING filePath)
"C++" FUNCTION BOOLEAN existsFile(IN STRING filePath)

-------------------------------------------------------------
-- OBTAIN THE UNITS OF A VARIABLE
-------------------------------------------------------------
"C++" FUNCTION STRING unitsOfVariable(IN STRING variableName)

--------------------------------------------------------------------------------
-- OBTAIN A HEADER WITH INFO ABOUT THE EXPERIMENT(name,date,library versions,etc.
--------------------------------------------------------------------------------
"C++" FUNCTION STRING versionReport()
```

```
-- Classes
--------------------------------------------------------------------------------
-- Top level class
--------------------------------------------------------------------------------
EXTERN CLASS INTEG_topClass
   DECLS
   METHODS
END CLASS INCLUDE "INTEG_simula.h" IN "INTEG.lib"

--------------------------------------------------------------------------------
-- Define main simulation class
--------------------------------------------------------------------------------
EXTERN CLASS INTEG_simula IS_A INTEG_topClass
   DECLS
      REAL TIME
      REAL TSTOP
   METHODS
      EXTERN METHOD INTEGER getNumberVars()
END CLASS INCLUDE "INTEG_simula.h" IN "INTEG.lib"

----------------------------------------------
-- WRAPPER TO TABLE CLASS (INTEG_table in C++)
----------------------------------------------
EXTERN CLASS TABLE IS_A INTEG_topClass
   METHODS
      -- Read table from file
      EXTERN METHOD BOOLEAN read(IN STRING fileName,IN INTEGER format=2,IN INTEGER dim=1)

      -- Free memory from tables and clear object entities
      EXTERN METHOD BOOLEAN clear()

      -- Return dimension
      EXTERN METHOD INTEGER dimension()

      -- Printing functions
      EXTERN METHOD BOOLEAN print(IN INTEGER format)

      -- Interpolation functions for tables giving the interpolation methods
      EXTERN METHOD REAL interp1D(IN ENUM T_INTERP tin, IN ENUM T_INTERP tex, IN REAL x)
      EXTERN METHOD REAL interp2D(IN ENUM T_INTERP tin, IN ENUM T_INTERP tex, IN REAL x, IN REAL
y)
      EXTERN METHOD REAL interp3D(IN ENUM T_INTERP tin, IN ENUM T_INTERP tex, IN REAL x, IN REAL
y, IN REAL z)

       -- Interpolation functions for tables with default interpolation methods
      EXTERN METHOD REAL interpd1D(IN REAL x)
      EXTERN METHOD REAL interpd2D(IN REAL x, IN REAL y)
      EXTERN METHOD REAL interpd3D(IN REAL x, IN REAL y, IN REAL z)

      -- Interpolation with history functions for tables
      EXTERN METHOD REAL interpHist1D(IN ENUM T_INTERP tin, IN ENUM T_INTERP tex, IN REAL x, OUT
INTEGER i)
      EXTERN METHOD REAL interpHist2D(IN ENUM T_INTERP tin, IN ENUM T_INTERP tex, IN REAL x, IN
REAL y, OUT INTEGER i, OUT INTEGER j)
      EXTERN METHOD REAL interpHist3D(IN ENUM T_INTERP tin, IN ENUM T_INTERP tex, IN REAL x, IN
REAL y, IN REAL z, OUT INTEGER i, OUT INTEGER j, OUT INTEGER k)

       -- Default nterpolation with history functions for tables
      EXTERN METHOD REAL interpHistd1D(IN REAL x, OUT INTEGER i)
      EXTERN METHOD REAL interpHistd2D(IN REAL x, IN REAL y, OUT INTEGER i, OUT INTEGER j)
      EXTERN METHOD REAL interpHistd3D(IN REAL x, IN REAL y, IN REAL z, OUT INTEGER i, OUT
INTEGER j, OUT INTEGER k)

      -- Inverse interpolation functions for tables
      EXTERN METHOD REAL invInterp1D(IN ENUM T_INTERP tin, IN ENUM T_INTERP tex, IN REAL val)

      EXTERN METHOD REAL invInterpd1D(IN REAL val)
```

```
END CLASS INCLUDE "INTEG_simula.h" IN "INTEG.lib"
```

And the following items are defined in the Experiments library:

```
-- Enumerative types
--------------------------------------------------------------------------------
-- Calculation status
--------------------------------------------------------------------------------
ENUM statusIntegration = {INTEG_IDLE, INTEG_BEGIN, IS_STEP, IS_CINT, INTEG_END,
   SATISFY_RES, STEADY_BEGIN, STEADY_OUTPUT, IS_EVENT, INTEG_UNKNOW, STEADY_NOK,
   INTEG_NOK}

--------------------------------------------------------------------------------
-- Tolerance types (Relative and Absolute)
--------------------------------------------------------------------------------
ENUM t_tolTypes = {RELTOL, ABSTOL, FRACTOL}

--------------------------------------------------------------------------------
-- Available non-linear solver methods
--------------------------------------------------------------------------------
ENUM steadyMethods = {NR, MINPACK}

--------------------------------------------------------------------------------
-- Available integration methods
--------------------------------------------------------------------------------
ENUM integMethods = {DASSL, DASSL_SPARSE, RK4, LSODA, LSODA_SPARSE}


-- Global variables
ENUM steadyMethods SMETHOD           "Non-linear solver method"
ENUM integMethods  IMETHOD           "Integration method"
ENUM t_tolTypes    TOLTYPE           "Type of tolerance"
ENUM statusIntegration REPORT_MODE   "Report communications refresh mode"
REAL CINT                            "Communication interval"
BOOLEAN WARN_STEP_REDUCT             "Step reductions warning messages flag RUE/FALSE)"
BOOLEAN CJVITG                       "Non-repeatable res function warning msgs flag"
BOOLEAN TSMITG                       "Non-linearities warnings messages flag"
REAL TJNITG                          "Threshold for non-linearities"
BOOLEAN FIND_EVENTS                  "Enforce to find the exact time for events"
BOOLEAN FOLLOW_RT                    "Enforce real-time execution flag (TRUE/FALSE)"
INTEGER DEBUG_LEVEL                  "Debugging level (0, 1, 2, 3)"
INTEGER RDIGITS                      "Precision digits for reports"
INTEGER NSTEPS                       "Number of integ. steps in a comm. interval"
INTEGER MAXITER                      "Max. number of iterations for non-linear solvers"
INTEGER MAXFEVAL                     "Max. number of res. evals for non-linear solvers"
INTEGER MAXJACEVAL                   "Max. number of Jacobian evals for non-linear sol"
INTEGER MAXBROYDEN                   "Max. number of consecutive Jacobian updates"
REAL    TOLERANCE                    "Tolerance for non-linear solvers"
REAL    ABS_ERROR                    "Absolute error tolerance for transient solvers"
REAL    REL_ERROR                    "Relative error tolerance for transient solvers"
REAL    INIT_INTEG_STEP              "Minimun integrations step for transient solvers"
REAL    DASSL_SPARSE_U = 0.25        "U factor for DASSL sparse transient solver"
REAL    DASSL_SPARSE_EPS = 0.4       "EPS factor for DASSL sparse transient solver"

-- Fucntions
--------------------------------------------------------------------------------
-- Residues function call (take default parameters)
--------------------------------------------------------------------------------
"C++" FUNCTION NO_TYPE FRES()

--------------------------------------------------------------------------------
-- Steady functions
--------------------------------------------------------------------------------
"C++" FUNCTION ENUM statusIntegration STEADY(OUT INTEGER info= DUMMY_INTEGER)
"C++" FUNCTION NO_TYPE FREEZE(OUT REAL dynVar)
"C++" FUNCTION NO_TYPE UNFREEZE(OUT REAL dynVar)
```

```
"C++" FUNCTION BOOLEAN ISFROZEN(OUT REAL dynVar)

--------------------------------------------------------------------------------
-- Integration functions
--------------------------------------------------------------------------------
"C++" FUNCTION ENUM statusIntegration INTEG(OUT INTEGER i=DUMMY_INTEGER)
"C++" FUNCTION ENUM statusIntegration INTEG_TO(REAL tstop, REAL cint, OUT INTEGER
info=DUMMY_INTEGER)
"C++" FUNCTION ENUM statusIntegration INTEG_STEP(OUT INTEGER info=DUMMY_INTEGER)
"C++" FUNCTION ENUM statusIntegration INTEG_CINT(OUT INTEGER info=DUMMY_INTEGER)
"C++" FUNCTION ENUM statusIntegration INTEG_CINT_EVENT(OUT INTEGER info=DUMMY_INTEGER)
"C++" FUNCTION ENUM statusIntegration CONTINUE(OUT INTEGER i=DUMMY_INTEGER)

--------------------------------------------------------------------------------
-- Sets component's INIT block execution flag
--------------------------------------------------------------------------------
"C++" FUNCTION NO_TYPE SET_INIT_ACTIVE(BOOLEAN state)

--------------------------------------------------------------------------------
-- Executes component's INIT blocks
--------------------------------------------------------------------------------
"C++" FUNCTION NO_TYPE EXEC_INIT()

--------------------------------------------------------------------------------
-- Model discrete events functions
--------------------------------------------------------------------------------
"C++" FUNCTION NO_TYPE EXEC_EVENTS()

--------------------------------------------------------------------------------
-- Model reset functions
--------------------------------------------------------------------------------
"C++" FUNCTION NO_TYPE RESET()
"C++" FUNCTION NO_TYPE RESET_EVENTS()
"C++" FUNCTION NO_TYPE RESET_VARIABLES()

--------------------------------------------------------------------------------
-- Report and plots functions
--------------------------------------------------------------------------------
"C++" FUNCTION NO_TYPE REPORT_LIST(STRING name, STRING vars)
"C++" FUNCTION NO_TYPE REPORT_TABLE(STRING name, STRING vars)
"C++" FUNCTION NO_TYPE LOG_STATUS()
"C++" FUNCTION NO_TYPE NEW_BRANCH(STRING label)
"C++" FUNCTION NO_TYPE SET_REPORT_ACTIVE(STRING name, BOOLEAN status)
"C++" FUNCTION NO_TYPE REPORT_REFRESH()

-------------------------------------------------------------------------------
-- Save and restore state functions
--------------------------------------------------------------------------------
"C++" FUNCTION NO_TYPE SAVE_STATE(STRING str)
"C++" FUNCTION NO_TYPE SAVE_VARS(STRING fname, STRING pattern)
"C++" FUNCTION NO_TYPE RESTORE_STATE(STRING str)

--------------------------------------------------------------------------------
-- Return some statistic values about the jabobian and residues evaluation
--------------------------------------------------------------------------------
"C++" FUNCTION INTEGER getJacobianEvals() - get the total number of Jacobian evaluations
"C++" FUNCTION INTEGER getResiduesEvals () - get the total number of residue function calls
"C++" FUNCTION INTEGER getIterations() - get the number of iterations
"C++" FUNCTION INTEGER getJacobianEvalsIncr ()- get the partial number of Jac.evaluations
"C++" FUNCTION INTEGER getResiduesEvalsIncr ()- get the partial number of residue calls

--------------------------------------------------------------------------------
-- Set info for default non-linear solver (algebraic variables and residue
-- equations)
--------------------------------------------------------------------------------
"C++" FUNCTION BOOLEAN setAlgVariableInfo(STRING name, REAL dxAbs, REAL dxRel, REAL maxStepAbs,
REAL maxStepRel)
"C++" FUNCTION BOOLEAN setResidueInfo(INTEGER residue, REAL tolerance,ENUM t_tolTypes
tolType,REAL fRef=1.0)
"C++" FUNCTION REAL    evalNormResidue(INTEGER index,REAL lhs, REAL rhs,REAL fRef=1.0)
```

```
--------------------------------------------------------------------------------
-- Non-linear system of equations solvers
--------------------------------------------------------------------------------
"C++" FUNCTION NO_TYPE nlsolver(FUNC_PTR fcn, INTEGER n, OUT REAL x[], OUT REAL residues[], OUT
INTEGER info)
"C++" FUNCTION NO_TYPE nldsolver(FUNC_PTR fcn, INTEGER n, OUT REAL x[], OUT REAL residues[], OUT
INTEGER info, OUT STRING equations[], OUT STRING names[])

--------------------------------------------------------------------------------
-- Set flags for tracing, debugging, postprocess, and warnings
--------------------------------------------------------------------------------
"C++" FUNCTION NO_TYPE setTraceProgramme(BOOLEAN status)
"C++" FUNCTION BOOLEAN traceProgramme()
"C++" FUNCTION NO_TYPE setPprProgramme(BOOLEAN status)
"C++" FUNCTION BOOLEAN pprProgramme()
"C++" FUNCTION NO_TYPE setLogProgramme(BOOLEAN status)
"C++" FUNCTION BOOLEAN logProgramme()
"C++" FUNCTION NO_TYPE setWarnProgramme(BOOLEAN status)
"C++" FUNCTION BOOLEAN warnProgramme()

--------------------------------------------------------------------------------
-- Set the DEBUG_LEVEL variable. Prefered to change directly DEBUG_LEVEL
--------------------------------------------------------------------------------
"C++" FUNCTION NO_TYPE setDebugLevel(INTEGER level)

--------------------------------------------------------------------------------
-- Set a flag to TRUE(solve) or FALSE(not solve) the internal algebraic boxes
--------------------------------------------------------------------------------
"C++" FUNCTION NO_TYPE solveAlgebraicBoxes(BOOLEAN flag)
```

17. Appendix C: Syntax of EL

This Appendix describes in detail the full syntax of EL language in BNF form.

```
modelling_item_s: <2> ( use_stm
                        | extern_func_decl
                        | function_def
                        | port_def
                        | enum_def
                        | enum_set_def
                        | expl_var_decl
                        | component_def
                      | class_def
                        | preprocessor_item
                      | library_info
                        | EOL
                        ) *


preprocessor_item: <210> "#line " INTEGER_VALUE STRING_VALUE


//********************  METAINFO LIBRARIES  ********************

library_info: <211> "#LIBRARY_INFO" EOL* "#VERSION" '=' STRING_VALUE EOL* "#DATE" '='
STRING_VALUE EOL* "#AUTHOR" '=' STRING_VALUE EOL*
                  "#COMMENTS" '=' STRING_VALUE EOL* END "#LIBRARY_INFO"


//********************  PORT, ENUM and SET TYPE DEFINITION

enum_def        : <9> ENUM  IDENTIFIER  '='  '{' EOL* enum_elem_s EOL* '}' STRING_VALUE?

enum_set_def    : <10> enum_set_decl  '='  '{' EOL* enum_elem_s EOL* '}' STRING_VALUE?

enum_elem_s     : enum_elem ( ',' EOL* enum_elem )*

enum_elem       : <15> IDENTIFIER

port_def        : <12> PORT IDENTIFIER EOL*
                         ( IS_A scoped_id_s EOL*)? param_def_s? (SINGLE (IN|OUT)? EOL*)?  EOL*
STRING_VALUE?  EOL*
                         DECLS? port_var_decl_s (CONTINUOUS conc_stm_s)?
                       END PORT

port_var_decl   : <13> (BOUND|EXPL|DISCR)? ( (SUM|EQUAL) (IN|OUT)? )?  obj_decl
                                ( '=' init_expression )? (UNITS (STRING_VALUE | name))?
obj_range? STRING_VALUE? EOL+

port_var_decl_s : port_var_decl*


//*********************  COMPONENT  DEFINITION

component_def   : <14> ABSTRACT? COMPONENT IDENTIFIER EOL*
                        ( IS_A scoped_id_s EOL*)?
                         param_def_s?  EOL*
                        STRING_VALUE?  EOL*
                        ( PORTS port_decl_s )?
                      ( DATA var_decl_s )?
                        ( DECLS comp_decl_s )?
```

```
                      ( OBJECTS class_instace_stm_s  )?
                      ( TOPOLOGY topology_stm_s )?
                        ( INIT (PRIORITY (MINUS | PLUS)? INTEGER_VALUE)? seq_stm_s )?
                      ( DISCRETE discrete_stm_s )?
                        ( CONTINUOUS labelled_stm_s )?
                       END COMPONENT

port_decl       : <16>  mode scoped_id EOL* par_dat_map_s? general_id_s ( CARDINALITY range )?
STRING_VALUE?

port_decl_s     :       ( port_decl | EOL ) *

comp_decl_s     : <17> ( expl_var_decl
                       | enum_def
                       | enum_set_def
                       | close_decl
                       | EOL
                       ) *

close_decl      : <18> CLOSE identifier_s ( '=' init_expression )?

expl_var_decl   : (BOUND|EXPL|DISCR|ALG)? var_decl

topology_connnect_stm  : connect_stm
                      | topology_expand_stm
                      | topology_expand_block_stm

topology_connnect_stm_s : ( topology_connnect_stm | EOL ) *

topology_stm_s   : ( comp_instance
                   | path
                  | connect_stm
                  | topology_expand_stm
                  | topology_expand_block_stm
                  | EOL
                  )*

comp_instance   : <19> scoped_id  (EOL* par_dat_map_s)? EOL* general_id_s EOL* par_dat_map_s?
(CATALOGUE scoped_id)? EOL* STRING_VALUE?

path            : <20> PATH IDENTIFIER TO IDENTIFIER

connect_stm     : <21> CONNECT name_s (TO name)* TO name_s

topology_expand_stm : <34> EXPAND  '('  loop_schema  ')' EOL* topology_connnect_stm
                |      EXPAND  '(' expression ')' EOL* topology_connnect_stm

topology_expand_block_stm : <340> EXPAND_BLOCK  '('  loop_schema  ')' (topology_connnect_stm_s)?
END EXPAND_BLOCK
                |      EXPAND_BLOCK  '(' expression ')' (topology_connnect_stm_s)? END
EXPAND_BLOCK

//****************************  CONCURRENT STATEMENTS

conc_stm        : <26> implicit? expression  '='  expression EOL+ inverse_stm?
                | <27> implicit? expression  '='
                       ( ZONE '(' expression ( TOL expression )? ')' EOL* expression EOL*) +
                       OTHERS  EOL* expression '='
                | <271> implicit? expression  '='
                       IF '(' expression ')' EOL* expression EOL*
                                        ( ELSEIF '(' expression ')' EOL* expression EOL*)*
                       ELSE  EOL* expression EOL
                | <28> func_call EOL+ inverse_stm?
               | <281> SEQUENTIAL seq_stm_s END SEQUENTIAL
                | <29> conc_expand_stm
                | <290> conc_expand_block_stm
                | <291> if_eqts_stm

conc_stm_s      : ( conc_stm | EOL )*
```

```
inverse_stm      : INVERSE '(' full_name ')' EOL* conc_stm

conc_expand_stm : <32> EXPAND  '('  loop_schema  ')' EOL* conc_stm
                |       EXPAND  '(' expression ')' EOL* conc_stm

conc_expand_block_stm : <320> EXPAND_BLOCK  '('  loop_schema  ')' conc_stm_s END EXPAND_BLOCK
                |       EXPAND_BLOCK  '(' expression ')' conc_stm_s END EXPAND_BLOCK

if_eqts_stm      : <465> IF expression EOL* INSERT conc_stm_s elseif_insert_alt_s? (ELSE
conc_stm_s)? END IF

elseif_eqts_alt       : <47> ELSEIF expression INSERT conc_stm_s

elseif_insert_alt_s     : (EOL* elseif_eqts_alt ) +

//*******************************  DISCRETE STATEMENTS

discrete_stm     : <33> disc_expand_stm
               | <33> disc_expand_block_stm
                | <33> when_stm
                | <33> assert_stm

discrete_stm_s   : ( discrete_stm | EOL ) *

disc_expand_stm : <34> EXPAND  '('  loop_schema  ')' EOL* discrete_stm
                |       EXPAND  '(' expression ')' EOL* discrete_stm

disc_expand_block_stm : <340> EXPAND_BLOCK  '('  loop_schema  ')' (discrete_stm_s)? END
EXPAND_BLOCK
                |       EXPAND_BLOCK  '(' expression ')' (discrete_stm_s)? END EXPAND_BLOCK

when_stm         : <35> WHEN expression ( TOL expression )?  EOL* THEN seq_stm_s END WHEN

//**************************** EXPRESSIONS  **********************

labelled_stm     : <84> ( LT ':'? IDENTIFIER GT EOL*)?  conc_stm

labelled_stm_s   : ( labelled_stm | EOL )*

implicit         : IMPL '(' (name | vname_deriv)? ')' EOL*

//************************ USE CLAUSE ******************

use_stm          : <3> USE IDENTIFIER (VERSION STRING_VALUE)?

//************************ EXTERN AND EL FUNCTION   **********

extern_func_decl: <1200> STRING_VALUE function_header STRING_VALUE? EOL*(IN EOL* STRING_VALUE+
EOL)?

function_def     : <5> function_header EOL* STRING_VALUE? EOL*
                                ( DECLS var_decl_s )?
                                ( OBJECTS class_instace_stm_s   )?
                                  BODY seq_stm_s
                                  END FUNCTION

function_header : <6> FUNCTION data_type IDENTIFIER EOL* '(' EOL* func_arg_decl_s EOL* ')'

func_arg_decl_s : <7> ( func_arg_decl ( ',' EOL* func_arg_decl )* ) ?

func_arg_decl    : <8> mode? func_obj_decl ('=' expression)? (UNITS (STRING_VALUE | name))?
STRING_VALUE?
                |      enum_set_decl ( '=' expression )? (UNITS (STRING_VALUE | name))?
STRING_VALUE?

enum_set_decl    : <11> SET_OF '(' scoped_id ')' IDENTIFIER

//*********************** SEQUENTIAL STATEMENTS  ************

seq_stm_s        : <39> ( seq_stm | EOL ) *
```

```
seq_stm          :       write_stm
                       | assign_stm EOL
                       | seq_func_call EOL
                       | if_stm
                       | for_stm
                       | while_stm
                       | seq_assert_stm
                       | seq_stop_stm
                       | return_stm EOL
                      | issueError_stm

if_stm           : <46> IF expression EOL* THEN seq_stm_s elseif_alt_s? ( ELSE seq_stm_s )?
                   END IF

elseif_alt       : <47> ELSEIF expression THEN seq_stm_s

elseif_alt_s     : (EOL* elseif_alt ) +

write_stm       : <471> WRITE '(' EOL* (STRING_VALUE | IDENTIFIER) EOL* (',' EOL* expression_s)?
')'
   | <471> WRITEF '(' EOL* (STRING_VALUE | name | func_call) EOL* ',' EOL* (STRING_VALUE |
IDENTIFIER) EOL* (',' EOL* expression_s)? ')'
   | <471> WRITES '(' EOL* name EOL* ',' EOL* (STRING_VALUE | IDENTIFIER) EOL* (',' EOL*
expression_s)? ')'

issueError_stm: <472> issueError '(' INTEGER_VALUE  EOL* (',' EOL*
          expression_s)? ')'

for_stm          : <48> FOR  '(' seq_for_schema  ')'  seq_stm_s END FOR

while_stm        : <49> WHILE expression  seq_stm_s END WHILE

seq_for_schema   : <51> assign_stm_s? ';' expression ';' assign_stm_s?
                 | <52> loop_schema

assert_stm       : <36> ASSERT expression (NOTE | WARNING | ERROR | KILLPOINT
                | FATAL ) expression

seq_assert_stm   : <99> assert_stm

seq_stop_stm   : <99> STOP STRING_VALUE

seq_func_call    : func_call

assign_stm       : <44> full_name '=' expression (AFTER expression)?
                 | <44> full_name "+=" expression (AFTER expression)?
                 | <44> full_name "-=" expression (AFTER expression)?

func_call        : <45> name  '('  expression_s? ')'

return_stm       : <50> RETURN expression?

assign_stm_s     : <53> assign_stm ( ',' EOL* assign_stm )*

//*************** AUXILIARY RULES FOR DEFINITIONS AND DECLARATIONS  ******

data_type        : <54> REAL
                 | <54> INTEGER
                 | <54> STRING
                | <54> FILEPATH
                 | <54> BOOLEAN
                 | <54> FUNC_PTR
                | <54> OBJ_PTR
                 | <54> ENUM scoped_id
                 | <54> TABLE_1D
                 | <54> TABLE_2D
                 | <54> TABLE_3D
                 | <54> NO_TYPE

mode             : <55> IN
```

```
                | <55> OUT

identifier_s    : <56> IDENTIFIER ( ',' EOL* IDENTIFIER )*

scoped_id       : <57> IDENTIFIER ( '.' IDENTIFIER )?

scoped_id_s     : scoped_id ( ',' EOL* scoped_id )*

general_id      : <58> IDENTIFIER ( '[' expression_s  ']' ) ?

func_general_id : <65> IDENTIFIER ( '[' expression_s?  ']' ) ?

general_id_s    :      general_id ( ',' EOL* general_id )*

var_decl        : <59> CONST? data_type general_id_s ( '=' init_expression )? (UNITS
(STRING_VALUE | name))? obj_range? STRING_VALUE?

var_decl_s      :     ( var_decl | EOL )*

obj_decl        : <60> data_type general_id

func_obj_decl   : <60> data_type func_general_id

param_def_s     : <61> '(' EOL* param_def ( ',' EOL* param_def )*  EOL* ')' EOL*

param_def       : <62> obj_decl ( '=' expression )? (UNITS (STRING_VALUE | name))? STRING_VALUE?
                | <63> enum_set_decl ( '=' expression )? (UNITS (STRING_VALUE | name))?
STRING_VALUE?

//**************************** EXPRESSIONS  *****************

expression      : <66> single_term
                | <67> expression POW_OP expression
                | <68> PLUS   expression %prec UN_OP
                | <68> MINUS expression %prec UN_OP
                | <67> expression STAR   expression
                | <67> expression SLASH expression
                | <67> expression PLUS   expression
                | <67> expression MINUS expression
                | <69> NOT expression
                | <70> expression EQ_EQ expression
                | <70> expression N_EQ   expression
                | <70> expression LT     expression
                | <70> expression GT     expression
                | <70> expression LE     expression
                | <70> expression GE     expression
                | <71> expression OR     expression
                | <71> expression AND    expression
                | <72> '(' expression ')'

single_term     : <73> INTEGER_VALUE
                | <73> DOUBLE_VALUE
                | <73> STRING_VALUE
                | <73> FALSE
                | <73> TRUE
               | <79> name
                | <74> vname_deriv
                | <76> func_call
                | <78> SUM '(' loop_schema1  ';'  expression ')'

loop_schema1    : <80> IDENTIFIER IN range1 ( EXCEPT expression_s )?

range1          : <81> expression
                | <81> expression ',' expressio

init_expression : <64> '{' EOL* init_expression_s EOL* '}'
                |      expression

name            : <79> IDENTIFIER
                |      name  '.' IDENTIFIER
                |      name '[' expression_s ']'
```

```
name_s          :      name ( ',' EOL* name )*

full_name       : <79> name
                | <75> name "'"+

vname_deriv     : <751> name "'"+

expression_s    : <83> expression ( ',' EOL* expression )*

init_expression_s: <83> init_expression ( ',' EOL* init_expression )*

loop_schema     : <80> IDENTIFIER IN range ( EXCEPT expression_s )?

range           : <81> expression
                | <81> expression ',' expression

obj_range       : <82> RANGE range

par_dat_map_s   : <24> '(' EOL* par_dat_map ( ',' EOL* par_dat_map )*  EOL* ')'

par_dat_map     :  <25> (expression  '=')? init_expression

class_instace_stm_s   : ( class_instance | EOL )*

class_instance    : PRIVATE? scoped_id  EOL* general_id_s EOL* STRING_VALUE?  EOL*

//************************ CLASS DEFINITION   ****************************

class_def    : <1202> EXTERN? CLASS IDENTIFIER EOL*
                      ( IS_A scoped_id_s EOL*)?
                      STRING_VALUE?  EOL*
                      ( DECLS      var_object_decl_s )?
                      ( OBJECTS  class_instace_stm_s )?
                       ( METHODS  method_def_s )?
                       END CLASS EOL*
                      (INCLUDE EOL* STRING_VALUE)? EOL* (IN EOL* STRING_VALUE+ EOL)?

method_def_s     : <1203>( method_def | extern_method_header | EOL )*

method_header : <6> PRIVATE? METHOD data_type IDENTIFIER EOL* '(' EOL* func_arg_decl_s EOL* ')'
EOL* STRING_VALUE?

extern_method_header : <6> EXTERN METHOD data_type IDENTIFIER EOL* '(' EOL* func_arg_decl_s EOL*
')' EOL* STRING_VALUE?

method_def     : <5> method_header
                      ( DECLS var_decl_s )?
                      ( OBJECTS class_instace_stm_s   )?
                      (  BODY seq_stm_s )?
                      (  END METHOD )?
                      ( EOL | yy_eof )

var_object_decl :  PRIVATE? CONST? data_type general_id_s ( '=' init_expression )? (UNITS
(STRING_VALUE | name))? STRING_VALUE? EOL*

var_object_decl_s   :  ( var_object_decl | EOL )*

//**********************  EXPERIMENT  DEFINITION  ****************


experiment_def    :  experiment_item_s EOL* EXPERIMENT IDENTIFIER EOL* ON scoped_id EOL*
                        ( DECLS var_decl_s )?
                      ( OBJECTS class_instace_stm_s   )?
                      ( INIT expt_assign_stm_s )?
                      ( BOUNDS expt_assign_stm_s )?
                        ( BODY seq_stm_s )?
                       END EXPERIMENT EOL

expt_assign_stm_s: ( assign_stm? EOL )*
```

```
experiment_item_s:  ( function_def
                      | extern_func_decl
                      | use_stm
                      | EOL
                    ) *
```

18. Appendix D: Validity of STATEMENT Instructions

The table below shows the usage of three types of EL instructions described in this Manual: sequential, discrete and continuous, for each of the action blocks of components, ports and functions (an empty cell indicates it is not applicable).

Block	Sequential Statements	Discrete Statements	Continuous Statements
INIT of COMPONENT	YES		
DISCRETE of COMPONENT		YES	
WHEN actions	YES		
CONTINUOUS of COMPONENT			YES
CONTINUOUS of PORT			YES
BODY of FUNCTION	YES		
BODY of EXPERIMENT	YES		
BODY of METHODS	YES		

19. Appendix E: List of errors

This section shows the different errors that may be presented to the EcosimPro user during the compilation of models.

The format of EcosimPro error messages is:

ID SEVERITY MESSAGE

The ID is the error number, the severity ranges from 0 (only warning) to 3 (exit). The MESSAGE is the warning/error message

```
  1 3 No input file specified.
  2 2 Unable to open file "%s".
  3 1 Ignoring unknown command line option "%s".
  5 1 Unable to open syntax assist file "%s".\nUsing default syntax assists.
  6 3 More than one input file specified "%s".
  7 1 Minimum display argument "%d" exceeds maximum severity 5.\nReplacing with default %d.
  8 1 Minimum display argument "%d" less than lowest severity 0.\nReplacing with default %d.
  9 1 Maximum errors abort argument "%d" less than 1.\nReplacing with default %d.
 10 1 Maximum warnings abort argument "%d" less than 1.\nReplacing with default %d.
 11 1 The option "%s" requires a filename following it.\nIgnoring the option.
 12 1 The option "%s" requires a number following it.\nIgnoring the option.
 13 1 Unable to open error file "%s".\nUsing default error message.
 14 3 Unable to open string because input is already opened.\nCheck for missing close on a
previous input open.
100 2 SYNTAX ERROR while processing "%s".
110 2 SYNTAX ERROR while lexing character "%c" \nafter seeing "%s"
125 1 Missing syntax assist for production %d.
200 3 Cannot satisfy residues\nReason: %s\nRead log file for more details
201 3 Cannot converge in the integration step from time %f to time %f\nReason: %s\nRead log file
for more details
202 1 Cannot find variable %s in current model
203 3 Cannot open file %s (maybe either it does not exist or it has no write permission)
204 0 Cannot find steady state\nReason: %s
205 2 Either you cannot modify variable %s (only DATA or BOUNDARY are allowed)\nor the data type
is not appropiate
206 3 Format of file %s is wrong
207 3 Cannot converge after an event occurs. Number of allowed iterations exceeded (%d)\nRead log
file for more details
208 3 System is singular (in box %d, factor U(i,i), i= %d), \nit is either badly modelled or the
data is incorrect
209 1 Trying to get the variable %s with a wrong type
210 2 Trying to get the physical address of variable "%s" with a not permitted type.\nCheck the
User Manual.
211 1 Cannot open file %s (maybe either it does not exist or it has no write permission)
212 0 ASSERT NOTE failed: %s
213 1 ASSERT WARNING failed: %s
214 2 ASSERT ERROR failed: %s
215 3 ASSERT FATAL failed: %s
216 2 ASSERT KILLPOINT failed: %s.
217 0 Initial and final integration times are not correct (TSTART=%f, TSTOP=%f)
218 0 Communication interval is not correct (CINT=%f), changed to a default value
219 0 annot write properly in file %s
220 3 Cannot find delayed value for variable %s at time %f
221 3 Pulse width must be less than pulse period
222 0 Trying to interpolate in an empty table (perhaps is read afterwards).
223 2 Enumerative value is wrong: %s
```

```
224 3 Table %s has no inverse
225 0 Post-process file %s not found
226 0 Post-process file %s has not a correct format
227 0 Wrong format in line using RESTORE_STATE() (%s)
228 0 Variable %s cannot be set with value %s
229 1 The model has no state variables, the dense method %s is used instead
230 3 Cannot find file %s
231 3 Error in function dimArray...() Array address (%s) not found
232 3 Error in function sizeArray...() Second argument (%d) in function is wrong (valid values
are from 0 to %d)
233 0 At time %g Jacobian nonlinear measure is %g\nVariable name: %s\nVariable type: %s\nDAE
equation with nonlinearity problems:\n%s\nTo deactivate message set TSMITG to FALSE or set
threshold TJNITG to a big value
234 0 At time %g the residues function is not repeatable in equation\n%s\nPossible causes: \n   -
Use of global data in a function\n   - A boundary condition is a function of a model variable\nTo
deactivate message set CJVITG to FALSE
235 0 At time %g integration timestep divided by %g\nWorst variable: %s\nTo deactivate message
set WARN_STEP_REDUCT to FALSE
236 3 File %s was not correctly preprocessed:\nCheck that file and include files exists.\nCheck
the content of include files.\nCheck that #define directive is not directly used in EL code\n(Use
#define directive only inside an include file)
237 3 While preprocessing file %s
238 2 %s cannot be set with an enumerative type declaration
239 2 %s must be algebraic or dynamic variables. Dismissed this setting.
240 2 Function %s called with wrong argument %s=%s. Valid range is %s
241 0 %s
242 1 %s
243 2 %s
244 3 %s
245 3 Wrong dimension for interpolation in table "%s"
246 3 Invalid interpolation method "%s" for "%s" table
247 3 Invalid interpolation method "%s" for "CONSTANT" extrapolation in "%s" table
248 3 Invalid extrapolation method "%s" for "%s" table
249 3 Value out of domain for "%s" table
250 3 Error opening %s file "%s":\n%s
251 3 Error reading %s file (name:%s). Reason: %s
255 3 Undefined default interpolation method for "%s" table
256 3 Undefined default extrapolation method for "%s" table
257 1 RANGE of variable "%s" violated, value = %g, valid range = [%g,%g]
258 3 Error code %d not found in erros table for library %s
262 3 %s "%s" doesn't exist
263 3 Vriable "%s" is not "%s"
264 3 Invalid value for variable "%s"
300 3 Maximum number of errors of %d exceeded.
301 3 Maximum number of warnings of %d exceeded.
302 5 Previous maximum severity errors prevent further processing.\nCompilation aborted.
400 3 Unexpected table error while processing lexer action code case %d.
401 3 Unexpected table error while processing parser action %d,\naction code case %d is not
legal.
402 3 Parser action %d has unknown opcode %d.
403 3 While executing parser action %d,\nthe parser opcode was overwritten with unknown value %d.
404 3 Lexer action %d has unknown opcode %d.
405 3 While executing lexer action %d,\nthe lexer opcode was overwritten with unknown value %d.
406 4 Not enough memory to process input file.
407 3 Keyword "%s" conflicts with another symbol of same name.\nThe keyword was not installed.
501 2 Not closed string.
502 2 Illegal identifier
601 2 Symbol %s not found \n
602 2 left of .%s must have COMPONENT/PORT type\n
603 2 This expression is incompatible with the argument declaration \n
604 2 Types in expression are different\n
605 2 Symbol %s %s not found or it is obsolete\n
606 2 Symbol %s redeclared\n
607 2 Element %s in SET_OF %s not present in parent ENUM\n
608 2 The name %s is not a valid name of a %s\n
609 2 Expression incompatible with declaration\n
610 2 Invalid expression for an argument of mode OUT \n
611 2 Variable/Method %s inside component/port/object unknown\n
612 2 Datum "%s" does not appear in the public interface \nor is already closed (using CLOSE
statement)\n
```

```
613 3 Only allowed one path\n
614 2 Trying to initialise twice the same datum  = %s\n
615 2 Datum "%s" has not been initialised\n
616 2 You have to name all data or none, but not a mixture\n
617 2 Language specified (%s) not valid\n
618 2 Missing index in array variable\n
619 2 Illegal type in time expression
620 2 Trying to initialise more data that in \nthe definition (there are %d data)\n
621 2 Referenced variable is not an array\n
622 2 %s must be a valid ENUM type\n
623 2 This variable cannot be assigned in this context\n
624 2 Construction parameter or datum must use a valid ID\n
625 2 It must be a string expression\n
626 2 Symbol %s is not a port definition\n
627 2 Expressions in arrays must be integer or ENUM\n
628 2 Number of arguments different from definition\n
629 2 Index in array is wrong (probably it takes values out of limits)\n
630 2 Initial value(s) types not compatible with definition\n
631 2 Ranges are only allowed for REAL and INTEGER types in the components\n
632 2 Initial value must be an ENUM or SET_OF type\n
633 2 Wrong number of index\n
634 2 Return data type not compatible with assignment expression\n
635 2 Tolerance must be a REAL number\n
636 2 Exception data incompatible\n
637 2 Trying to inherit from a wrong item (non component/class)\n
638 2 Derivatives can be applied only to REAL variables\n
639 2 Problems with the dimensions in the initialisation\n
640 2 Expression (%s) needs to be fixed at generation time.\nOnly construction parameters and
closed data/variables (with CLOSE) \ncan be used in these expressions\n
641 2 Operation invalid\n
642 2 Cannot instantiate an abstract component\n
643 2 Library %s unknown\n
644 2 Expected NUMBER type (REAL or INTEGER) in expression\n
645 2 Expected BOOLEAN value in expression\n
646 2 An array index must be either INTEGER or ENUM variable declared in\nthe CONSTRUCTION
PARAMETERS block (DATA not valid)\n
647 2 Const variable has no initialisation\n
648 2 Returned value does not match the definition\n
649 2 Last statement in this function must be a return\n
650 2 Return statement incorrect here\n
651 2 Derivatives can be applied only to valid variables\n
652 2 Due to previous errors, inconsistent data information
653 2 Initial value contains current initialised variable
654 2 Not found in any parent component an equation with label "%s" to override\n
655 2 SUM variables must be of type REAL
656 2 This object is not a function
657 2 NOT USED
658 2 A table is not allowed in this context
659 2 An array of tables is not allowed
660 2 Table aggregates must contain numbers
661 2 The function must have, at least, one  OUT argument
662 2 Equations of non REAL type must denote an equivalence between two variables
663 2 Input values of a table must be ordered
664 2 A derivative can't appear in a function
665 2 Range is not allowed in current scope
666 2 Argument type not valid in FORTRAN
667 2 Some C function arguments are not valid
668 2 A timed assignment is not allowed in a FUNCTION
669 2 AFTER expression (%s) cannot be negative (%f)
670 2 Ranges are only valid for numerical variables
671 2 Illegal access to declaration "%s" in instantiated component
672 2 Cannot use delay expression in this context
673 2 Expected "=" assignment in this context
674 2 Variable TIME Cannot be used in non experiment functions
675 2 Selected var must be a non-constant REAL variable
676 2 Variables to freeze/unfreeze must be dynamic
677 2 CONST not allowed here
678 2 NO_TYPE is an illegal type for objects
679 2 FUNC_PTR/OBJ_PTR types is only allowed as function argument
680 2 FUNC_PTR/OBJ_PTR types can only be passed to another function argument
681 2 This argument must be a constant
```

```
682 2 The equation label "%s" defined twice (maybe in a parent component)
683 2 Initial value is not in range
684 2 A constant cannot be modified
685 2 A datum (DATA type) cannot be modified in this context
686 2 Variable iterative cannot be modified
687 2 A common ancestor is not allowed in multiple inheritance ("%s" component/class)
688 2 Using multiple inheritance,\ndetected two inherited symbols with the same name "%s"
689 2 Using multiple inheritance,\ntwo inherited equations labels cannot have the same name ("%s"
label)
690 2 This string cannot be empty
691 2 This statement only applies to Construction Parameters, Data or Ports. "%s" is not allowed
692 2 Declaration "%s" is already closed.
693 2 "%s" port variables must be closed by equations (if necessary)
694 2 An expression value is necessary to close "%s"
695 2 Element "%s" does not belong to the visible interface (Closed internally)
696 2 Inverse statements are only allowed for equations
697 2 This statement is not allowed as inverse statement
698 2 Cannot access a component or port definition in this context
699 2 The range values must be integers
700 2 Two multiple ports Cannot be connected together. Put connections on one of them.
701 2 Intermediate names must be component instance names
702 2 Names in a connection must correspond to ports or component instances
703 2 Cannot connect directly instances of component "%s" (component without PATH declaration)
704 2 Connected ports must be of the same type
705 2 Connection must identify a single port, it is not valid to denote an array port
706 2 "%s" should be a port declaration
707 2 The port name %s is not valid
708 2 Auto-instantiation is not allowed
709 1 This port is implicitly restricted to single IN connections (no SUM/EQUAL flag)
710 1 This port is implicitly restricted to single OUT connections(no SUM/EQUAL flag)
711 1 This port variable implicitly forces to single IN connections
712 1 This port variable implicitly forces to single OUT connections
713 2 Definition of these ports doesn't allow connections of the same effective mode
714 2 The port def. of this multi-connected port imposes SINGLE IN connections
715 2 The port def. of this multi-connected port imposes SINGLE OUT connections
716 2 The port def. forbids connection between two multiple ports ( this and %s )
717 1 Eliminating auto-connection
718 2 Return type not valid in FORTRAN
719 2 Variable for delay not valid, it must be a continuous model variable not constant
720 2 Boundary variable with assignment cannot have an AFTER statement associated
721 1 You are using the statement "LIBRARY %s", but your source file is in other library
722 2 Range values cannot be negative
723 2 BOUND,EXPL,DISCR and ALG qualifiers can be applied only to REAL variables
724 2 Missing default value for argument %d.\nDefault arguments go from right to the left. \nNo
argument in between without default value is allowed.
750 2 Equation should assign explicitly the variable "%s"
751 2 Initialisation for boundaries can contain only:\nDATA, constants, functions, experiment
variables and TIME variable\nThey cannot contain:\nboundaries, state, algebraic or derivative
variables
752 2 Variable %s already present in the model, use a different name
753 2 Wrong experiment name "%s" in experiment code. It should be "%s".
754 2 Wrong partition name "%s" in experiment code. It should be "%s".
755 2 Construction parameter "%s" has not been initialised\n
756 2 Construction parameter "%s" does not appear in the public interface \nor it is already
closed (with CLOSE statement)\n
757 2 Trying to initialise twice the same construction parameter  = %s\n
758 2 You have to name all construction parameters or none, but not a mixture\n
759 2 Trying to initialise more construction parameters that in \nthe definition (there are %d
construction parameters defined)\n
760 2 Construction parameters must be initalised with constants or \nother construction
parameters (DATA are not valid)
761 1 Item %s has been compiled from file %s,\nbut previously was compiled from file %s\n(this
can lead to problems).
762 1 Cannot document %s component instance
763 0 Function %s is using the old format.\nYou can remove the last argument (anyway it will
work)
764 2 A SEQUENTIAL statement can not be empty
765 2 DATA cannot be initalised with expressions containing variables
766 2 %s catalog name is not valid for this object
```

767 2 Initial value for OUT arguments must be a valid DUMMY global variable\nValid OUT default variables are: DUMMY_REAL, DUMMY_INTEGER, DUMMY_BOOLEAN and DUMMY_STRING.\nFor example (OUT REAL x= DUMMY_REAL)
768 2 Arrays and tables cannot have default values when used as function arguments.
769 2 Multiplicity qualifiers are different from parent(s)
770 2 This port inherits from two or more ports with diferent multiplicity qualifiers
771 2 Not allowed to initialize arrays of objects of classes
772 2 SET_OF type is not allowed in classes
773 2 All construction arguments for classes require a default value.
774 2 It is not allowed to initialize a function pointer argument
775 2 If the class is external it requires an include file associated
776 2 If the class has an include file or some external object libraries associated\n it must be declared as external (with EXTERN)
777 2 %s is not a valid component name
778 2 The method name "%s" is used in a parent class with different arguments or return type. Only allowed this if they are identical.
779 2 %s cannot be set with an ENUM or SET_OF type name
780 2 Trying to access private attribute/method "%s"
781 0 %s
782 1 To update this item validate the corresponding partition
783 2 Version format "%s" is not correct. Valid syntax is number.number(.number) (e.g. 3.4 or 3.4.12)
784 2 Problems with library versions. Required library %s version %s, but the actual one is version %s
785 2 Argument type not valid in C
786 2 Argument type not valid in C++
787 2 The only valid type here is constant string (arrays are not valid)
800 2 Cannot create new library %s:\nReason: %s
801 2 Detected problems reading library %s.\nThe binary file (%s) was corrupted (probably due a system failure in last session).\nThis library has been created again and it is needed to recompile it manually
802 2 Library "%s" was stored with name "%s"
803 2 Cannot save library %s in file %s:\n\nMaybe the file has read only permission
804 2 Cannot find item "%s" in library "%s"
805 2 Item "%s" is obsolete .
806 2 Cannot erase an opened item
807 2 Cannot open library item file %s \nEither the file does not exists or the binary format is out of date,\nin the last case recompile the element %s again.
808 2 Cannot erase library item file"%s" \nMaybe the file has read only permission
809 2 Can not save library item in file "%s":\nChecks whether this file exists with read only permission
810 1 Library "%s" not found. Probably you need to load this library since other existing is using it.
811 2 Item "%s" used in this component/port/function is visible in libraries %s and %s.\nYou need to specify a library prefix (e.g. MATH.PI)
812 2 Name "%s" used twice for the same item in the same file\naborted the compilation
813 2 Library "%s" already exists. Delete previous one before.
814 1 No libraries are active
815 1 Unknown directory associated to library "%s"
816 1 Directory "%s" associated to library "%s" does not exist
817 1 Creating default library "%s"
818 3 Cannot create the default library
819 2 Cannot set "%s" as the working library at startup
820 2 Cannot erase the file of library "%s"
821 3 A working library is necessary
822 2 Cannot open library item %s because of unopened library item %s
823 2 "WORK" Cannot be the name of a library
824 2 Cannot open partition file "%s".\nChecks that this file exits, if not, validate the partition again and it will be regenerated
825 2 Cannot save partition in file "%s". \nChecks whether this file exists with read only permission
826 2 AVAILABLE.
827 3 Cannot read the binary file for %s \n\nChecks that this file exits, if not, validate the partition again and it will be regenerated
828 2 Name too long for a global item "%s"
829 0 Compiling "%s" to update "%s"
830 2 Library %s is protected, you cannot modify it
831 2 Library %s is already locked
832 2 Wrong password to unlock library %s
833 2 Not allowed to set boundaries in experiment functions
834 2 Variable %s can not be set as %s

```
835 0 Cannot find component/port "%s" (probably its library is not open or it is already deleted)
900 2 Cannot generate a model from an abstract component
901 2 Cannot initialise %s
902 2 Cannot copy the concurrent statement ( in path "%s" )
903 2 The port of this variable does not exist ( in path "%s" )
904 2 This equation implies the equivalence of two constants ( in path "%s" )
905 2 Index expression should be evaluable ( %s "%s" )
906 2 The array size of "%s" must be greater than 0
907 2 Trying to connect array ports with different construction parameters.\nArray "%s" has range
"%s" and array "%s" has range "%s".\nProbably you have connected to ports or array ports with
different construction \nparameters (e.g. SET_OF's)
908 2 Exception expressions must be evaluable ( in path "%s" )
909 2 Some expressions in the range are not evaluable \nor some non constant expressions are
used( in path "%s" )
910 2 The tolerance must be a positive value at generation time ( in path "%s" )
911 2 Tolerance must be greater than zero (in path "%s" ).
912 2 Cannot copy the sequential statement ( in path "%s" )
913 2 "%s" must have an initial value
914 2 Incorrect equation
915 2 Cannot find the default function "%s"
916 2 Cannot use a variable index for a port array or array of components \nin sequential
statements ( %s "%s" )
917 3 Trying to divide by zero \n%s
918 2 This statement does not contain any unknown variable which can be matched
919 2 Cannot associate known variable "%s" to an argument of mode OUT
920 2 Var "%s" is explicit or was previously associated to another OUT argument
921 2 Cannot get a matching between vars and equations
922 2 Cannot tear the box of this statement. Maybe you are forcing some variables to be explicit
by using EXPL operator.
923 2 Cannot assign to an unknown derivative in INIT/WHEN statement
924 2 Delay expression must be evaluable and >= 0.0 ( in path "%s" )
925 2 Explicit var "%s" is already calculated by another statement
926 2 Explicit var "%s" not assigned explicitly
927 2 This statement can not be written explicitly
928 2 Bounds of variable "%s" must be evaluable
929 2 Variable of type "%s" is not a valid INIT variable  (it is neither dynamic nor derivative
nor algebraic)
930 2 This variable of type "%s" is not a boundary variable
931 2 Variable of type "%s" cannot be assigned in experiment body (or experiment function)
932 2 Trying to modify a constant variable
933 1 Trying to rename a REPORT/PLOT with a previously used name (%s)
934 2 Cannot find any REPORT with name %s
935 0 System needs to be initialised properly with the default libraries
936 2 Variable "%s" has not been matched to an equation. It won't be acceptable as
boundary.\n\tYou must write more equations
937 2 The following dynamic variables must be included in selection:\n\t%s
938 2 %d of the following dynamic variables must be included in selection:\n\t%s
939 2 It is needed variable %s to solve this high index problem,\nbut it has been selected in a
previous problem,\nplease start again
940 2 Detected redundant equations with incompatible independent values.\nRedundant equation is:
%s
941 3 Trying to get a root of a negative number \n%s
942 3 Trying to get the logarithm of a negative number \n%s
943 3 Trying to calculate function asin() or acos() with |value| > 1;\n%s
944 3 Trying to make pow( base<0, exp<0 or base<0,exp not integer)\n%s
945 1 Cannot derive this statement
946 3 System was determined as linear, but is not linear (problems in equation %s)
947 2 Illegal calculation of var "%s" in a ZONE branch
948 2 "%s" is not a DATA
949 0 Cannot find variable "%s" in current model
950 2 Current model is not a high index problem
951 2 Current model is a high index problem
952 2 Stop validating. Cannot get a valid matching
953 0 There is no dynamic variable "%s" in current high index problem
954 2 Cannot set variable "%s" as BOUNDARY
955 2 Cannot break box %d with old selection
956 2 Cannot select variable "%s" as ALGEBRAIC
957 2 There should be %d vars selected instead of %d
958 2 Variable "%s" was previously unclassified as dynamic
959 1 he array element will not be considered as a discrete variable
```

```
960 2 You have exceeded the limits of your Simulation Tool version\n(this version cannot cope
with %d equations, your maximum is %d)
961 2 You have exceeded the limits of your Simulation Tool version\n(this version cannot cope
with %d discrete events, your maximum is %d)
962 0 Detected a linear box too big (rows= %d,cols=%d), the tool will not try to reduce it
963 2 In a CLOSE statement, use a constant value to initialise variable %s
964 2 Cannot validate this partition, probably the mathematical has changed and the tool cannot
take the same conditions.\nTry to create a new one similar to the previous partition
965 2 Cannot generate this model. If you cannot find the exact error, please report the problem
to EA International.
966 2 Cannot compile due to abnormal conditions.\nReport the problem to EA International .
967 2 Left limit (%f) can not be greater than right limit (%f)
968 2 Violated the cardinality in port %s\nThe number of CONNECT statements for this port
(%d)\nis not in the defined cardinality range [%d,%d]
969 2 Connection invalid, this two ports have different construction parameters
970 1 Discarding repeated connection (CONNECT %s TO %s) in component %s
971 0 Working to reduce a big linear system (%d x %d).\nIt can take several minutes, please
wait...
972 2 Variable "%s" is calculated in more than a SEQUENTIAL block, this can lead to errors
973 2 Variable %s declared as EXPL cannot be calculated explicitly.
974 1 Proposed boundary %s cannot be set as boundary.
975 0 Changed tolerance type to FRACTOL, RELTOL is not allowed in local residues
999 1 Top Item Name "%s" seems to be equal to internal decl (only in debug mode)
```

20. Appendix F: Valid Formats for WRITE() Family of Functions

This documentation provides the different options which are available to be used in functions WRITE(), WRITEF() and WRITES() such as:

```
WRITE("Value of variable x is %.4f", x)
```

The part using the % sign is the format, in the last example it is %.4f. This is the format in which the variable x is to be output in this case: number of decimals, alignment, etc. This appendix outlines the different options that a modeller can use for writing this format. A format specification, which consists of optional and required fields, has the following form:

% [flags] [width] [.precision] {h | l | I64 | L}]type

Each field of the format specification is a single character or a number signifying a particular format option. The simplest format specification contains only the percent sign and a type character (for example, %s). If a percent sign is followed by a character which has no meaning as a format field, the character is copied to stdout. For example, to print a percent-sign character, use %%.

The optional fields, which appear before the type character, control other aspects of the formatting, as follows:

type

A required character that determines whether the associated argument is interpreted as a character, a string, or a number.

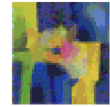
flags

Optional character or characters that control justification of output and printing of signs, blanks, decimal points, and octal and hexadecimal prefixes. More than one flag can appear in a format specification.

width

Optional number that specifies the minimum number of characters output.

precision



Optional number that specifies the maximum number of characters printed for all or part of the output field, or the minimum number of digits printed for integer values.

h | l | I64 | L

Optional prefixes to type - that specify the size of argument.

20.1. Flag Directives

The first optional field of the format specification is flags. A flag directive is a character that justifies output and print signs, blanks, decimal points, and octal and hexadecimal prefixes. More than one flag directive may appear in a format specification.

20.1.1.1. Flag Characters Table

Flag	Meaning	Default
–	Left align the result within the given field width.	Right align.
+	Prefix the output value with a sign (+ or –) if the output value is of a signed type.	Sign appears only for negative values (–).
0	If width is prefixed with 0, zeros are added until the minimum width is reached. If 0 and – appear, the 0 is ignored. If 0 is specified with an integer format (i, u, x, X, o, d) the 0 is ignored.	No padding.
blank (' ')	Prefix the output value with a blank if the output value is signed and positive; the blank is ignored if both the blank and + flags appear.	No blank appears.
#	When used with the o, x, or X format, the # flag prefixes any nonzero output value with 0, 0x, or 0X, respectively.	No blank appears.
	When used with the e, E, or f format, the # flag forces the output value to contain a decimal point in all cases.	Decimal point appears only if digits follow it.
	When used with the g or G format, the # flag forces the output value to contain a decimal point in all cases and prevents the truncation of trailing zeros. Ignored when used with c, d, i, u, or s.	Decimal point appears only if digits follow it. Trailing zeros are truncated.

20.2. Width Specification

The second optional field of the format specification is the width specification. The width argument is a nonnegative decimal integer controlling the minimum number of characters printed. If the number of characters in the output value is less than the

specified width, blanks are added to the left or the right of the values — depending on whether the – flag (for left alignment) is specified — until the minimum width is reached. If width is prefixed with 0, zeros are added until the minimum width is reached (not useful for left-aligned numbers).

The width specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified width, or if width is not given, all characters of the value are printed (subject to the precision specification).

If the width specification is an asterisk (*), an integer argument from the argument list supplies the value. The width argument must precede the value being formatted in the argument list. A nonexistent or small field width does not cause the truncation of a field; if the result of a conversion is wider than the field width, the field expands to contain the conversion result.

20.3. Precision Specification

The third optional field of the format specification is the precision specification. It specifies a nonnegative decimal integer, preceded by a period (.), which specifies the number of characters to be printed, the number of decimal places, or the number of significant digits. Unlike the width specification, the precision specification can cause either truncation of the output value or rounding of a floating-point value. If precision is specified as 0 and the value to be converted is 0, the result is no characters output, as shown below:

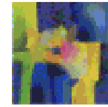
```
WRITE( "v= %.0d", 0 );       -- No characters output
```

If the precision specification is an asterisk (*), an int argument from the argument list supplies the value. The precision argument must precede the value being formatted in the argument list.

The type determines the interpretation of precision and the default when precision is omitted, as shown in the table.

□ How Precision Values Affect Type

Type	Meaning	Default
c, C	The precision has no effect.	Character is printed.
d, i, u, o, x, X	The precision specifies the minimum number of digits to be printed. If the number of digits in the argument is less than precision, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds precision.	Default precision is 1.



e, E	The precision specifies the number of digits to be printed after the decimal point. The last printed digit is rounded.	Default precision is 6; if precision is 0 or the period (.) appears without a number following it, no decimal point is printed.
f	The precision value specifies the number of digits after the decimal point. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.	Default precision is 6; if precision is 0, or if the period (.) appears without a number following it, no decimal point is printed.
g, G	The precision specifies the maximum number of significant digits printed.	Six significant digits are printed, with any trailing zeros truncated.
s, S	The precision specifies the maximum number of characters to be printed. Characters in excess of precision are not printed.	Characters are printed until a null character is encountered.

If the argument corresponding to a floating-point specifier is infinite, indefinite, or NaN, WRITE() gives the following output.

Value	Output
+ infinity	1.#INFrandom-digits
– infinity	–1.#INFrandom-digits
Indefinite (same as quiet NaN)	digit.#INDrandom-digits
NAN	digit.#NANrandom-digits

20.4. WRITE Type Field Characters

The type character is the only required format field; it appears after any optional format fields. The type character determines whether the associated argument is interpreted as a character, string, or number.

WRITE Type Field Characters Table

Character	Type	Output Format
c	int or wint_t	When used with WRITE functions, specifies a single-byte character
C	int or wint_t	When used with WRITE functions, specifies a wide character
d	int	Signed decimal integer
i	int	Signed decimal integer
o	int	Unsigned octal integer
u	int	Unsigned decimal integer
x	int	Unsigned hexadecimal integer, using “abcdef”
X	int	Unsigned hexadecimal integer, using “ABCDEF”

e	double	Signed value having the form [–]d.dddd e [sign]ddd where d is a single decimal digit, dddd is one or more decimal digits, ddd is exactly three decimal digits, and sign is + or –
E	double	Identical to the e format except that E rather than e introduces the exponent
f	double	Signed value having the form [–]dddd.dddd, where dddd is one or more decimal digits. The number of digits before the decimal point depends on the magnitude of the number, and the number of digits after the decimal point depends on the requested precision
g	double	Signed value printed in f or e format, whichever is more compact for the given value and precision. The e format is used only when the exponent of the value is less than –4 or greater than or equal to the precision argument. Trailing zeros are truncated, and the decimal point appears only if one or more digits follow it
G	double	Identical to the g format, except that E, rather than e, introduces the exponent (where appropriate)

21. APPENDIX G: XML FORMATS

21.1. TABLES IN XML

This section describes the XML syntax to represent tables. XML files are divided into the following parts:

- General XML header
- Table header
- Default interpolation method and valid methods
- Default extrapolation method and valid methods
- Values of the axes
- Result values
- Footer of the table

21.1.1.1. HEADER: Tag <?xml>

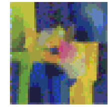
A mandatory initial line on any XML file with the following syntax:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

21.1.1.2. TABLE HEADER: Tag <table>

The "table" schema starts with the following fields:

- **type**: 1D, 2D or 3D
- **name**: name of the table
- **description**: description of the table
- **version**: current version of the table
- **cdate**: creation date
- **mdate**: modification date



□ **revision**: revision number

For example:

```
<table type="1D" name="PRsurge_vs_Wc_sec" description="Surge
pressure ratio versus corrected massflow rate" version="1.0"
cdate="01/01/2007" mdate="01/08/2007" revision="0.1">
```

21.1.1.3. INTERPOLATION METHOD: Tag <interp>

The default interpolation method ("default") and the valid interpolation methods during the simulation ("valid") must be indicated.

```
<interp default="SPLINE" valid="{CONSTANT, LINEAR, SPLINE}" />
```

The TABLE class allows interpolation and extrapolation in tables with a number of predefined methods. There is an enumeration in EL with all the identifiers of the methods, which are as follows:

```
ENUM T_INTERP = {DEFAULT,   FORBIDDEN, CONSTANT, LINEAR,
                 QUADRATIC, CUBIC,     SPLINE}
```

21.1.1.4. EXTRAPOLATION METHOD: Tag <extrap>

The default extrapolation method ("default" attribute) and the valid extrapolation methods during the simulation ("valid" attribute) must be indicated.

```
<extrap default="LINEAR" valid="{FORBIDDEN, CONSTANT, LINEAR,
SPLINE}" />
```

21.1.1.5. VALUES OF THE AXES: Tags: <axis1>, <axis2> and <axis3>

They are used to provide the data for the various axes (if the table is 1D, only axis1 shall be defined, if it is 2D, axis1 and axis2 shall be defined, and if it is 3D, axis1, axis2 and axis3 shall be defined). This row has the following fields:

□ **id**: name of the variable

□ **description**: description of the variable

□ **value**: list of values. It can be in EL format with symbols "{" "}" "," as separators.

For example:

```
<axis1 id="Wc" description="Corrected Massflow Rate (kg/sec)"
value="{0.21556, 0.34721, 0.50444, 0.58864, 0.86094, 0.93168}"
/>
```

21.1.1.6. RESULT VALUES: Tag <return>

This row has to include the numerical values of the results in the table. The total number of result values shall be the multiplication of the elements in axis1, axis2 and axis3. This field includes:

- **id**: result variable
- **description**: description of the result variable
- **value**: list of result values. It can be in EL format with symbols "{" "}" "," as separators.

For example:

```
  <return id="PRsurge" description="Surge Pressure Ratio (-)"
value="{1.03581, 1.10500, 1.21050, 1.28809, 1.34042, 1.40804,
1.46745, 1.52683, 1.59777, 1.69644}" />
```

21.1.1.7. TABLE FOOTER: Tag </table>

It closes the definition of the table. For example:

```
</table>
```

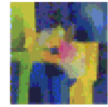
21.1.1.8. COMPLETE EXAMPLE

The following code shows an example of a 1D table in XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE eds SYSTEM "DTDFILE_FULLPATH">
<table type="1D" name="PRsurge_vs_Wc_sec" description="Surge pressure ratio versus corrected
massflow rate" version="1.0" cdate="01/01/2007" mdate="01/08/2007" revision="0.1">
  <interp default="SPLINE" valid="{CONSTANT, LINEAR, SPLINE}" />
  <extrap default="LINEAR" valid="{FORBIDDEN, CONSTANT, LINEAR, SPLINE}" />
  <axis1 id="Wc" description="Corrected Massflow Rate (kg/sec)" value="{0.21556, 0.34721,
0.50444, 0.58864, 0.65517, 0.74116, 0.77724, 0.81336, 0.86094, 0.93168}" />
  <return id="PRsurge" description="Surge Pressure Ratio (-)" value="{1.03581, 1.10500, 1.21050,
1.28809, 1.34042, 1.40804, 1.46745, 1.52683, 1.59777, 1.69644}" />
  </table>
```

The following code shows an example of a 3D table in XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE eds SYSTEM "DTDFILE_FULLPATH">
   <table type="3D" name="mu_T" description="Dynamic Viscosity as a function of T, FARB and
WAR)">
      <interp default="LINEAR" valid="{CONSTANT, LINEAR, SPLINE}" />
      <extrap default="CONSTANT" valid="{FORBIDDEN, CONSTANT, LINEAR, SPLINE}" />
      <axis1 id="WAR" description="Water to Air Ratio (-)" value="{ 0.2,0.4 }" />
      <axis2 id="T" description="Temeperature(K)" value="{ 0.7,0.8,0.9}" />
      <axis3 id="FARB" description="Burnt Fuel to Air Ratio (-)" value="{6.0,-2.0,-3}" />
      <return id="mu" description="Dynamic Viscosity (Ns/m2)" value="{{ {4.0,1.0,5.4}
,{6.0,3.0,1.7},{7.0,0.0 ,2}},{{2.0,5.0,-34},{3.0,7.0,-4},{5.0,9.12e-3,567}}}" />
   </table>
```



21.2. SYSTEM_IOREADER IN XML

This section describes the XML syntax used to represent configuration files for executions using the class SYSTEM_IOREADER. XML files are formed by the following parts:

- General XML header
- Header of the iofile
- Generics block
- Iolist block
- Reports block
- Footer of the table

21.2.1.1. XML HEADER: Tag <?xml>

This is an XML row that follows the format:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

21.2.1.2. HEADER: Tag <iofile>

The "iofile" tag starts with the following data:

- name: name of the iofile
- version: current version of the iofile
- type: type of iofile (it can be "NORMAL" or "DECK")

For example:

```
<iofile version="" name="myfile.xml" type="NORMAL">
```

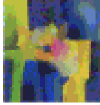
21.2.1.3. GENERIC BLOCK: Tag <generics>

Example of entries in this block:

```
<generic id="nsteps" value="5"/>
```

21.2.1.4. IOLIST BLOCK: Tag <iolist>

Example of entries in this block:




```
<ioitem id="variable" alias="alias" group="mygroup" type="REAL"
mode="INPUT-OUTPUT" inputCandidates="" inputValue="5.3"
inputUnits="m2" outputUnits=""/>
```

21.2.1.5. REPORT BLOCK: Tag <reports>

Example of entries in this block:

```
<report id="report_steady_default" format="LIST"
filename="my_file.txt" vars="*"/>
```

21.2.1.6. FOOTER: Tag </iofile>

This is used to close the definition of the data in the file. For example:

```
</iofile>
```

21.2.1.7. COMPLETE EXAMPLE

The following code provides an example of a SYSTEM_IOREADER file in XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<iofile version="" name="" type="NORMAL">
  <generics>
    <generic id="iterations_limit" value="3"/>
    <generic id="nsteps" value="5"/>
  </generics>
  <iolist>
    <ioitem id="variable" alias="alias" group="mygroup" type="REAL" mode="INPUT-OUTPUT"
inputCandidates="" inputValue="5.3" inputUnits="m2" outputUnits=""/>
  </iolist>
  <reports>
    <report id="report_steady_default" format="LIST" filename="my_file.txt" vars="*"/>
  </reports>
</iofile
```

22. Index by words

A

abs(), 249

ABS_ERROR, 162

ABSTOL, 224

ABSTRACT, 87

acos(), 249

AFTER, 48

aggregation, 4, 87

ALG, 96

AND, 17

Argument List in functions, 120

array, 28

asin(), 249

ASSERT sequential, 57

ASSERT, DISCRETE, 73

Assignments, 48

atan(), 249

atan2(), 249

Automatic documentation, 43

B

BODY, experiments, 162

BODY, functions, 124

BOOLEAN, 22

BOUND, 95

BOUNDS Block, 161

C

CARDINALITY, 90

cellCrossing1D, 137

cellCrossing2D, 137

cellCrossing3D, 137

cellLinearInterp1D, 138

cellLinearInterp2D, 138

cellLinearInterp3D, 138

Ch

checkFileExists, 24

C

CINT, 162

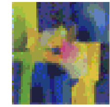
CJVITG, 189

CLASS, 195

clearTable1D(), 129

clearTable2D(), 129

clearTable3D(), 129



CLOSE, 97

comments, 10

COMPONENT, 85

CONNECT, 102

connection port, 77

CONST, 33

CONSTANT interpolation, 241

Construction parameters, 91

CONTINUE, 166

CONTINUOUS in COMPONENT, 110

CONTINUOUS in PORT, 80

Continuous statements, 59

cos(), 249

cosh(), 249

CUBIC interpolation, 241

D

DASSL, 167

DASSL_SPARSE, 167

DASSL_SPARSE_U, 193

DATA, 93

DEBUG_LEVEL, 163

DECLS in CLASS, 198

DECLS in COMPONENT, 93

DECLS in FUNCTION, 123

DECLS in PORT, 78

DEFAULT interpolation, 240

delay(), 126

dfpow2(), 249

dfsqrt(), 249

dimArrayBool, 143

dimArrayInt, 143

dimArrayName, 144

dimArrayReal, 143, 144

dimArrayString, 143

DISCR, 95

DISCRETE Block, 107

Discrete Statements, 70

E

encapsulation, 2

ENUM, 27

EQUAL, 80

EQUAL OUT, 83

ERROR, 58

evalNormResidue(), 222

EXCEPT, 61

EXEC_EVENTS(), 177

EXEC_INIT, 174

existsFile, 24

existsVariable, 203

exp(), 249

EXPAND in CONTINUOUS, 61

EXPAND in DISCRETE, 73

EXPAND_BLOCK in CONTINUOUS, 62

EXPAND_BLOCK in DISCRETE, 75

expandFilePath, 24

EXPERIMENT, 157

EXPL, 96

expression, 14

EXTERN CLASS, 211

EXTERN METHOD, 212

External functions, 124

F

FALSE, 22

FATAL, 58

FILEPATH, 23

FIND_EVENTS, 184

FOLLOW_RT, 183

FOR, 56

FORBIDDEN interpolation, 240

fpow2(), 249

FRACTOL, 224

FREEZE(), 227

Freezing Variables, 227

FRES, 193

fsqrt(), 249

FUNC_PTR, 26

FUNCTION, 119

Function calls, 49

FUNCTION in EXPERIMENT, 172

G

getArray1D(), 205

getArray2D(), 205

getArray3D(), 205

getIterations(), 254

getJacobianEvals, 185

getJacobianEvals(), 254

getJacobianEvalsIncr, 185

getNumberVars, 203

getResiduesEvals(), 185

getResiduesEvalsIncr, 185

getResidueValue(), 150

getValueAsString(), 204

getValueBool, 204

getValueEnum, 204

getValueInt, 204

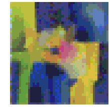
getValueReal, 203

getValueString, 204

getVarCategoryStr, 203

getVarName, 203

getVarTypeStr, 203



gnameBool, 51

gnameEnum, 51

gnameInt, 51

gnameReal, 51

gnameString, 51

gvalBool, 51

gvalEnum, 51

gvalEnum2, 51

gvalString, 51

I

IF-Equations, 66

IF-INSERT, 62

IF-THEN-ELSE, 55

IMETHOD, 163

IMPL operator, 117

IN in CLASS, 216

IN in FUNCTION, 120, 125

IN_STEADY, 169

IN_TRANSIENT, 169

INCLUDE, 211

Inheritance, 3, 86, 88

INIT in COMPONENT, 105

INIT in EXPERIMENT, 159

INIT_INTEG_STEP, 163

INSERT, 62

INTEG(), 165

INTEG_BEGIN, 163

INTEG_CINT(), 166

INTEG_END, 163

INTEG_STEP (), 166

INTEG_TO(), 165

INTEGER, 22

Integration Functions, 164

interpolation in tables, 131

INVERSE, 115

invSplineInterp1D, 133

IS_A in CLASS, 196

IS_A in COMPONENT, 88

IS_A in PORT, 79

IS_CINT, 163, 178

IS_EVENT, 178

IS_STEP, 163, 178

ISFROZEN, 228

issueError(), 151

K

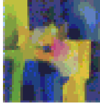
KILLPOINT, 58

L

Libraries, 39

LIBRARY, 37

LIBRARY_INFO, 40



EA Internacional

LINEAR interpolation, 241

linearInterp1D, 131

linearInterp2D, 131

linearInterp3D, 131

linearInterpHist1D, 132

linearInterpHist2D, 132

linearInterpHist3D, 132

log(), 249

LOG_STATUS, 142

log10(), 249

logProgramme, 209

M

Mathematical equations, 60

MAXBROYDEN, 225

MAXFEVAL, 225

MAXITER, 225

MAXJACEVAL, 225

METHOD, 200

METHODS, 200

MINPACK, 221

N

NEW_BRANCH(), 145

nldsolver, 232

nlsolver, 232

NO_TYPE, 20

NOT, 17

NOTE, 58

NSTEPS, 180

O

OBJ_PTR, 27

Object-oriented modelling, 2

OBJECTS in CLASS, 199

OBJECTS in COMPONENT, 98

OBJECTS in EXPERIMENT, 158

OBJECTS in FUNCTION, 123

ON in EXPERIMENT, 157

OR, 17

OTHERS, 65

OUT in FUNCTION, 120

P

Parameters, 91

Partitions, 153

PATH, 101

periodTimeTableInterp, 134

periodTimeTableStep, 134

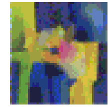
PORT, 77

Port connecting restrictions, 82

PORTS, 90

PRINT() function, 53

PRINTF(), 53



printTable1D(), 129

printTable2D(), 129

printTable3D(), 129

PRIORITY in INIT, 105

PRIVATE in DECLS, 199

PRIVATE in METHODS, 200

pulse(), 141

Q

QUADRATIC interpolation, 241

R

ramp(), 142

RANGE, 34

RDIGITS, 177

readTable1D(), 127

readTable2D(), 127

readTable3D(), 127

readTableCols1D(), 130

REAL, 21

regular expression, 247

REL_ERROR, 163

REPORT_LIST( ), 170

REPORT_MODE, 178

REPORT_REFRESH, 172

REPORT_TABLE(), 170

Reports generation, 169

Reserved Words, 12

RESET(), 163

RESET_EVENTS, 164

RESET_VARIABLES, 164

RESTORE_STATE, 181

RETURN, 58

RK4, 167

S

SAVE_STATE, 181

SAVE_VARS(), 181

scope of variables, 34

SEQUENTIAL, 68

Sequential statements, 47

SET_INIT_ACTIVE, 175

SET_OF, 30

SET_REPORT_ACTIVE, 171

setAlgVariableInfo, 226

setArray1D(), 205

setArray2D(), 205

setArray3D(), 205

setDebugLevel(), 186

setLogProgramme, 209

setofElem, 31

setofPos, 32

setofSize, 32

setResidueInfo, 222, 226

setTraceProgramme, 208

setValueBool, 204

setValueEnum, 204

setValueInt, 204

setValueReal, 204

setValueString, 204

setWarnProgramme, 209

sin(), 249

SINGLE IN, 82

sinh(), 249

sizeArrayBool, 143

sizeArrayInt, 143

sizeArrayName, 144

sizeArrayReal, 143, 144

sizeArrayString, 143

SMETHOD, 163, 225

solveAlgebraicBoxes(), 151

SPLINE interpolation, 241

splineInterp1D, 133

splineInterp2D, 133

spow2(), 249

sqrt(), 249

square(), 140

ssqrt(), 249

STEADY(), 167, 219

step(), 140

STOP, 58

STRING, 23

SUM, 16, 80

System Libraries, 42

SYSTEM_IOREADER in XML, 284

T

TABLE class, 237

TABLE in XML, 281

TABLE_1D., 25

TABLE_2D, 25

TABLE_3D, 25

tan(), 249

tanh(), 249

TIME, 37, 162

timeTableInterp, 134

timeTableStep, 134

TJNITG, 192

TOL in WHEN, 70

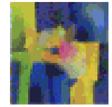
TOL in ZONE, 65

TOLERANCE, 163, 225

TOLTYPE, 163, 225

TOPOLOGY, 99

traceProgramme, 208



TRUE, 22

TSMITG, 191

TSTOP, 37, 162

U

UNFREEZE(), 227

UNITS, 33

unitsOfVariable, 149

USE, 41

V

VERSION, 41

versionReport, 149

Virtual equations, 113

W

WARN_STEP_REDUCT, 188

WARNING, 58

warnProgramme, 209

Waveform Functions, 140

WHEN, 70

WHILE, 56

Z

ZONE, 65

EA Internacional
Magallanes, 3. 28015 Madrid. SPAIN.
E-mail: info@ecosimpro.com
Web: www.ecosimpro.com

EA Internacional

© 1992-2008 EA Internacional. All rights reserved.