POLITECNICO DI MILANO

School of Industrial and Information Engineering

M.S.c. Automation & Control Engineering

Representation of PLC Ladder Diagrams in the Modelica Language

Supervisor
Prof. Leva Alberto

Author
Tsipoulidis Miltiadis

Contents

Chapter 1

Introduction

The goal of this thesis is to represent PLC programs written in the Ladder Diagram language in Modelica. In other words, we want to create a *User Friendly model library,* suitable for use in the OpenModelica environment, that looks like the commonly known – used PLC programming tools because OpenModelica is a strong tool already used and is to be used more and more in the nearest feature mainly by the Engineering community(more details about OpenModelica will be provided in the next Chapter).
Generally the idea is with this tool to set the foundations for the next generations of students – research community in order to develop it at a point where it can be used in the Industry as a professional tool.
Of course what needs to be also said here is that OpenModelica is constantly under development with the help of the research community (reports – feedback) so problems were also faced with errors of the language which led us to make a detour and solve the issue with a different approach.
From an application-oriented standpoint, the motivation of the presented work may be summarized as follows.

- PLCs are very common control devices, and the LD language is a very common tool for their programming. On the other hand, Modelica (and object-oriented modeling at large) provides a powerful paradigm to represent controlled plants with fidelity. However, bridging the two is far from trivial and may require quite complex co-simulation architectures. We wanted to see how feasible it is to have everything represented into one of the two worlds above, that for apparent reasons has to be modeling (not programming) one.
- On a similar front, we wanted to see whether or not and if the affirmative case up to what extent, one can use an equation-oriented language to represent a program, while preserving the visual assembling paradigm of the former.

The program will give the user the ability to create simple Rung programs using the blocks:

- Open contact
- Closed contact
- Supply Rail (the starting block of every rung with individual numbering of the Rung)

- Ground Rail (the ending block of each rung with individual numbering of the Rung)
- Coil

Afterwards the user can simulate the program and see the results.

More blocks are to be developed in the future with exactly the same logic(such as counters,timers,Structure Text blocks,logic gates, etc. . .).

The developer may also invent new blocks that can be handy as well or modify the existing ones. The limitation that may be there is the memory capacity of the PLC but this is a future discussion.

There is no work done before towards this direction hence everything done in this project is the fruit of

- research,
- solid background knowledge of both the Supervisor and the Author of the PLC ladder Language (Protocol IEC 61131-6)
- designing experience
- cooperation

Chapter 2

PLC – Modelica

The first part of this chapter will be devoted to the PLC,Ladder Language,IEC 61131-6 protocol while in the second part the Modelica Language will be explained explicitly (algorithims,functions, variables , events).

2.1 Programmable Logic Controller (PLC)

A Programmable Logic Controller, PLC or Programmable Controller is a digital computer used for automation, such as control of machinery on factory assembly lines, amusement rides, or light fixtures. PLCs are used in many industries and machines. Unlike general- purpose computers, the PLC is designed for multiple inputs and outputs arrangement, extended temperature ranges, immunity to electrical noise and resistance to vibration and impact. Programs to control machine operation are typically stored in battery- backed- up or non- volatile memory. A PLC is an example of a hard real-time system since output results must be produced in response to input conditions within a limited time, otherwise unintended operation will result.

2.1.1. PLC features

A PLC program is generally executed repeatedly while the controlled system is running. The status of physical input points is copied to an area of memory accessible to the processor which is called the I/O Image Table. The program is then run from its first instruction line to the last line. It takes some time for the processor of the PLC to evaluate all the lines and update the I/O image table with the status of outputs. This scan time may be a few milliseconds for a small program or on a fast processor, but older PLCs running very large programs could take much longer to execute the program.
A PLC has a fixed number of connections built in for inputs and outputs. However, expansions can be added if the base model has insufficient I/O.
A human-machine interface (HMI) can be used in order to interact with people for the purpose of configuration. HMIs are also referred to as man-machine interfaces (MMIs) and graphical user interface (GUIs).
A simple system may use buttons and lights to interact with the user.
Text displays are available as well as graphical touch screens. More complex systems use programming and monitoring software installed on a computer, with the PLC connected via a communication interface.
PLCs have built in communication ports, usually 9-pin RS-232, but optionally EIA- 485 or Ethernet. Modbus, BACnet or DF1 is usually included as one of the communications protocols. Other options include various fieldbuses such as DeviceNet or Profibus. Other communication protocols that may be used are listed in the list of automation protocols.
Most modern PLCs can communicate over a network with some other system, such as a computer running a SCADA (Supervisor Control and Data Acquisition) system or a web browser. PLCs used in larger I/O system may have peer- to- peer (P2P) communication between processors. This allows separate parts of a complex process to have individual control while allowing the subsystems to co- ordinate over the communication link. These communication links are also often used for HMI devices such as keypads or PC- type workstations.
A PLC is programmed using PLC programming tools, then the program is downloaded via connection cable or over a network to the PLC. The program is stored in the PLC either in battery- backed- up RAM or some other non- volatile flash memory.
Under the IEC 61131- 3 standard, by using based programming languages PLCs can be programmed. Most PLCs are programmed by Ladder Logic Diagram. It is a model which works as electromechanical control panel devices (such as the contact and coils of relays) which PLCs replaces.

The programming languages which are available are defined by IEC 61131- 3 standard:

1) Function block diagram (FBD),
2) Ladder diagram (LD),
3) Structured text (ST; similar to C programming language)
4) Instruction list (IL; similar to assembly language) and
5) Sequential function chart (SFC)

2.1.2. PLC Scanning Cycle

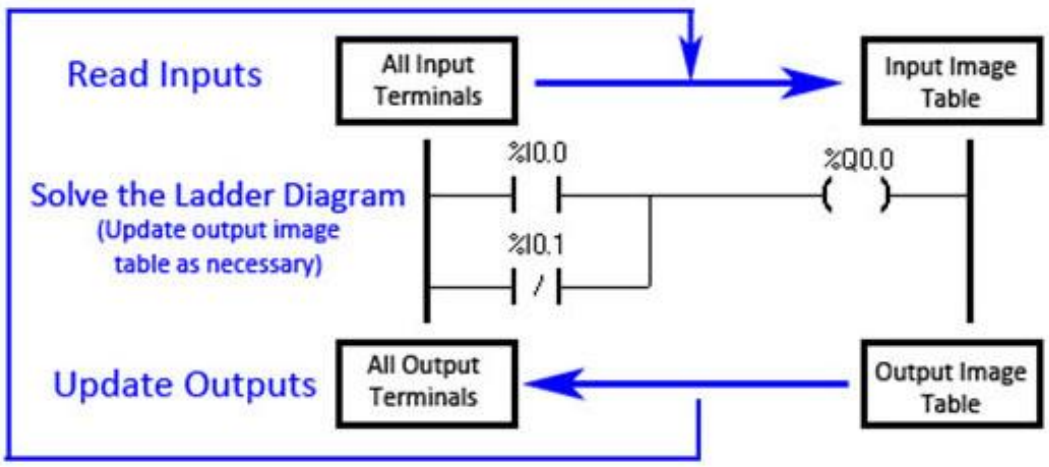
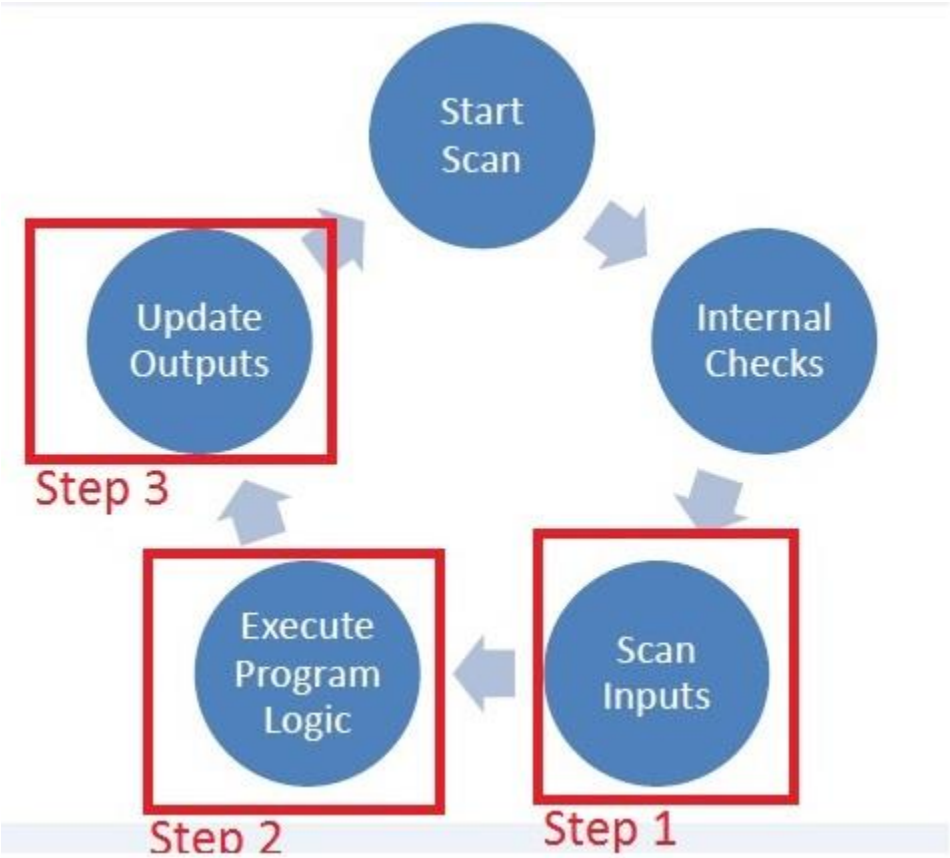


Fig.2 Scan Cycle

Processor scan sequence.

For the sake of simplicity, the following discussion is based on an **Allen-Bradley SLC-5/04**. The scan sequence of most other processors is identical or very similar. This discussion is limited to discrete (off or on) signals associated with modules located in the same chassis with the processor. In other words, with “local I/O”. We are not using Immediate Input or Immediate Output instructions in this simple discussion. **We are not using Jumps (to Labels) or Subroutines in this simple discussion**. In other words, we are trying to keep this discussion as simple as possible.

Mentally divide the processor SCAN into three separate steps.



Step 1 - **update the input image table**. In this step, the processor checks each input address in the following manner: If there IS a signal present at the input module screw, put a “1” in the bit associated (by its address) with that screw. If there is NOT a signal present at the input module screw, put a “0” in the bit associated (by its address) with that screw. Specific definition: input bits are bits whose addresses begin with the letter “I”.

Step 2 - **execute the ladder logic**. In this step, the processor executes the rungs one at a time, in order, going from top to bottom. If the instructions on a rung command the processor to turn ON a particular bit, then the processor will place a “1” in that bit. If the instructions on a rung command the processor to turn OFF a particular bit, then the processor will place a “0” in that bit. It does NOT matter whether the particular bit being controlled is an output bit - or an internal bit - or even (believe it or not) an input bit. The processor will write the commanded “1” or “0” instantly - before moving on to the next instruction. The processor scans ALL of the ladder rungs before it goes on to the next step.

Step 3 - **send the output image table to the output modules**. In this step, the processor takes the “1” or “0” contents of the output bits and sends this information

to the output modules. At this point, any real-world outputs which are marked with “1” are turned ON.
Any real-world outputs which are marked with “0” are turned OFF. Specific definition: output bits are bits whose addresses begin with the letter “O”.

Now the processor goes back to Step 1 - and starts the scan over again.

Elaborating on Step 2 - Suppose that the conditions on a rung at the top of the ladder command the processor to turn ON a particular output bit. But then the conditions on a rung at the bottom of the ladder command the processor to turn OFF the SAME output bit. (Normally this would be an error in programming). Question: will the real-world output (example: a lamp) be ON - or will it be OFF - or will the lamp FLICKER on and off very rapidly? Answer: the real-world output will be OFF. This is because the last rung which executed caused the processor to write a “0” into the output bit. And remember, the processor does not actually send the contents of the output table to the output modules until Step 3 - at the end of the scan. Yes, the processor DID change the contents of the output bit to a “1” when the first rung executed - but the output module had no way of knowing about that intermediate ON condition. Simply put: the processor controlled the bit - but the processor didn’t tell the output module about the bit’s ON/OFF status until after all of the ladder rungs were scanned.

Basic idea: a BIT is a box in the processor’s memory. This box can hold either a “1” or a “0”.

Basic idea: the INSTRUCTIONS in the ladder program are called “instructions” for a reason. They literally are instructions FROM (you) the programmer TO the processor. These instructions tell the processor to “examine a bit to see if it contains a 1” or to “examine a bit to see if it contains a 0” or to “go write a 1 or a 0” into a particular bit. The processor obeys these instructions instantly as soon as the rungs are executed. Specifically, the processor does exactly what the ladder commands it to do and “examines a bit” or “writes to a bit” instantly - whether the bit in question is an output bit (example: O:2/0) or an internal bit (example: B3/0) or even an input bit (example: I:1/0) makes no difference.

On the other hand, the processor only communicates with the input modules BEFORE any of the ladder rungs are executed. And further, the processor only communicates with the output modules AFTER all of the ladder rungs are executed.

2.1.3. IEC 61131-3 PROTOCOL

OVERVIEW

IEC 61131-3 is the first vendor independent standardized programming language for industrial automation. Established by the International Electrotechnical Commission (IEC) a worldwide standard organization founded in 1906 and recognized worldwide for standards in the controls industry by over 50 countries. The standard is already well established in Europe and is rapidly gaining popularity in North America and Asia as the programming standard for industrial and process control.

The adoption of IEC 61131-3 by the industry is driven by the increasing software complexity of control and automation requirements. The time to create, labor cost, and maintainability of control software has a major impact on control projects which can be improved using the IEC 61131-3 vendor independent programming language standard. Applying a standard programming language has a positive impact on the software life-cycle that includes requirements analysis, design, construction, testing (validation), installation, operation, and maintenance. The impact on maintenance is important since control software maintenance, including upgrades, is generally 2- 4 times the labor of initial programming.

The IEC 61131-3 standard combined with new powerful Freescale chip architectures enables an entire controller to be delivered in an embedded device. Control programs can run distributed and independently rather than concentrated in large controllers. No longer are thousands of lines of control programs required running in one controller for complex automation applications. This increases performance, improves reliability, and simplifies programs.

IEC 61131-3 provides multiple language support within a control program. The control program developer can select the language that is best suited to a particular task, greatly increasing their productivity. Plus with a standardized programming interface that is completely independent of the hardware platform, users can greatly reduce the cost of program maintenance and training across company wide automation applications.

IEC 61131-3 is hardware independent. The ability to transport automation solutions to other platforms is vastly improved over PLC applications offering users and System Integrators a level of reusability never before available. IEC 61131 increases the efficiency and speed of implementing new automation solutions by using readily available control components developed on other projects and by outside developers.
Companies that have chosen to implement IEC 61131-3 find that they reduce human resource costs in training, debugging and maintenance, and improve productivity from the higher reusability.

TECHNOLOGY OVERVIEW

IEC 61131-3 is the international standard for programmable controller programming languages. As such, it specifies the syntax, semantics and display for the following suite of PLC programming languages:

- Ladder diagram (LD)
- Sequential Function Charts (SFC)
- Function Block Diagram (FBD)
- Structured Text (ST)
- Instruction List (IL)

IEC 61131-3 is the third component (Part 3) of IEC 61131 family that consists of

- Part 1 General Overview
- Part 2 Hardware
- Part 3 Programming Languages
- Part 4 User Guidelines
- Part 5 Communication

The easiest way to view the standard is to split it into two parts, Common Elements and Programming Languages.

COMMON ELEMENTS

Data Typing

Data Typing is a common element of the standard with the purpose to prevent errors early on in development. It defines the type of parameters that will be used, and attempts to avoid errors like dividing a Date by an Integer. The different type of data supported are Boolean, Integer, Real, Byte, Word, Date, Time-of-Day and String. The Standard also allows users to define their own variables. These are known as derived data types. In this way an engineer would be able to define an analog input channel as a data type and re-use it over and over again.

Variables are assigned only to explicit hardware addresses or explicit inputs and outputs. These can be assigned in custom configurations and programs. An IEC 61131 system is highly independent and able to function with little to no messaging from an external network.

The Scope of the variable is limited to the organization unit in which they are declared. The great benefit of this feature is that their names can be reused in other

parts without any conflict, elimination of another source of errors. If the variables have Global Scope they can declared as global. Parameters can be assigned their initial value at start up and restart.

Configuration, Resources, and Tasks

At the highest level, the entire software required to solve a particular control problem can be formulated as a Configuration. A Configuration is specific to a particular type of control system, including the arrangement of the hardware, i.e. processing resources, memory addresses for I/O channels, and system capabilities.

Within a configuration one can define resources. A resource can be thought of as a processing facility that is able to execute IEC programs. Within a resource, one or more Tasks can be defined. Tasks control the execution of a set of programs and/or function blocks. These can either be executed periodically or upon occurrence of a specified trigger.

For instance, in an IEC 61131 enabled drive, a trigger could be set when RPMs fall below a predefined value. The trigger could start a task to increase speed. These results are instant and come directly from the drive. There is no lag or handshaking by an external PLC. This means that there is virtually no risk of losing a message or miscommunication. Feedback is nearly instantaneous compared to a Programmable Controller with an I/O and Program Scan time.

Programs are built from a number of different software elements written in any of the IEC defined languages; Ladder Diagrams, Sequential Function Charts, Function Block Diagrams, Structured Text or Instruction List. It is typical for a program to consist of a series of high level functions blocks written in one or more of these languages.

Program Organizations Units

Within IEC 61131-3, the programs, function blocks, and functions are called program organization units, or POUs.

IEC 61131-3 includes defined standard functions instances, ADD, ABS, SQRT, SIN, and COS. Or the user can create a custom function block and use that function block multiple times.

Function blocks are software objects that represent a level of more detailed control. They can contain data as well as an algorithm. As software objects they have a well-defined interface and hidden internals. This creates a clear line between the different levels of the programs. With these characteristics, functions, and function

blocks reflects best practices as embraced by object oriented programming principles

Function blocks can be written in any of the IEC languages in most cased even “C". Programs can be written using the any of the above mentioned basic building blocks.

Sequential Function Charts or SFCs are used to control the sequential behavior of a control program and support synchronization and concurrency.

PROGRAMMING LANGUAGES

Within IEC 61131-3, the syntax and semantics are defined for five standard programming languages, leaving no room for dialects. Once you have learned them, you can use a wide variety of systems based on this standard.

The end user is able to choose a programming language based on their knowledge, the problem at hand, external components, interfaces, or simple preference. All languages are linked and provided a common suite, with a link to existing experience. In this way they also provide a communication tool, combining people of different backgrounds. Because of the standards structure built on functions and function blocks users are able to adopt either a top-down or bottom-up strategy to develop their programs.

2.1.4. LADDER LOGIC

I will give information on the ladder logic in the following part which will be the logic on which this project is based on.
Also I will explain some ladder blocks and some simple ladder examples will be given.

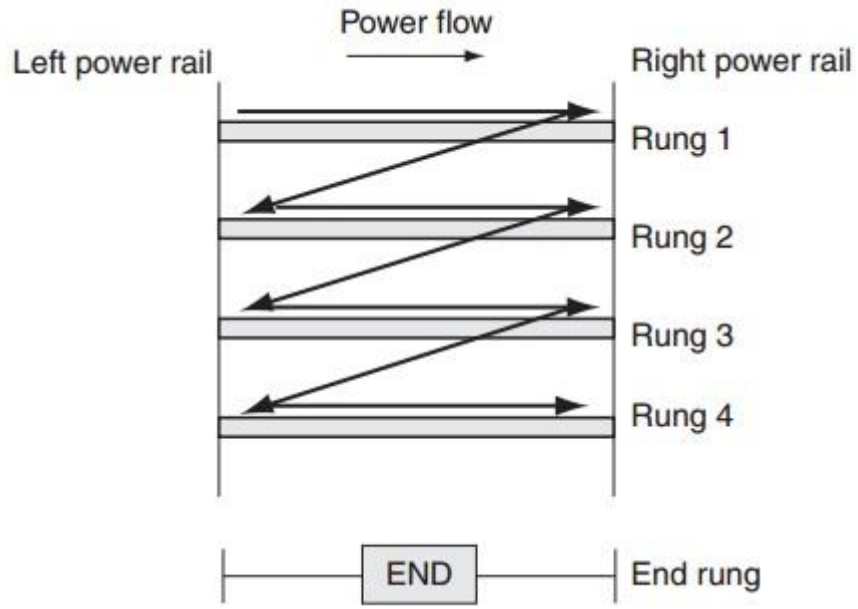
Ladder logic was originally a written method to document the design and construction of relay racks as used in manufacturing and process control. Each device in the relay rack would be represented by a symbol on the ladder diagram with connections between those devices shown. In addition, other items external to the relay rack such as pumps, heaters, and so forth would also be shown on the ladder diagram. Although the diagrams themselves have been used since the days when logic could only be implemented using switches and electromechanical relays, the term 'ladder logic' was only latterly adopted with the advent of solid state programmable logic.

Ladder logic has evolved into a programming language that represents a program by a graphical diagram based on the circuit diagrams of relay logic hardware. Ladder logic is used to develop software for programmable logic controllers (PLCs) used in industrial control applications. The name is based on the observation that programs in this language resemble ladders, with two vertical rails and a series of horizontal rungs between them. While ladder diagrams were once the only available notation for recording programmable controller programs, today other forms are standardized in IEC 61131-3. LD is widely used to program PLCs, where sequential control of a process or manufacturing operation is required. Ladder logic is useful for simple but critical control systems or for reworking old hardwired relay circuits. As programmable logic controllers became more sophisticated it has also been used in very complex automation systems. Often the ladder logic program is used in conjunction with an HMI operating on a computer workstation.

2.1.4.1 LADDER PROGRAMMING

A very commonly used method of programming PLCs is based on the use of ladder diagrams. Writing a program is then equivalent to drawing a switching circuit. The ladder diagram consists of two vertical lines representing the power rails. Circuits are connected as horizontal lines, i.e., the rungs of the ladder, between these two verticals.
In drawing a ladder diagram, certain conventions are adopted:

1. The vertical lines of the diagram represent the power rails between which circuits are connected. The power flow is assumed to be from the left-hand vertical across a rung.
2. Each rung on the ladder defines one operation in the control program.
3. A ladder diagram is read from left to right and from top to bottom, the figure bellow is showing the scanning motion employed by the PLC. The top rung is read from left to right. Then the second rung down is read from left to right and so on.



When the PLC is in its run mode, it goes through the entire ladder program to the end, the end rung of the program being clearly denoted, and then promptly resumes at the start. This procedure of going through all the rungs of the program is termed a cycle. The end rung might be indicated by a block with the word END or RET for return, since the program promptly returns to its beginning.

4. Each rung must start with an input or inputs and must end with at least one output. The term input is used for a control action, such as closing the

contacts of a switch, used as an input to the PLC. The term output is used for a device connected to the output of a PLC, e.g., a motor.

5. Electrical devices are shown in their normal condition. Thus a switch, which is normally open until some object closes it, is shown as open on the ladder diagram. A switch that is normally closed is shown closed.
6. A particular device can appear in more than one rung of a ladder. For example, we might have a relay that switches on one or more devices. The same letters and/or numbers are used to label the device in each situation.
7. The inputs and outputs are all identified by their addresses, the notation used depending on the PLC manufacturer. This is the address of the input or output in the memory of the PLC

2.1.4.2. LADDER BLOCKS

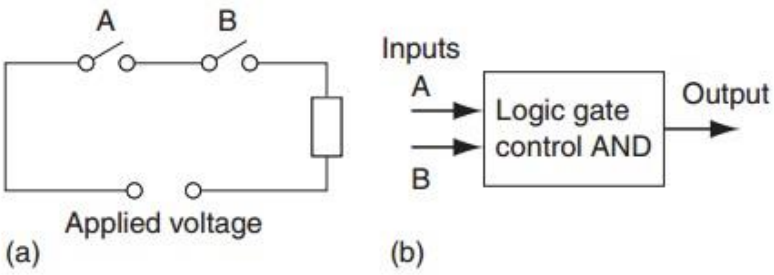
Logic Functions

There are many control situations requiring actions to be initiated when a certain combination of conditions is realized. Thus, for an automatic drilling machine, there might be the condition that the drill motor is to be activated when the limit switches are activated that indicate the presence of the workpiece and the drill position as being at the surface of the workpiece. Such a situation involves the AND logic function, condition A and condition B having both to be realized for an output to occur. This section is a consideration of such logic functions.

AND

The figure bellow shows a situation where an output is not energized unless two, normally open, switches are both closed. Switch A and switch B have both to be closed, which thus gives an AND logic situation. We can think of this as representing a control system with two inputs A and B. Only when A and B are both on is there an output. Thus if we use 1 to indicate an on signal and 0 to represent an off signal, then for there to be a 1 output we must have A and B both 1. Such an operation is said to be controlled by a logic gate and the relationship between the inputs to a logic gate and the outputs is tabulated in a form known as a truth table. Thus for the AND gate we have:

Inputs		Output
A	B	
0	0	0
0	1	0
1	0	0
1	1	1

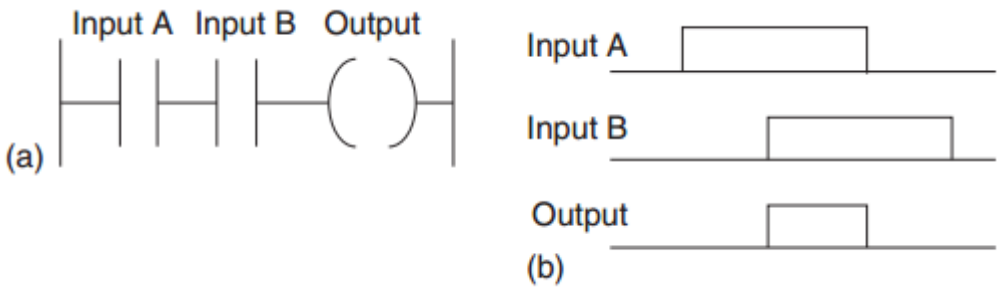


An example of an AND gate is an interlock control system for a machine tool so that it can only be operated when the safety guard is in position and the power switched on.

The figure bellow shows an AND gate system on a ladder diagram. The ladder diagram starts with ||, a normally open set of contacts labeled input A, to represent switch A and in series with it ||, another normally open set of contacts labeled input B, to represent switch B. The line then terminates with O to represent the output. For there to be an output, both input A and input B have to occur, i.e., input A and input B contacts have to be closed.

In general:

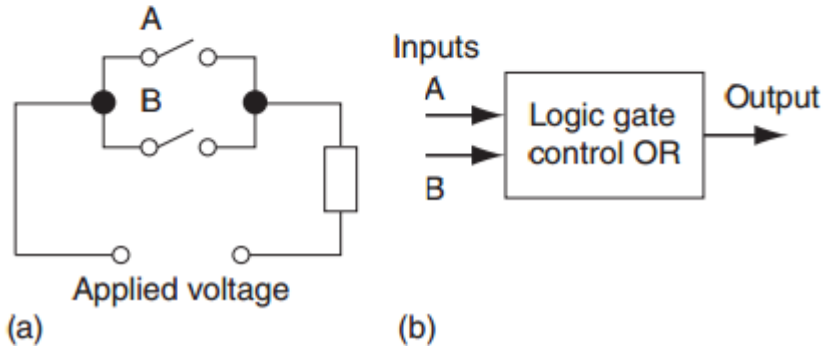
On a ladder diagram contacts in a horizontal rung, i.e., contacts in series, represent the logical AND operations.



OR

The figure bellow shows an electrical circuit where an output is energized when switch A or B, both normally open, are closed. This describes an OR logic gate in that input A or input B must be on for there to be an output. The truth table is:

Inputs		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	1

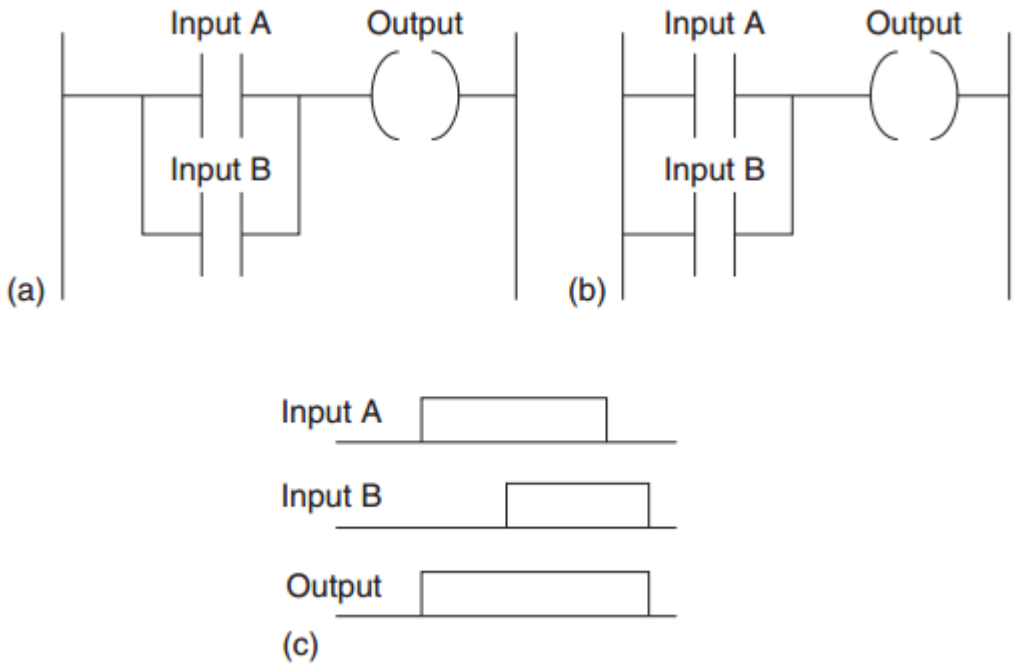


The 1st figure bellow (left) shows an OR logic gate system on a ladder diagram, while the 2nd figure (right) showing an equivalent alternative way of drawing the same diagram. The ladder diagram starts with ||, normally open contacts labeled input A, to represent switch A and in parallel with it ||, normally open contacts labeled input B, to represent switch B. Either input A or input B have to be closed for the output to be energized. The line then terminates with O to represent the output. In general:

Alternative paths provided by vertical paths from the main rung of a ladder diagram,i.e., paths in parallel represent logical OR operations.

An example of an OR gate control system is a conveyor belt transporting bottled products to packaging where a deflector plate is activated to deflect bottles into a

reject bin if either the weight is not within certain tolerances or there is no cap on the bottle



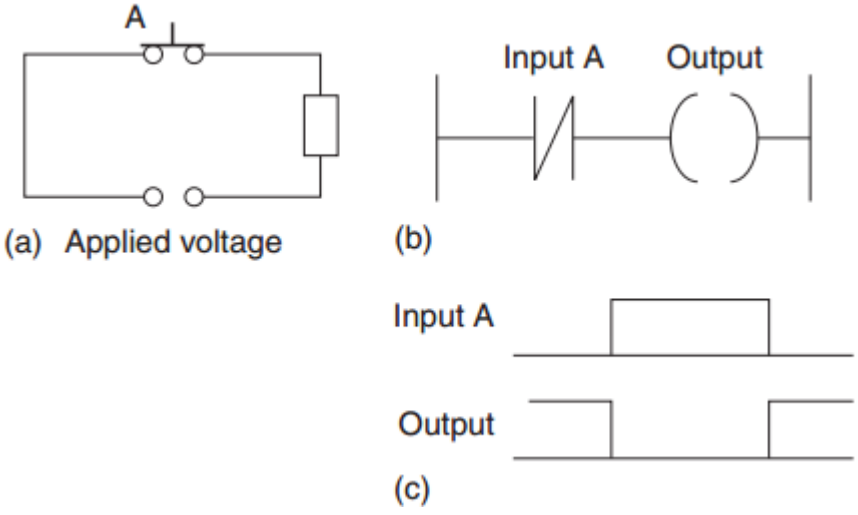
NOT (or known as Normally Closed contact)

The truth table is:

Input A	Output
0	1
1	0

The figure (a) bellow shows an electrical circuit controlled by a switch that is normally closed. When there is an input to the switch, it opens and there is then no current in the circuit. This illustrates a NOT gate in that there is an output when there is no input and no output when there is an input (figure c). The gate is sometimes referred to as an inverter.
The figure (b) shows a NOT gate system on a ladder diagram. The input A contacts are shown as being normally closed. This is in series with the output ( ). With no input to input A, the contacts are closed and so there is an output. When there is an input to input A, it opens and there is then no output.

An example of a NOT gate control system is a light that comes on when it becomes dark, i.e., when there is no light input to the light sensor there is an output.



As mentioned in the introduction as well there are more blocks such as counters , timers, Structure Text blocks,logic gates, etc. . . but in this project we are keeping it simple and break it down to use simple Rung programs with n/o , n/c contacts thus we feel there is no room for further explanation of all the remaining blocks.

2.2 MODELICA

2.2.1 Introduction to Modelica

In this section information on the modelica language will be given along with some of the functions of the language that have been used such as Algorithms, Equations , Events , Global variables etc.

There definitely is an interoperability problem amongst the large variety of modeling and simulation environments available today, and it gets more pressing every year with the trend towards ever more complex and heterogeneous systems to be simulated. The main cause of this problem is the absence of a state-of-the-art, standardized external model representation. Modeling languages, where employed, often do not adequately support the structuring of large, complex models and the process of model evolution in general. This support is usually provided by sophisticated graphical user interfaces - an approach which is capable of greatly improving the user's productivity, but at the price of specialization to a certain modeling formalism or application domain, or even uniqueness to a specific software package. It therefore is of no help with regard to the interoperability problem.

Among the research results in modeling and simulation, two concepts have strong relevance to this problem:

- *Object oriented modeling languages* already demonstrated how object oriented concepts can be successfully employed to support hierarchical structuring, reuse and evolution of large and complex models independent from the application domain and specialized graphical formalisms.
- *Non-causal modeling* demonstrated that the traditional simulation abstraction – the input/output block - can be generalized by relaxing the causality constraints, i.e., by not committing ports to an 'input' or 'output' role early, and that this generalization enables both more simple models and more efficient simulation while retaining the capability to include submodels with fixed input/output roles.

Examples of object-oriented and/or non-causal modeling languages and tools include: ASCEND, Dymola, gPROMS, NMF, ObjectMath, Omola, SIDOPS+, Smile, U.L.M., ALLAN, and VHDL-AMS.

The combined power of these concepts together with proven technology from existing modeling languages justifies a new attempt at introducing interoperability and openness to the world of modeling and simulation systems. Having started as an action within ESPRIT project "Simulation in Europe Basic Research Working Group (SiE-WG)" and currently operating as Technical Committee 1 within Eurosim and Technical Chapter on Modelica within Society for Computer Simulation International, a working group made up of simulation tool builders, users from different application domains, and computer scientists has made an attempt to unify the concepts and introduce a common modeling language.

This language, called ***Modelica***, is intended for modeling within many application domains (for example: electrical circuits, multi-body systems, drive trains, hydraulics, thermodynamical systems and chemical systems) and possibly using several formalisms (for example: ODE, DAE, bond graphs, finite state automata and Petri nets). Tools which might be general purpose or specialized to certain formalism and/or domain will store the models in the Modelica format in order to allow exchange of models among tools and among users. Much of the Modelica syntax will be hidden from the end-user because, in most cases, a graphical user interface will be used to build models by selecting icons for model components, using dialog boxes for parameter entry and connecting components graphically. The work started in the continuous time domain since there is a common mathematical framework in the form of differential-algebraic equations (DAE) and there are several existing modeling languages based on similar ideas. There is also significant experience of using these languages in various applications. It thus seems to be appropriate to collect all knowledge and experience and design a new unified modeling language or neutral format for model representation. The short range goal was to design a modeling language for differential-algebraic equation systems with some discrete event features to handle discontinuities and sampled systems. The design should be extendible in order that the goal can be expanded to design a multi-formalism, multi-domain, general-purpose modeling language. This is a report of the design state as of December 1998, Modelica version 1.1.

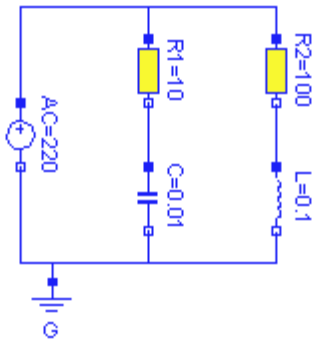
The object-oriented, non-causal modeling methodology and the corresponding standard model representation, Modelica, should be compared with at least four alternatives. Firstly, established commercial *general purpose simulation tools*, such as ACSL, EASY5, SIMULINK, System Build and others, are continually developed and Modelica will have to offer significant practical advantages with respect to these. Secondly, *special purpose simulation programs* for electronics (Spice, Saber, etc), multibody systems (ADAMS, DADS, SIMPACK, etc), chemical processes (ASPEN Plus, SpeedUp, etc) have specialized GUI and strong model libraries. However, they lack the multi-domain capabilities. Thirdly, many industrial simulation studies are still done without the use of any general purpose simulation tool, but rather relying on *numerical*
subroutine libraries and traditional programming languages. Based on experience with present tools, many users in this category frequently doubt that any general purpose method is capable of offering sufficient efficiency and robustness for their

application. Forthly, an IEEE supported *alternative language standardization effort* is underway: VHDL-AMS.

Most engineers and scientists recognize the advantages of an expressive and standardized modeling language. Unlike a decade ago, they are today ready to sacrifice reasonable amounts of short-term advantages for the benefit of abstract things like potential abundance of compatible tools, sound model architecture, and future availability of ready-made model libraries. In this respect, the time is ripe for a new standardization proposal. Another significant argument in favor of a new modeling language lies in recent achievements by present languages using a *noncausal* modeling paradigm. In the last decade, it has in several cases been proved that noncausal simulation techniques not only compare to, but outperform special purpose tools on applications that are far beyond the capability of established block oriented simulation tools. Examples exist in multi-body and mechatronics simulation, building simulation, and in chemical process plant simulation. A combination of modern numerical techniques and computer algebra methods give rise to this advantage. However, these non-causal modeling and simulation packages are not general enough, and exchange of models between different packages is not possible, i.e. a new unified language is needed.

2.2.2 MODELICA at a Glance

To give an introduction to Modelica we will consider modeling of a simple electrical circuit as shown below.



The system can be broken up into a set of connected electrical standard components. We have a voltage source, two resistors, an inductor, a capacitor and a ground point. Models of these components are typically available in model libraries and by using a graphical model editor we can define a model by drawing an object diagram very similar to the circuit diagram shown above by positioning icons that represent the models of the components and drawing connections.

A Modelica description of the complete circuit looks like

```
model circuit
Resistor R1(R=10);
Capacitor C(C=0.01);
Resistor R2(R=100);
Inductor L(L=0.1);
VsourceAC AC;
Ground G;
equation
connect (AC.p, R1.p); // Capacitor circuit
connect (R1.n, C.p);
connect (C.n, AC.n);
connect (R1.p, R2.p); // Inductor circuit
connect (R2.n, L.p);
connect (L.n, C.n);
connect (AC.n, G.p); // Ground
end circuit;
```

For clarity, the definition of the graphical layout of the composition diagram (here: electric circuit diagram) is not shown, although it is usually contained in a Modelica

model as annotations (which are not processed by a Modelica translator and only used by tools). A composite model of this type specifies the topology of the system to be modeled. It specifies the components and the connections between the components. The statement

```
Resistor R1(R=10);
```

declares a component ***R1*** to be of class ***Resistor*** and sets the default value of the resistance, R, to 10. The connections specify the interactions between the components. In other modeling languages connectors are referred as cuts, ports or terminals. The language element **connect** is a special operator that generates equations taking into account what kind of quantities that are involved as explained below.

The next step in introducing Modelica is to explain how library model classes are defined.

A connector must contain all quantities needed to describe the interaction. For electrical components we need the quantities voltage and current to define interaction via a wire. The types to represent them are declared as

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

where `Real` is the name of a predefined variable type. A real variable has a set of attributes such as unit of measure, initial value, minimum and maximum value. Here, the units of measure are set to be the SI units.
In Modelica, the basic structuring element is a **class**. There are seven *restricted* classes with specific names, such as **model**, **type** (a class which is an extension of built-in classes, such as **Real**, or of other defined types), **connector** (a class which does not have equations and can be used in connections). For a valid model it is fully equivalent to replace the **model**, **type**, and **connector** keywords by the keyword **class**, because the restrictions imposed by such a specialized class are fulfilled by a valid model.

The concept of restricted classes is advantageous because the modeller does not have to learn several different concepts, but just one: the class concept. All properties of a class, such as syntax and semantic of definition, instantiation, inheritance, genericity are identical to all kinds of restricted classes. Furthermore, the construction of Modelica translators is simplified considerably because only the syntax and semantic of a **class** has to be implemented along with some additional checks on restricted classes. The basic types, such as `Real` or `Integer` are builtin **type** classes, i.e., they have all the properties of a class and the attributes of these basic types are just parameters of the class.
There are two possibilities to define a class: The standard way is shown above for the definition of the electric circuit (**model** circuit). A short hand notation is possible, if a new class is identical to an existing one and only the default values of attributes are changed. The types above, such as *Voltage*, are declared in this way.

A connector class is defined as

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

A connection **connect** `(Pin1, Pin2)`, with `Pin1` and `Pin2` of connector class `Pin`, connects the two pins such that they form one node. This implies two equations, namely `Pin1.v = Pin2.v` and `Pin1.i + Pin2.i = 0`. The first equation indicates that the voltages on both branches connected together are the same, and the second corresponds to Kirchhoff's current law saying that the currents sum to zero at a node (assuming positive value while flowing into the component). The sum-to-zero equations are generated when the prefix **flow** is used. Similar laws apply to flow rates in a piping network and to forces and torques in mechanical systems.
When developing models and model libraries for a new application domain, it is good to start by defining a set of connector classes. A common set of connector classes used in all components in the library supports compatibility of the component models.
A common property of many electrical components is that they have two pins.

This means that it is useful to define an "interface" model class,

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

that has two pins, p and n, a quantity, v, that defines the voltage drop across the component and a quantity, i, that defines the current into the pin p, through the component and out from the pin n.
The equations define generic relations between quantities of a simple electrical component. In order to be useful a constitutive equation must be added. The keyword **partial** indicates that this model class is incomplete. The key word is optional. It is meant as an indication to a user that it is not possible to use the class as it is to instantiate components. Between the name of a class and its body it is allowed to have a string. It is treated as a comment attribute and is meant to be a documentation that tools may display in special ways.
To define a model for a resistor we exploit TwoPin and add a definition of parameter for the resistance and Ohm's law to define the behavior:

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

The keyword **parameter** specifies that the quantity is constant during a simulation run, but can change values between runs. A parameter is a quantity which makes it simple for a user to modify the behavior of a model.

A model for an electrical capacitor can also reuse the TwoPin as follows:

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

where `der(v)` means the time derivative of `v`. A model for the voltage source can be defined as

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

In order to provide not too much information at this stage, the constant PI is explicitly declared, although it is usually imported from the Modelica standard library. Finally, we must not forget the ground point.

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

The purpose of the ground model is twofold. First, it defines a reference value for the voltage levels. Secondly, the connections will generate one Kirchhoff's current law too many. The ground model handles this by introducing an extra current quantity p.i, which implicitly by the equations will be calculated to zero.

2.2.3 Algorithms

The basic describing mechanism of Modelica are *equations* and not assignment statements. This gives the needed flexibility, e.g., that a component description can be used with different causalities depending on how the component is connected. Still, in some situations it is more convenient to use assignment statements. For example, it might be more natural to define a digital controller with ordered assignment statements since the actual controller will be implemented in such a way.

It is possible to call external functions written in other programming languages from Modelica and to use all the power of these programming languages. This can be quite dangerous because many difficult-to-detect errors are possible which may lead to simulation failures. Therefore, this should only be done by the simulation specialist if tested legacy code is used or if a Modelica implementation is not feasible. In most cases, it is better to use a Modelica **algorithm** which is designed to be much more secure than calling external functions.

As an example let us write an algorithm to evaluate a polynomial...

```
algorithm
y := 0;
xpower := 1;
for i in 1:n+1 loop
y := y + a[i]*xpower;
xpower := xpower*x;
end for;
```

The vector xvec in the polynomial evaluator above had to be introduced in order that the number of unknowns are the same as the number of equations. Such a recursive calculation scheme is often more convenient to express as an algorithm, i.e., a sequence of assignment statements, ifstatements and loops, which allows multiple assignments:

A Modelica algorithm is a function in the *mathematical sense*, i.e. without internal memory and side-effects. That is, whenever such an algorithm is used with the same inputs, the result will be exactly the same. If a function is called during *continuous* integration this is an absolute prerequisite. Otherwise the mathematical assumptions on which the integration algorithms are based on, would be violated. An internal memory in an algorithm would lead to a model giving different results when using different integrators. With this restriction it is also possible to symbolically form the Jacobian by means of automatic differentiation. This requirement is also present for functions called only at **event** instants (see below). Otherwise, it would not be
possible to restart a simulation at any desired time instant, because the simulation environment does not know the actual value of the internal algorithm memory.

In the **algorithm** section, ordered assignment statements are present. To distinguish from equations in the **equation** sections, a special operator, :=, is used in assignments (i.e. given causality) in the **algorithm** section. Several assignments to the same variable can be performed in one algorithm section. Besides assignment statements, an algorithm may contain if-then-else expressions, if-then-else constructs (see below) and loops using the same syntax as in an equation-section.

Variables that appear on the left hand side of the assignment operator, which are conditionally assigned, are *initialized* to their start value (for algorithms in functions, the value given in the binding assignment) *whenever the algorithm is invoked.* Due to this feature it is impossible for a function to have a memory. Furthermore, it is guaranteed that the output variables always have a well-defined value.
Within an equation section of a class, algorithms are treated as a set of equations. Especially, algorithms are sorted together with all other equations. For the sorting process, the calling of a function with *n* output arguments is treated as *n* implicit equations, where **every** equation depends on all output and on all input arguments. This ensures that the implicit equations remain together during sorting (and can be replaced by the algorithm invocation afterwards), because the implicit equations of the function form one algebraic loop.

In addition to the for loop, there is a while loop which can be used within algorithms:

```
while condition loop
{ algorithm }
end while;
```

2.2.4 Functions

The polynomial evaluator above is a special input-output block since it does not have any states. Since it does not have any memory, it would be possible to invoke the polynomial function as a function, i.e. memory for variables are allocated temporarily while the algorithm of the function is executing. Modelica allows a specialization of a class called *function* which has only public inputs and outputs, one algorithm and no equations. The polynomial evaluation can thus be described as:

```
function PolynomialEvaluator2
input Real a[:];
input Real x;
output Real y;
protected
Real xpower;
algorithm
y := 0;
xpower := 1;
for i in 1:size(a, 1) loop
y := y + a[i]*xpower;
xpower := xpower*x;
end for;
end PolynomialEvaluator2;
```

A function declaration is similar to a class declaration but starts with the **function** keyword. The input arguments are marked with the keyword **input** (since the causality is input). The result argument of the function is marked with the keyword **output**.
No internal states are allowed, i.e., the der- and pre- operators are not allowed. Any class can be used as an input and output argument. All public, non-constant variables of a class in the output argument are the outputs of a function.

Instead of creating a polyeval object as was needed for the block PolynomialEvaluator:

```
PolynomialEvaluator polyeval(a={1, 2, 3, 4}, x=time, y=p);
```

it is possible to invoke the function as usual in an expression.

```
p = PolynomialEvaluator2(a={1, 2, 3, 4}, x=time);
```

It is also possible to invoke the function with positional association of the actual arguments:

```
p = PolynomialEvaluator2({1, 2, 3, 4}, time);
```

2.2.5 Events and Discrete Events

Events are ordered in time and form an event history

- A point in time that is instantaneous, i.e., has zero duration
- An event condition that switches from false to true in order for the event
- to take place
- A set of variables that are associated with the event, i.e. are referenced or explicitly changed by equations associated with the event
- Some behavior associated with the event, expressed as conditional equations that become active or are deactivated at the event.Instantaneous equations is a special case of conditional equations that are only active at events

Event creation – if

If-equations have the following syntax:

```
if expression then

{ equation ";" }
{ elseif expression then
{ equation ";" }
}
[ else
{ equation ";" }
] end if ";"
```

The expression of an if- or elseif-clause must be a scalar Boolean expression. One if-clause, and zero or more elseif-clauses, and an optional else-clause together form a list of branches. One or zero of the bodies of these if-, elseif- and else-clauses is selected, by evaluating the conditions of the if- and elseif-clauses sequentially until a condition that evaluates to true is found. If none of the conditions evaluate to true the body of the else-clause is selected (if an else-clause exists, otherwise no body is selected). In an equation section, the equations in the body are seen as equations that must be satisfied. The bodies that are not selected have no effect on that model evaluation.

If-equations in equation sections which do not have exclusively parameter expressions as switching conditions shall have an else-clause and each branch shall have the same number of equations. [*If this condition is violated, the single assignment rule would not hold, because the number of equations may change during simulation although the number of unknowns remains the same*].

Example:

```
model Diode "Ideal diode"
```

```
  extends TwoPin;
  Real s;
  Boolean off;
equation
   off = s < 0;
   if off then
     v=s
   else
     v=0;
   end if;
   i = if off then 0 else s;
end Diode;
```

Event creation – when

When-equations have the following syntax:

```
when expression then

{ equation ";" }
{ elsewhen expression then
{ equation ";" } }
end when ";"
```

The expression of a when-equation shall be a discrete-time Boolean scalar or vector expression. The statements within a when-equation are activated when the scalar expression or any of the elements of the vector expression becomes true.

Example:

The order between the equations in a when-equation does not matter, e.g.

```
equation
when x > 2 then
y3 = 2*x +y1+y2; // Order of y1 and y3 equations does not matter
y1 = sin(x);
```

Time Event

```
when time >= 10.0 then //Only dependent on time
   …
end when;
```

State Event

```
when sin(x) > 0.5 then // Related to a state.Check for zero-crossing.
     …
```

```
end when;
```

Synchronous Data-flow Principle and Single Assignment Rule

Modelica is based on the synchronous data flow principle and the single assignment rule, which are defined in the following way:

1. All variables keep their actual values until these values are explicitly changed. Variable values can be accessed at any time instant during continuous integration and at event instants.
2. At every time instant, during continuous integration and at event instants, the active equations express relations between variables which have to be fulfilled concurrently (equations are not active if the corresponding if-branch, when-clause or block in which the equation is present is not active).
3. Computation and communication at an event instant does not take time. [*If computation or communication time has to be simulated, this property has to be explicitly modeled*].
4. The total number of equations is identical to the total "number of unknown variables" (= single assignment rule).

Events and Synchronization

The integration is halted and an event occurs whenever a Real elementary relation, e.g. "x > 2", changes its value. The value of such a relation can only be changed at event instants [*in other words, Real elementary relations induce state or time events*]. The relation which triggered an event changes its value when evaluated literally before the model is processed at the event instant [*in other words, a root finding mechanism is needed which determines a small time interval in which the relation changes its value; the event occurs at the right side of this interval*]. Relations in the body of a when-clause are always taken literally. During continuous integration a Real elementary relation has the constant value of the relation from the last event instant.

Example:

```
y = if u > uMax then uMax else if u < uMin then uMin else u;
```

During continuous integration always the same if-branch is evaluated. The integration is halted whenever `u-uMax` *or* `u-uMin` *crosses zero. At the event instant, the correct if-branch is selected and the integration is restarted.*

It is a quality of implementation issue that the following special relations

```
time >= discrete expression
time < discrete expression
```

trigger a time event at "time = discrete expression", i.e., the event instant is known in advance and no iteration is needed to find the exact event instant.

Relations are taken literally also during continuous integration, if the relation or the expression in which the relation is present, are the argument of the `noEvent(..)` function.

The `smooth(p,x)` operator also allows relations used as argument to be taken literally. The `noEvent` feature is propagated to all subrelations in the scope of the noEvent function. For smooth the liberty to not allow literal evaluation is propagated to all subrelations, but the smooth-property itself is not propagated.

Example:

```
x = if noEvent(u > uMax) thenuMax elseifnoEvent(u in) thenMin else u;
y = noEvent( if u > uMax then uMax elseif u < uMin then uMin else u);
z = smooth(0, if u > uMax then uMax elseif u < uMin then uMin else u);
```

In this case x=y=z, but a tool might generate events for z. The if-expression is taken literally without inducing state events.

The `smooth` function is useful, if e.g. the modeler can guarantee that the used if-clauses fulfill at least the continuity requirement of integrators. In this case the simulation speed is improved, since no state event iterations occur during integration. The noEvent function is used to guard against "outside domain" errors, e.g. `y = if noEvent(x >= 0) then sqrt(x) else 0`.
All equations and assignment statements within when-clauses and all assignment statements within function classes are implicitly treated with the noEvent function, i.e., relations within the scope of these operators never induce state or time events. Using state events in when-clauses is unnecessary because the body of a when-clause is not evaluated during continuous integration.

Example:

```
 Limit1 = noEvent(x1 > 1);

 // Error since Limit1 is a discrete-time variable
 when noEvent(x1>1) or x2>10 then
 // error, when-conditions is not a discrete-time expression
 Close = true;
end when;
```

Modelica is based on the synchronous data flow principle

The rules for the synchronous data flow principle guarantee that variables are always defined by a unique set of equations. It is not possible that a variable is e.g. defined by two equations, which would give rise to conflicts or non-deterministic behavior. Furthermore, the continuous and the discrete parts of a model are always automatically "synchronized".

Example:

```
 equation // Illegal example

 when condition1 then
 close = true;
 end when;
 when condition2 then
 close = false;
end when;
```

This is not a valid model because rule 4 is violated since there are two equations for the single unknown variable close. If this would be a valid model, a conflict occurs when both conditions become true at the same time instant, since no priorities between the two equations are assigned. To become valid, the model has to be changed to:

```
 equation

 when condition1 then
 close = true;
 elsewhen condition2 then
 close = false;
 end when;
```

*Here, it is well-defined if both conditions become true at the same time instant (*condition1 *has a higher priority than* condition2*).*

There is no guarantee that two different events occur at the same time instant.

As a consequence, synchronization of events has to be explicitly programmed in the model, e.g. via counters.

Example:

```
 Boolean fastSample, slowSample;

 r ticks(start=0);
 Intege equation
 ple = sample(0,1);
 fastSam algorithm
 when fastSample then
ticks := if pre(ticks) < 5 then pre(ticks)+1 else 0;
 slowSample := pre(ticks) == 0;
 end when;
 algorithm
 when fastSample then // fast sampling
 ...
 end when;
 algorithm
 when slowSample then // slow sampling (5-times slower)
 .
 .. end when;
```

The `slowSample` when-clause is evaluated at every 5th occurrence of the fastSample when-clause.
The single assignment rule and the requirement to explicitly program the synchronization of events allow a certain degree of model verification already at compile time.

2.2.6 Classes and Packages

Assume we would like to connect two filters in series. Instead of repeating the filter equation, it is more convenient to make a definition of a filter once and create two instances. This is done by declaring a *class*. A class declaration contains a list of component declarations and a list of equations preceded by the keyword **equation**. An example of a low pass filter class is shown below.

```
class LowPassFilter
  parameter Real T=1;
  Real u, y(start=1);

equation
  T*der(y) + y = u;
end LowPassFilter;
```

The model class can be used to create two instances of the filter with different time constants and "connecting" them together as follows

```
class FiltersInSeries
  LowPassFilter F1(T=2), F2(T=3);

equation
  F1.u = sin(time);
  F2.u = F1.y;
end FiltersInSeries;
```

In this case we have used a *modification* to modify the time constant of the filters to T=2 and T=3 respectively from the default value T=1 given in the low-pass filter class. Dot notation is used to reference components, like u, within structured components, like F1. For the moment it can be assumed that all components can be reached by dot-notation. Restrictions of accessibility will be introduced later. The independent variable is referenced as **time**.

If the FiltersInSeries model is used to declare components at a higher hierarchical level, it is still possible to modify the time constants by using a hierarchical *modification:*

```
model ModifiedFiltersInSeries
  FiltersInSeries F12(F1(T=6), F2(T=11));
end ModifiedFiltersInSeries;
```

The class concept is similar as in programming languages. It is used for many purposes in Modelica, such as model components, connection mechanisms, parameter sets, input-output blocks and functions. In order to make Modelica classes easier to read and to maintain, special keywords have been introduced for such special uses, **model**, **connector**, **record**, **block**, **type** and **package**. It should be noted though that the use of these keywords only apply certain restrictions, like records are not allowed to contain equations. However, for a valid model, the replacement of these keywords by **class** would give exactly the same model behavior. In the following description we will use the specialized keywords in order to convey their meaning.

Packages

Class declarations may be nested. One use of that is maintenance of the name space for classes, i.e., to avoid name clashes, by storing a set of related classes within an enclosing class. There is a special kind of class for that, called **package**. A package may only contain declarations of constants and classes. Dot-notation is used to refer to the inner class. Examples of packages are given in the appendix of the Language Specification, where the Modelica standard package is described which is always available for a Modelica translator.

Modelica models are structured in hierarchical libraries (package)

```
package Modelica
    package Mechanics
      package Rotational
        model Inertia     ← Modelica.Mechanics.Rotational.Inertia
          ...
        end Inertia;
        model Torque
          ...
        end Torque;
        ...
      end Rotational;
    end Mechanics;
    ...
end Modelica;
```

The first part of a hierarchical name is searched from "lower" to "upper" hierarchies within a package:

```
package Modelica
    package Mechanics
      package Rotational
        package Intefaces
          connector Flange_a
            ...
          end Flange_a;
```

```
        end Interfaces
        model Inertia
          Interfaces.Flange_a flange_a;
          Modelica.Mechanics.Rotational.Interfaces.Flange_a a;
        end Inertia;
        model Torque
          ...
        end Torque;
        ...
      end Rotational;
    end Mechanics;
    ...
end Modelica;
```

Note that: **package** Intefaces, Interfaces.Flange_a flange_a , Modelica.Mechanics.Rotational.Interfaces.Flange_a are identical definitions.

2.2.7 Connections

We have seen how classes can be used to build-up hierarchical models. It will now be shown how to define physical connections by means of a restricted class called **connector**.

We will study modeling of a simple electrical circuit. The first issue is then how to represent pins and connections. Each pin is characterized by two variables, voltage and current. A first attempt would be to use a connector as follows.

```
connector Pin
 Real v, i;
end Pin;
```

and build a resistor with two pins p and n like

```
model Resistor
   Pin p, n; // "Positive" and "negative" pins.
   parameter Real R "Resistance";

equation
   R*p.i = p.v - n.v;
   n.i = p.i; // Assume both n.i and p.i to be positive
              // when current flows from p to n.
end Resistor;
```

A descriptive text string enclosed in " " can be associated with a component like R. A comment which is completely ignored can be entered after //. Everything until the end of the line is then ignored. Larger comments can be enclosed in /* */.

A simple circuit with series connections of two resistors would then be described as:

```
model FirstCircuit
  Resistor R1(R=100), R2(R=200);
equation
  R1.n = R2.p;
end FirstCircuit;
```

The equation R1.n = R2.p represents the connection of pin n of R1 to pin p of R2. The semantics of this equation on structured components is the same as

```
R1.n.v = R2.p.v
R1.n.i = R2.p.i
```

This describes the series connection correctly because only two components were connected.Some mechanism is needed to handle Kirchhoff's current law, i.e. that the currents of all wires connected at a node are summed to zero. Similar laws apply to flows in a piping network and to forces and torques in mechanical systems. The default rule is that connected variables are set equal. Such variables are called *across* variables. Real variables that should be summed to zero are declared with prefix **flow**. Such variables are also called *through* variables. In Modelica we assume that such variables are positive when the flow (or corresponding vector) is into the component.

```
connector Pin
  Real v;
  flow Real i;
end Pin;
```

It is useful to introduce *units* in order to enhance the possibility to generate diagnostics based on redundant information. Modelica allows deriving new classes with certain modified attributes. The keyword **type** is used to define a new class, which is derived from the built-in data types or defined records. Defining Voltage and Current as modifications of Real with other attributes and a corresponding Pin can thus be made as follows:

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

connector Pin
  Voltage v;
  flow Current i;
end Pin;
model Resistor
  Pin p, n; // "Positive" and "negative" pins.
  parameter Real R(unit="Ohm") "Resistance";

equation
  R*p.i = p.v - n.v;
  p.i + n.i = 0; // Positive currents into component.

end Resistor;
```

We are now able to correctly connect three components at one node.

```
model SimpleCircuit
  Resistor R1(R=100), R2(R=200), R3(R=300);

equation
  connect(R1.p, R2.p);
  connect(R1.p, R3.p);

end SimpleCircuit;
```

connect is a special operator that generates equations taking into account what kind of variables that are involved. The equations are in this case equivalent to

```
R1.p.v = R2.p.v;
R1.p.v = R3.p.v;
R1.p.i + R2.p.i + R3.p.i = 0;
```

In certain cases, a model library might be built on the assumption that only one connection can be made to each connector. There is a built-in function **cardinality**(c) that returns the number of connections that has been made to a connector c. It is also possible to get information about the direction of a connection by using the built-in function **direction**(c) (provided cardinality(c) == 1). For a connection, connect(c1, c2), direction(c1) returns -1 and direction(c2) returns 1.

Chapter 3

Approaches to the problem

During the process of representing LD as a Modelica model assembled visually (i.e. not by translating LD logic into a Modelica algorithm) we came across various difficulties mostly having to deal with the Modelica environment – language interpretation, in other words “errors” coming from the compiler. Some of them actually were unexpected so we were forced to follow other paths.
In this section I will show our 2 main approaches and the reason why we chose the 1st one and why we had to proceed to the 2nd one.

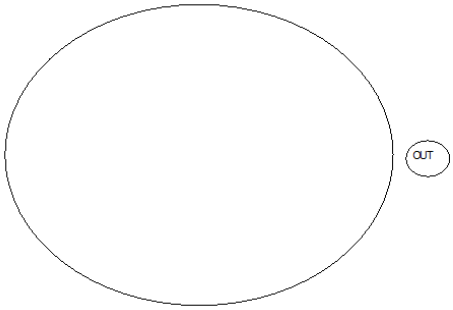
3.1 Approach No1

In the first approach we wanted to ensure that the token is being passed from component to component without any hierarchy violations and we have achieved that.

Components used in 1st approach

1. **Starter**
2. **SeqBlock**
3. **Coil**
4. **Connectors In – Out**

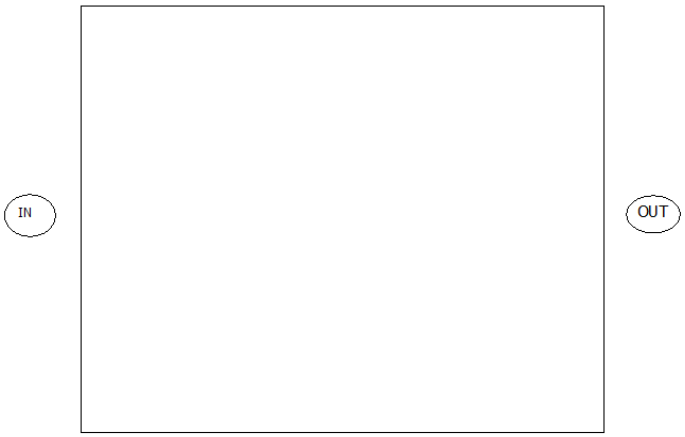
Starter




```
model Starter2014
parameter Real Tc = 0.1;
algorithm
when sample(0, Tc) and ActiveRung == MyRungNo then
OUT.clk:=not OUT.clk;
end when;
OUT.datum:=0;
end Starter2014;
```

The starter is the clock we have used in the start of the rung to give a “pulse” and start the program.

SeqBlock

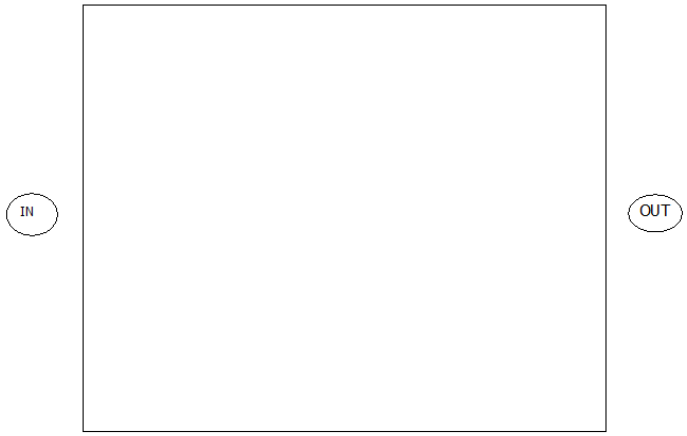



```
model SeqBlock
algorithm
```

```
when IN.clk and not pre(IN.clk) or not IN.clk and pre(IN.clk) then
OUT.clk:=not OUT.clk;
end when;
OUT.datum:=IN.datum + 1;
end SeqBlock;
```

The seq block is the so called normally closed contact that takes the token from the left side and sends it out from the right side as an output.

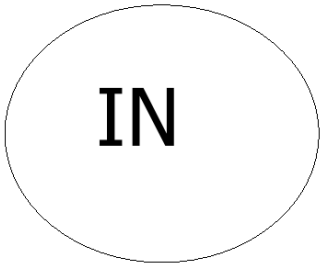
Coil




```
model coil
algorithm
when IN.clk and not pre(IN.clk) or not IN.clk and pre(IN.clk) then
OUT.clk:=not OUT.clk;
end when;
OUT.datum:=IN.datum + 1;
end coil;
```

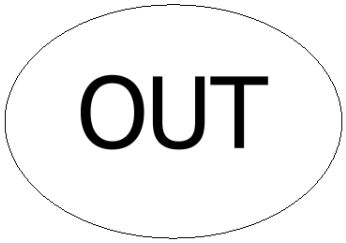
The Coil is the well-known coil of the LD and is the end of the program that gives the output of the rung.

Connectors In – Out




```
connector SeqPortIn
input Boolean clk;
input Integer datum;
end SeqPortIn;
```

Is the input of every component of type Connector getting the token and the pulse.

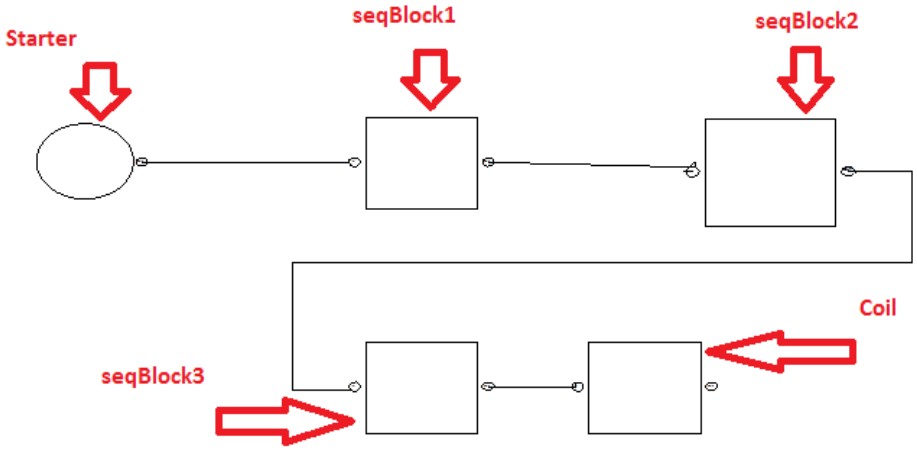


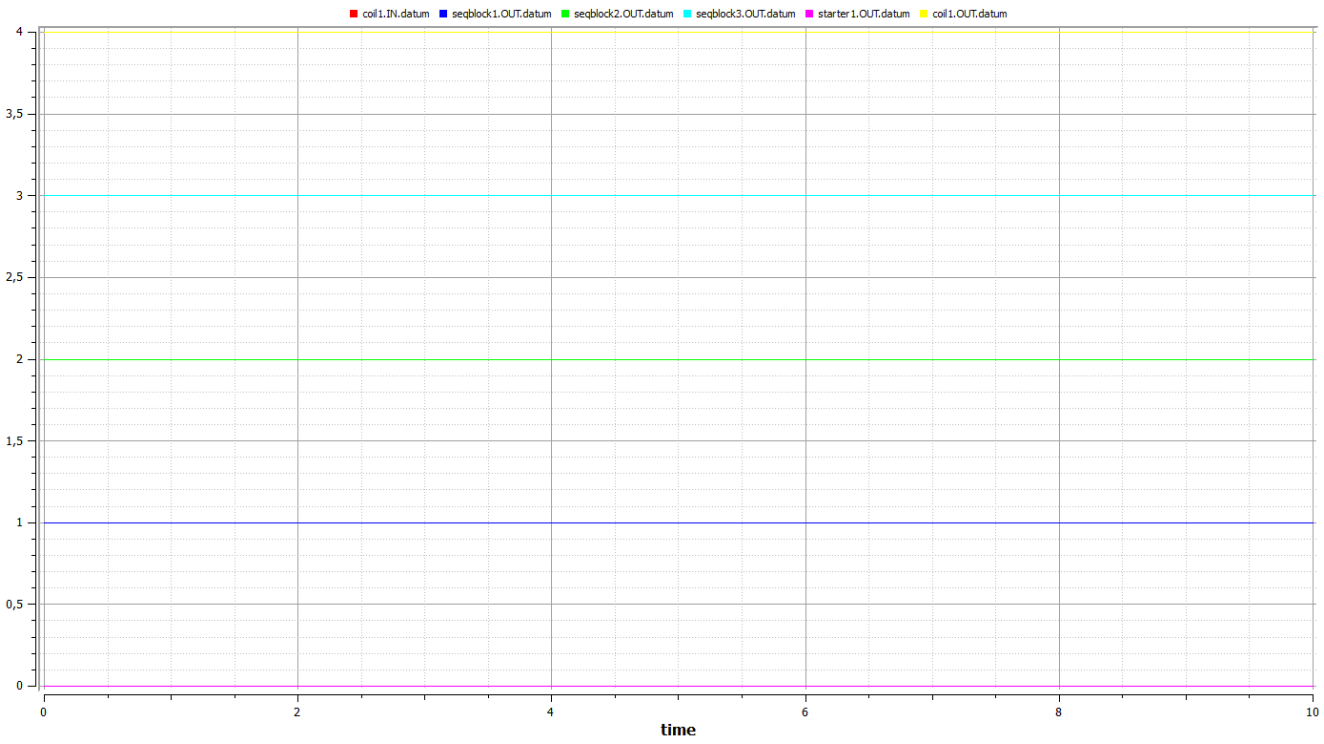

```
connector SeqPortOut
output Boolean clk;
output Integer datum;
end SeqPortOut;
```

Is the output of every component of type Connector passing the token and the pulse.

Example and Simulation Result

In this example we have simple put in series a starter (type Clock) with 3 seqBlocks(NC contacts) and 1 Coil. The starter will pass the token on the 1st positive edge and then each block will pass the token to the next block connected on the RIGHT side till we reach the Coil where we stop and we have the final output.





Starter = 0 because it was preset to be 0. (`OUT.datum:=0;`)
SeqBlock1 = 1 because (`OUT.datum:=IN.datum + 1`)
SeqBlock2 = 2 because (`OUT.datum:=IN.datum + 1`)
SeqBlock3 = 3 because (`OUT.datum:=IN.datum + 1`)
Coil = 4 because (`OUT.datum:=IN.datum + 1`)

Clarifications:
datum : is the so called token that is passed from component to component (variable of type Integer)
clk : is the clock's bit (the edge)
SeqBlock: is the so called Normally Closed Contact
For the time being we are using algorithm with no problems.

So with the first simulation example our idea works and we can move onto the second where we want to add Rungs into play and start scanning them.

3.2 Approach No2

So in this approach we will use the token passing idea for a single Rung and try to go deeper , using more than 1 rung and introduce a jumpCoil also (this coil will tell the program to jump onto a specific rung and not the next one).

Here we are introducing the inner/outer clause having to do with global variables. We needed 1 global variable to be updated every time we meet a coil and become +1 so the starter of the next Rung will be activated through the command

```
when sample(0, Tc) and ActiveRung == MyRungNo then
```

The inner command is put into the model of the whole example – program with the command

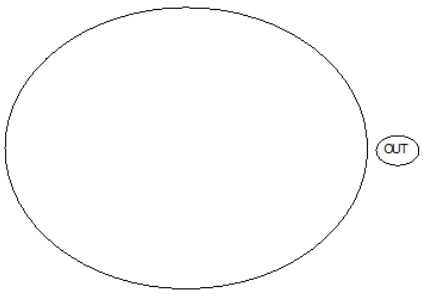
```
inner Integer ActiveRung(start = 1);
```

This means that we are defining ActiveRung to be = 1 so that the starter on the 1st rung can be activated and the process to start.

Components used in 2nd approach

1. **Starter**
2. **NCcontact**
3. **Coil**
4. **Pre Block**
5. **JumpCoil**
6. **Connectors In – Out**

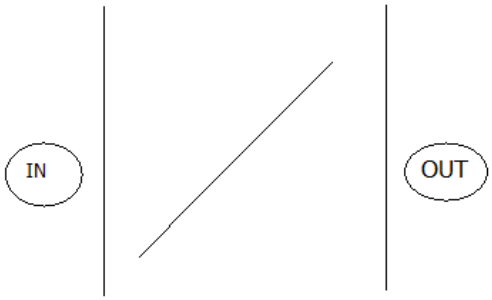
Starter




```
model Starter_test_new
parameter Real Tc = 0.1;
parameter Integer MyRungNo = 1;
outer Integer ActiveRung;
algorithm
when sample(0, Tc) and ActiveRung == MyRungNo then
OUT.clk:=not OUT.clk;
end when;
OUT.datum:=0;
end Starter_test_new;
```

Explanation:
The start works like the previous one but here we have an extra condition.We want the starter – clock to work ONLY when the Number of the rung that the start is connect is equal the our ActiveRung Number.

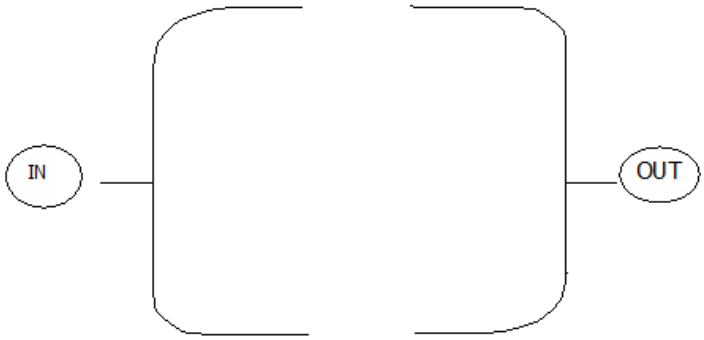
NCcontact




```
model NCcontact
algorithm
when IN.clk and not pre(IN.clk) or not IN.clk and pre(IN.clk) then
OUT.clk:=not OUT.clk;
end when;
OUT.datum:=IN.datum + 1;
end NCcontact;
```

The Closed Contact block works like before.

Coil




```
model coil
outer Integer ActiveRung;
algorithm
when IN.clk and not pre(IN.clk) or not IN.clk and pre(IN.clk) then
ActiveRung:=pre(ActiveRung) + 1;
end when;
OUT.datum:=IN.datum + 1;
end coil;
```

The coil works like before but here we have the extra condition where we are updating the global variable ActiveRung and make it ActiveRung +1.

Note: you can notice from the components so far the we are using the prefix (pre) in front of some random places and they seem to not make sense or lead you to think of the word *previous.*
The truth is that none of them is correct. The reason why we are doing this is because when you connect Booleans in series and you close the loop the solution is not solvable hence you need to break the loop in at least one place.
For that we are putting the pre.

You can either write it in the code or connect the Pre block in your system.

Pre Block

Modelica.Blocks.Logical.Pre

Breaks algebraic loops by an infinitesimal small time delay (y = pre(u): event iteration continues until u = pre(u))

This block delays the Boolean input by an infinitesimal small time delay and therefore breaks algebraic loops. In a network of logical blocks, in every "closed connection loop" at least one logical block must have a delay, since algebraic systems of Boolean equations are not solveable.

The "Pre" block returns the value of the "input" signal from the last "event iteration". The "event iteration" stops, once both values are identical (u = pre(u)).

Extends from **Blocks.Interfaces.partialBooleanSISO** (Partial block with 1 input and 1 output Boolean signal).

Parameters

Type	Name	Default	Description
Boolean	pre_u_start	false	Start value of pre(u) at initial time

Connectors

Type	Name	Description
input BooleanInput	u	Connector of Boolean input signal
output BooleanOutput	y	Connector of Boolean output signal

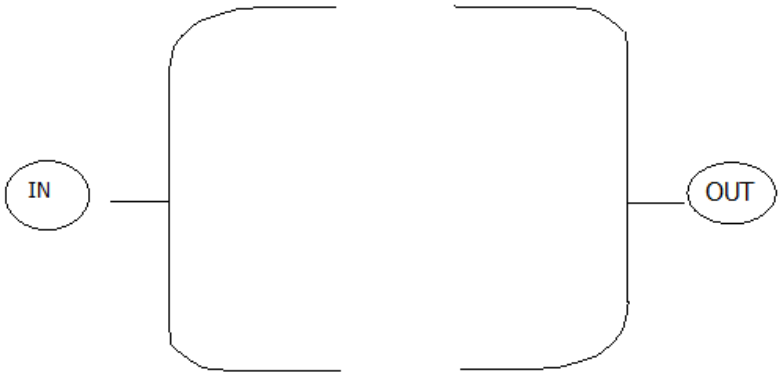
Modelica definition

```
model Pre
  "Breaks algebraic loops by an infinitesimal small time delay (y = pre(u): event iteration
continues until u = pre(u))"

  parameter Boolean pre_u_start = false "Start value of pre(u) at initial time";
  extends Blocks.Interfaces.partialBooleanSISO;

initial equation
  pre(u) = pre_u_start;
equation
  y = pre(u);
end Pre;
```

JumpCoil



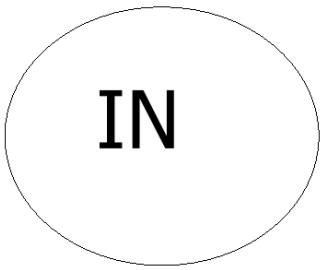

```
model jumpcoil
outer Integer ActiveRung;
parameter Integer WhereToJump = 1;
algorithm
when IN.clk and not pre(IN.clk) or not IN.clk and pre(IN.clk) then
ActiveRung:=WhereToJump;
end when;
OUT.datum := IN.datum + 1;
end jumpcoil;
```

This Coil works exactly like the other one but with the difference in the line

```
ActiveRung:=WhereToJump;
```

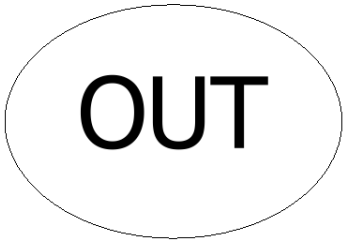
This means that we will not Jump onto the next Rung but we will *Jump* where the parameter *WheretoJump* says.

Connectors In – Out



```
connector SeqPortIn
input Boolean clk;
input Integer datum;
end SeqPortIn;
```

Connector In is identical to the previous one.

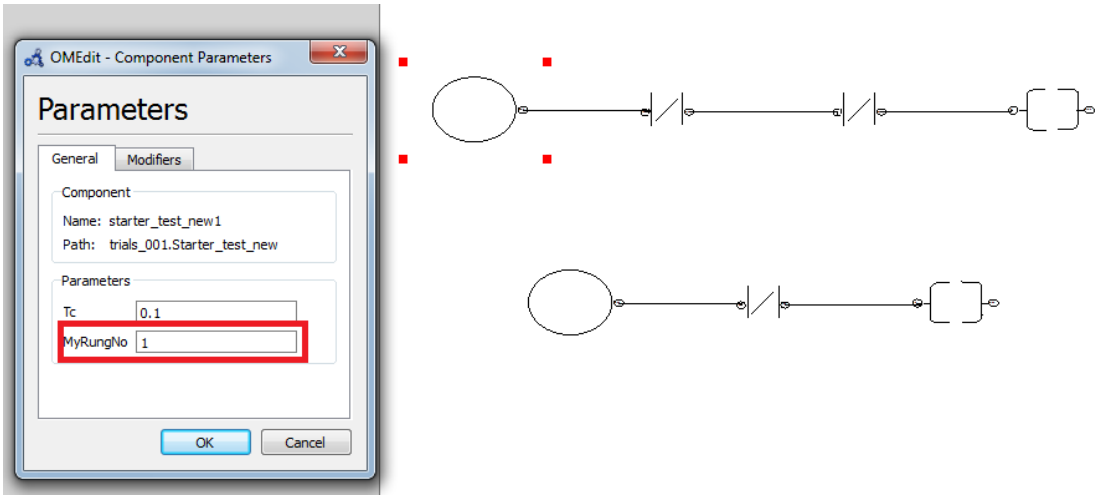



```
connector SeqPortOut
output Boolean clk;
output Integer datum;
end SeqPortOut;
```

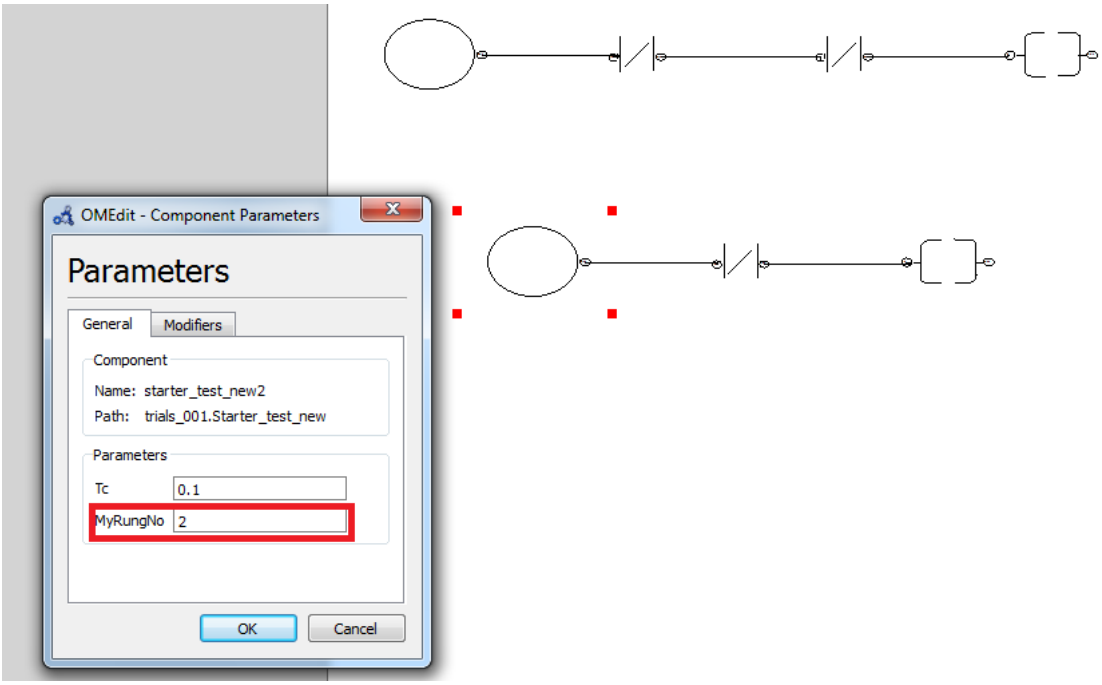
Connector Out is identical to the previous one.

Manual numbering of the Rungs

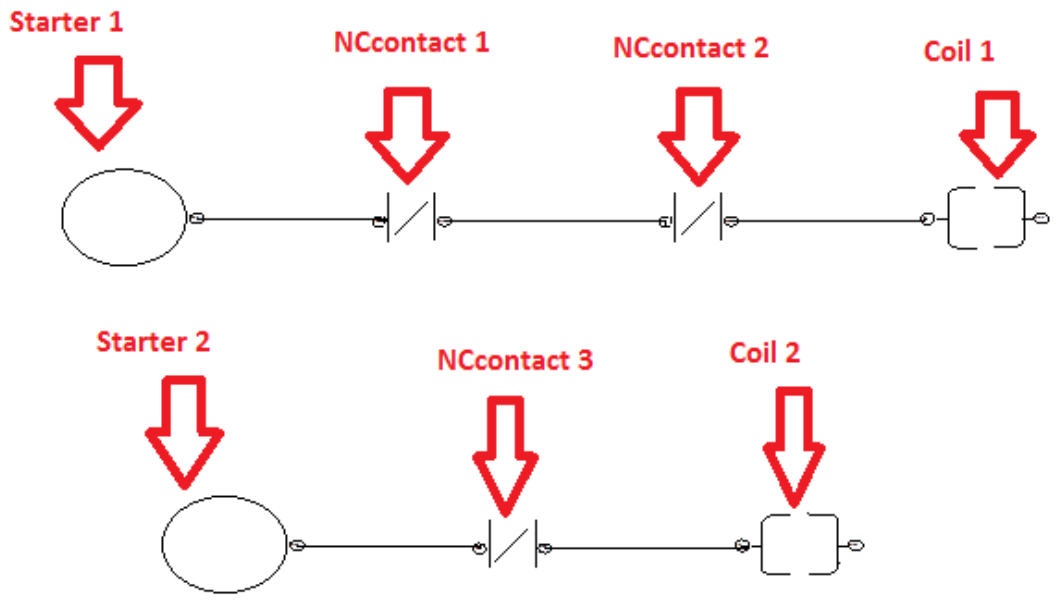
In Modelica you can set the parameters on a block by clicking on them. Change their number or start (true,false).



And for the second starter we set the RungNo to 2.



Example and Simulation Result



This simple example was expected to run smoothly by passing the token from Starter 1 up to Coil 1.
Then the ActiveRung would be updated to ActiveRung + 1 = 1 + 1 = 2.

So since we had already set MyRungNo = 2 on starter 2, as soon as ActiveRung becomes 2 ➔ Starter 1 stops working ➔ Starter 2 is activated and the token is passed up to Coil2 where the ActiveRung becomes = 3 and the program stops. However we have an Unsolvable for the time error on the compiler where it would see more variables than needed. We have tried various ways to solve this problem and one of them was Approach No3.

3.3 Approach No3

In this approach we have tried to tell the program the state (true or false) of every component and also set the opposite state meaning that we would say that

A_trigger = true;
A_trigger = false;

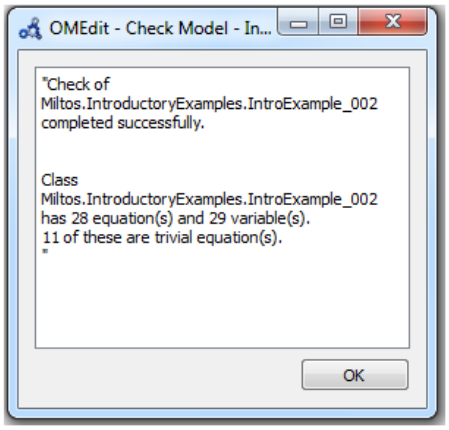
Trigger stands for input and **Done** stands for output.

```
model IntroExample_002
parameter Real Tc = 0.1;
discrete Integer ActiveRung;
discrete Boolean A_trigger;
discrete Boolean A_in;
discrete Boolean A_out;
discrete Boolean A_done;
discrete Boolean B_trigger;
discrete Boolean B_in;
discrete Boolean B_out;
discrete Boolean B_done;
discrete Boolean coilX_trigger;
discrete Boolean coilX_in;
discrete Boolean coilX_out;
discrete Boolean coilX_done;
discrete Boolean X_trigger;
discrete Boolean X_in;
discrete Boolean X_out;
discrete Boolean X_done;
discrete Boolean C_trigger;
discrete Boolean C_in;
discrete Boolean C_out;
discrete Boolean C_done;
discrete Boolean coilY_trigger;
discrete Boolean coilY_in;
discrete Boolean coilY_out;
discrete Boolean coilY_done;
discrete Boolean iA,iB,iX,iC;
initial algorithm
ActiveRung:=1;
A_trigger:=false;
B_trigger:=false;
coilX_trigger:=false;
X_trigger:=false;
C_trigger:=false;
coilY_trigger:=false;
```

```
algorithm
when edge(A_trigger) then
A_out:=A_in and iA;
A_done:=true;
end when;
//A_trigger := false;
when edge(B_trigger) then
B_out:=B_in and not iB;
B_done:=true;
end when;
//B_trigger := false;
when edge(coilX_trigger) then
coilX_out:=X_in;
coilX_done:=true;
end when;
//coilX_trigger := false;
when edge(X_trigger) then
X_out:=X_in and iX;
X_done:=true;
end when;
//X_trigger := false;
when edge(C_trigger) then
C_out:=B_in and iC;
C_done:=true;
end when;
//C_trigger := false;
when edge(coilY_trigger) then
coilY_out:=X_in;
coilY_done:=true;
end when;
//coilY_trigger := false;
// Start the exploration of the whole LD
when sample(0, Tc) then
iA:=not pre(iA);
iB:=if time < 0.5 then true else false;
iC:=if time > 1 then false else true;
ActiveRung:=1;
A_done:=false;
B_done:=false;
coilX_done:=false;
X_done:=false;
C_done:=false;
coilY_done:=false;
end when;
// Generate inputs
when ActiveRung == 1 then
A_trigger:=true;
end when;
when ActiveRung == 1 and coilX_done then
ActiveRung:=2;
end when;
when ActiveRung == 2 and coilY_done then
ActiveRung:=1;
end when;
equation
B_trigger = A_done;
coilX_trigger = B_done;
C_trigger = X_done;
coilY_trigger = C_done;
A_in = true;
B_in = A_out;
```

```
coilX_in = B_out;
X_in = true;
iX = coilX_out;
C_in = X_out;
coilY_in = C_out;
end IntroExample_002;
```

Sadly the error here was of the same nature:



Translation	18:40:00	0:0-0:0	Internal error Transformation Module PFPlusExt index Reduction Method Pantelides failed!
Symbolic	18:40:00	0:0-0:0	Too few equations, under-determined system. The model has 21 equation(s) and 22 variable(s).

So after noticing that the problem comes from the usage of algorithms we have decided to give it a go by stop using algorithms and solve the problem by using Boolean equations only.
And this brings us to Approach No4.

Chapter 4

The Adopted Approach

First, let us spend a few more words, also in the light of the previously presented attempts, on why representing a program in a modeling language is nontrivial.

The difference between modeling and programming is the difference between saying "what" and saying "how". This is not an absolute binary distinction, but is rather a spectrum. However, a modeling language must be more about "what" than about "how". In this sense models are declarative (in the sense of being "statements of what is needed", not in the sense of being opposite of imperative). We may also say that models are specifications. The fact that some models can be directly executed does not mean that they are code.

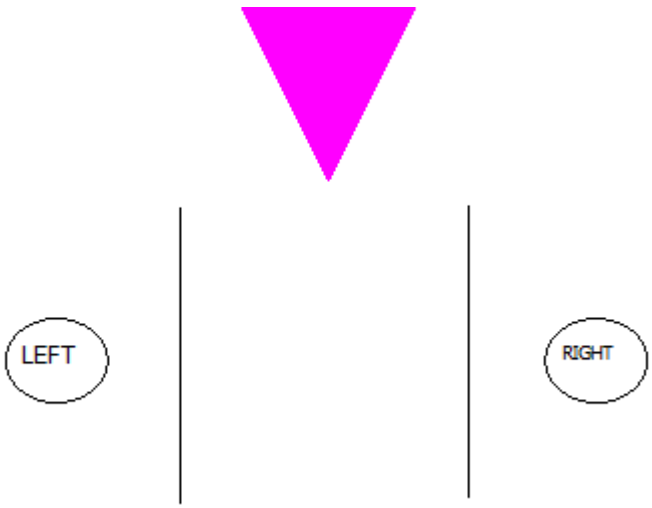
Therefore it is clear and obvious that when someone is trying to program using a modeling language it is like trying to explain “what” using “how” tools.

In theory and on paper someone can still reason and write a code or a sequence that makes sense and should work without any problems (like in the programming languages).
However when someone tries to put these things into a modeling language , errors will start to pop up and here comes the nontrivial relationship of programming with a modeling language.
As a consequence the models are structured in such a way that we will have no collisions with the modeling language.

Components used in the adopted approach

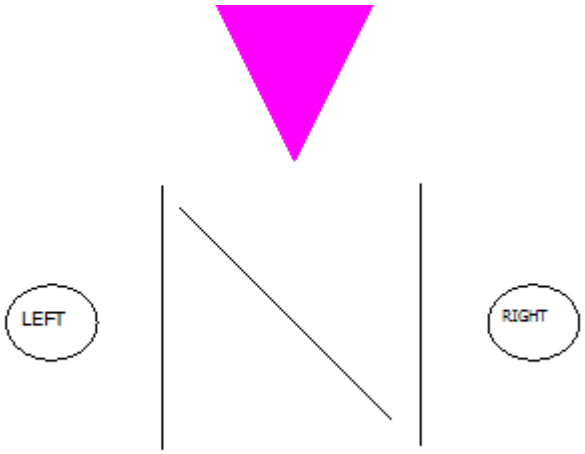
1. SupplyRail
2. GroundRail
3. NCcontact
4. Coil
5. Connectors Cleft-Cright-Rail_Cright-Rail_Cleft

NOcontact




```
model Contact_NO
Boolean compute_out;
initial equation
pre(compute_out) = CL.R2L and not CR.L2R;
equation
compute_out = pre(compute_out);
CL.L2R = compute_out;
CR.R2L = compute_out;
CR.s = CL.s and bit;
end Contact_NO;
```

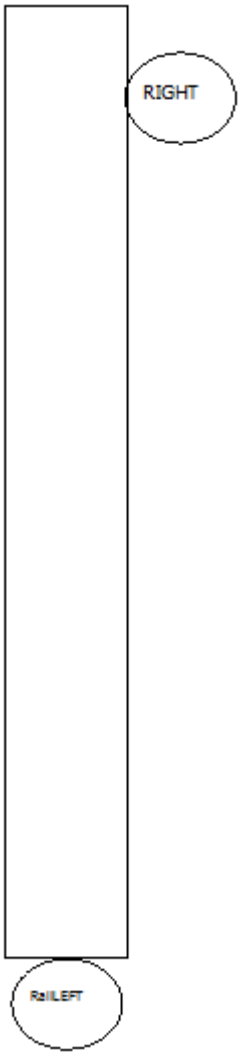
NCcontact




```
model Contact_NC
Boolean compute_out;
initial equation
pre(compute_out) = CL.R2L and not CR.L2R;
equation
compute_out = pre(compute_out);
CL.L2R = compute_out;
CR.R2L = compute_out;
CR.s = CL.s and not bit;
end Contact_NC;
```

For the NO and NC contacts we have an extra input for the Boolean Pulse block that we use from the standard modelica library in place of the previously used Starter. What the contacts do is to pass the token (bit) from left to right and preserve the sequential balance.

SupplyRail

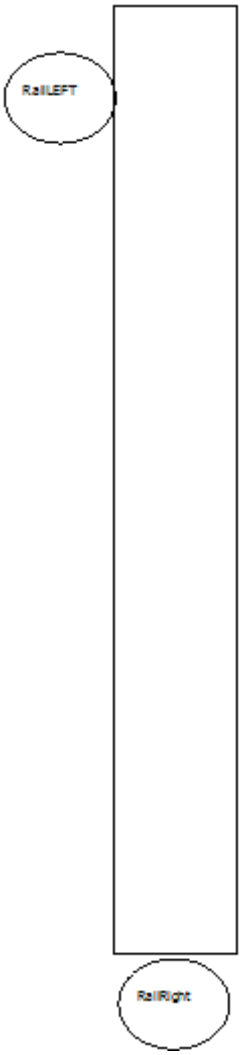



```
model SuppyRail
parameter Integer Nrungs = 1;
Boolean compute_out;
discrete Integer ActiveRung(start = 1);
discrete Integer ActiveRungPrev;
Modelica.Math.BooleanVectors.anyTrue(ToRungs.L2R);
equation
compute_out = FromGround.R2L and not
Modelica.Math.BooleanVectors.anyTrue(ToRungs.L2R);
FromGround.L2R = compute_out;
for i in 1:Nrungs loop
ToRungs[i].s = true;
end for;
for i in 1:Nrungs loop
ToRungs[i].R2L = compute_out and i == ActiveRung;
```

```
end for;
algorithm
when compute_out then
if FromGround.NEXT == (-1) then
ActiveRung := pre(ActiveRungPrev) + 1;
else
ActiveRung := pre(FromGround.NEXT);
end if;
ActiveRungPrev := ActiveRung;
end when;
end SuppyRail;
```

The Supply Rail has more functions. First it is designed to handle “i” rungs. When you drag and drop the supply rail into the environment it asks you the number of the rungs you will have as a parameter but this is dynamic so you can change it also later if u wish to add more rungs to the program.
Moreover when one wants to connect the rail to another component it asks you to declare the number of the rung that the component belongs to (also dynamic if you make a mistake or change something).
Continuing the rail also passes the token (bit) and preserves the sequence but this is in cooperation with the ground rail. If the ground rail sends -1 then the supply rail will pass the token to the next rung (e.g. from rung No1 to rung No2) otherwise it will jump to where the coil says to (checking the NEXT bit that will be not -1 but will have the number of the rung to jump to).
At the end our new ActiveRung is saved as ActiveRungPrev for the cycle to follow.

GroundRail

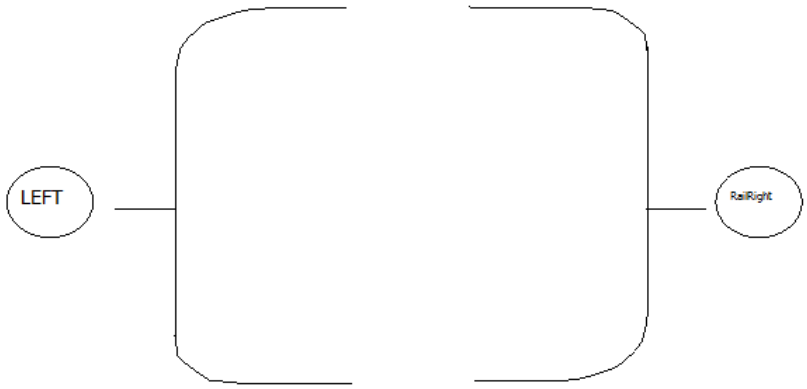



```
model GroundRail
parameter Integer Nrungs = 1;
Boolean compute_out;
Modelica.Math.BooleanVectors.anyTrue(FromCoils.R2L) and not
ToSupplyRail.L2R;
equation
compute_out = Modelica.Math.BooleanVectors.anyTrue(FromCoils.R2L) and
not ToSupplyRail.L2R;
for i in 1:Nrungs loop
FromCoils[i].L2R = pre(compute_out);
end for;
ToSupplyRail.R2L = pre(compute_out);
ToSupplyRail.NEXT = sum(FromCoils.NEXT);
end GroundRail;
```

The Ground Rail is also designed for “i” rungs. Exactly like the supply rail it asks for the same declarations and is also dynamic.

It also receives the bit from the coil before and it will carry the information of where to jump to the supply rail. It preserves the sequential balance as well.

Coil

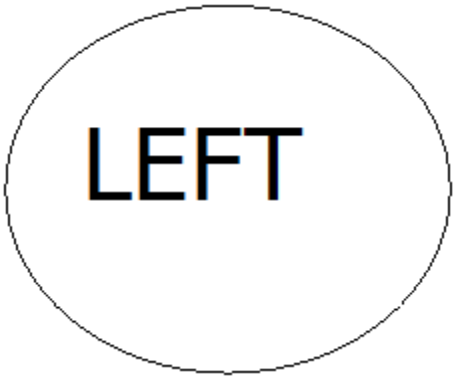



```
model Coil
output Boolean bit;
Boolean compute_out;
equation
compute_out = CL.R2L and not CRR.L2R;
bit = CL.s;
CL.L2R = compute_out;
CRR.R2L = compute_out;
CRR.NEXT = -1;
end Coil;
```

The coil preserves the sequence and pass the token (bit) . It also carries the jump information.

Connectors Cleft-Cright-Rail_Cleft-Rail_Cright

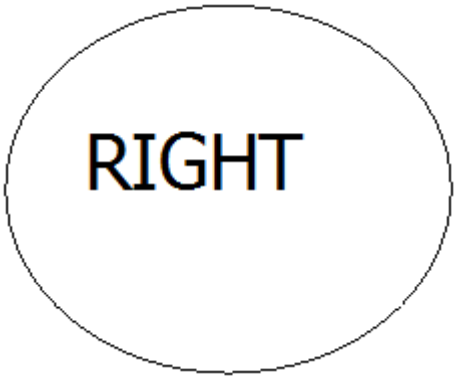
Cleft




```
connector Cleft
input Boolean R2L;
output Boolean L2R;
input Boolean s;
end Cleft;
```

Left connector is the input from the left side of the component for the information and the sequence. It also works as output from right to left.

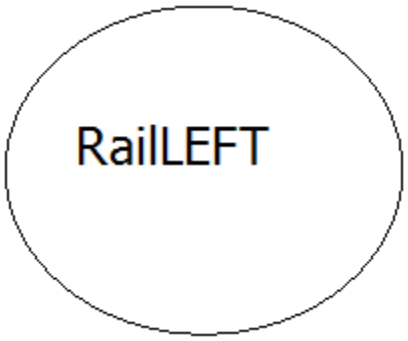
Cright




```
connector Cright
input Boolean L2R;
output Boolean R2L;
output Boolean s;
end Cright;
```

Right connector is the output from the right side of the component for the information and the sequence. It also works as input from right to left.

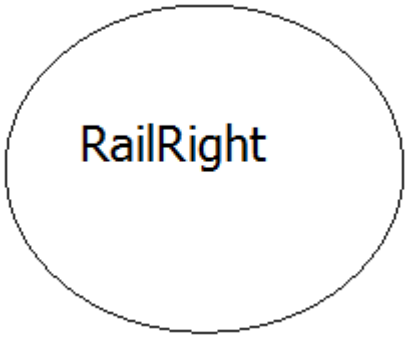
Rail_Cleft




```
connector Rail_Cleft
input Boolean R2L;
output Boolean L2R;
input Integer NEXT;
end Rail_Cleft;
```

Works like the left connector but here it carries the jump information.

Rail_Cright



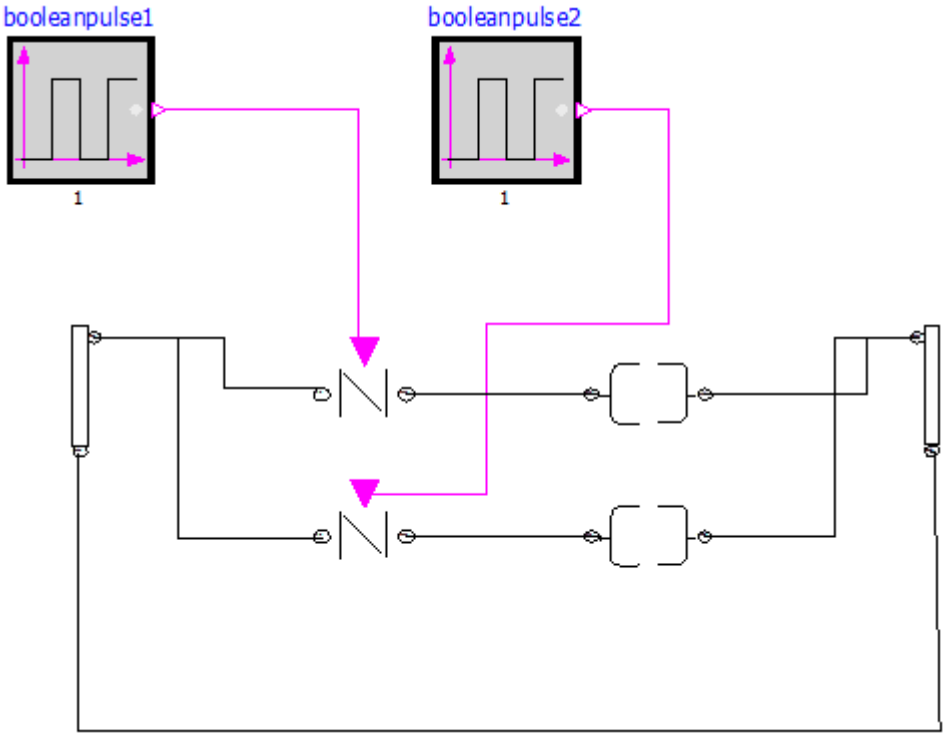

```
connector Rail_Cright
output Boolean R2L;
input Boolean L2R;
output Integer NEXT;
end Rail_Cright;
```

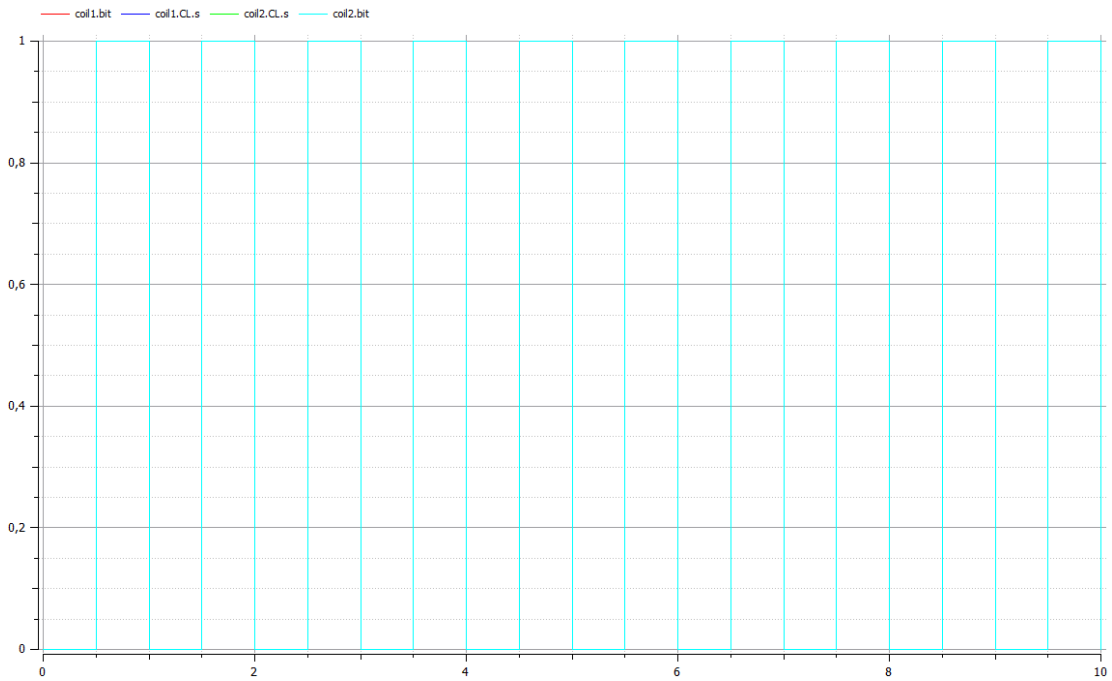
Works like the right connector but here it carries the jump information.

Chapter 5

Adopted Simulation Example

Here we can see a double rung program with simple coils.





Chapter 6

Conclusions and future work

The purpose of this thesis was to represent PLC programs written in the Ladder Diagram language in Modelica. We wanted to see how feasible it is to have everything represented into one of the two worlds above, that for apparent reasons has to be modeling (not programming) one.
Also we wanted to see whether or not one can use equation-oriented language to represent a program while preserving visual assembling paradigm of the former.
In the adopted approach we have managed to have a program working properly (passing the token-bit respecting the priority) with no collisions for n-rungs using the Supply rail, ground rail , normally-closed contact and coil. The latter was solved with the use of equations only (algebraically).

As a consequence, the achieved results can be outlined as follows:

- Addressed representation of LD in Modelica with visual elements
- Solved the problem of representing the code flow

Future works:

- Map LD elements onto the PLC memory to manage variables, static entities and the like.
- Realize additional elements (counters, timers, set/reset coils, jump coils …)

Bibliography

- **Principles of Object-Oriented Modeling and Simulation with Modelica 2.1**
by Peter Fritzson

- **Introduction to Physical Modeling with Modelica**
by Michael Tiller

- **IEC 61131–3: Programming Industrial Automation Systems**
by Karl-Heinz John, Michael Tiegelkamp

- **Automation with Programmable Logic Controllers**
by Peter Rohner

- **www.modelica.org**

- **openmodelica.org**

- **en.wikipedia.org**