OBJECT ORIENTED SIMULATION OF RELAY LADDER LOGIC

J. A. Ferreira, J. L. Azevedo, J. P. Estima de Oliveira
University of Aveiro
Portugal

ABSTRACT

Since their appearance, Programmable Logic Controllers (PLCs) have gain a very strong position in the industrial automation field. The growing complexity of the applications using this type of equipment strongly depends on programmers and maintenance personnel, and so more and better PLC education is of great demand nowadays.

The present work deals with an object oriented simulator for the relay ladder logic language. The simulator is a module of a more general system that emulates a real PLC. This system provides a user friendly graphical interface and allows PLC programming and debugging. The simulator runs as a Microsoft Windows application and was developed using C++. With the same algorithm two simulation modes, allowing different analysis of PLC programs, were implemented: fast simulation and real time simulation.

KEYWORDS

Programmable Logic Controllers, Ladder Diagrams, Object Oriented Programming, Object Oriented Simulation.

1. INTRODUCTION

In the sixties, the need to reduce the costs of the frequent changes in the relay based industrial control systems lead to the development of the concepts associated with programmable logic controllers. The reliability and the flexibility, along with the simple programming language initially used, are some of the factors which influenced their strong implantation in the industrial environment. Regardless of the size, and complexity, almost all PLCs have the same basic components and the same functional characteristics [1].

The spreading of the PLC market and the growing complexity of the applications increased the need of programmers and maintenance personnel, as also the need of more and better training. Lower costs of formation in automation control programs can be achieved through the use of simulation tools. This idea was successfully implemented, namely in instructing industrial controls using ladder diagrams [2].

Our work deals with the design of a virtual PLC (VPLC), intended to be used as a (PLCless) tool to teach relay ladder programming and to develop and debug PLC applications [3]. The VPLC runs in a personal computer (PC) under the Microsoft Windows, and was built using object oriented techniques with C++ [4].

The basic architecture for a real PLC, and the proposed architecture for the virtual PLC are shown in figure 1. In a real PLC, a Central Processing Unit (CPU) executes cyclically a program contained in the program memory. For each cycle, the inputs are acquired at its beginning and, in accordance with them and with the control program, the outputs are updated at its end. During program execution, the CPU uses another PLC memory to store and transfer data.

In the virtual PLC the program is written (using the Relay Ladder Language) and supported by the programming editor. In run mode, the stimuli editor reads the inputs, the VPLC makes the simulation of the ladder program and modifies its memory in accordance with the inputs and the program; at the end it updates the outputs and show them, in a window, using the output graphic module.

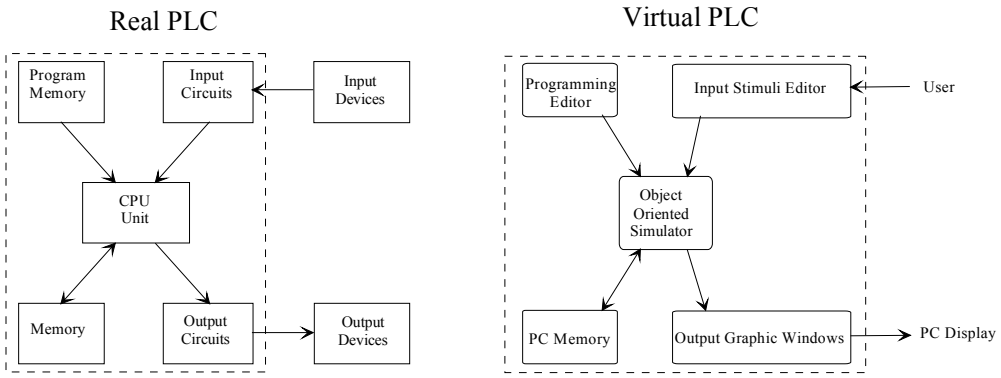


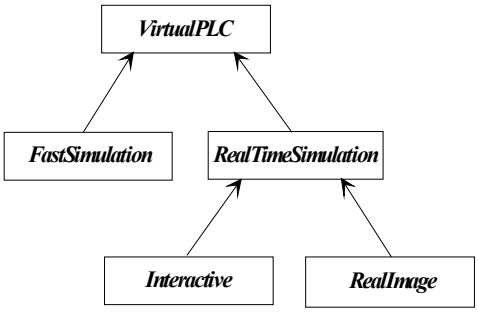
Fig. 1 - Real and Virtual PLC architectures

2. PRESENTATION OF THE SIMULATION MODES

Two simulation modes, allowing different analysis of the results of a program, has been implemented: fast and real time simulations. In the fast simulation mode, the program is executed without pauses and the results are shown in the same stimuli editor that was used to draw the inputs.

The operation of the real time simulation mode is similar to the fast, but the scan cycles are controlled by a clock. The results of this mode can be visualized in two different ways: in a window containing a picture of the real PLC or in an interactive window where the inputs can be modified on-line by the user.

An hierarchy of virtual PLCs were designed, in order to support different simulation modes. This way, the design of a new simulation mode is easy to obtain by deriving a new descendent class. Figure 2 presents the hierarchy for virtual PLCs and the most important data and methods of the class *VirtualPLC*.




```
class VirtualPLC {
protected:
static LadderProgram *prog;
InputStimuli *inputStim;
OutputResults *outputRes;
public:
virtual BOOL VOID InitSimulation(VOID);
virtual BOOL LadderSimulation(VOID);
virtual VOID UpdateOutputStimuli(WORD oStepTime);
virtual VOID UpdateInputStimuli(WORD oStepTime);
virtual VOID StopSimulation(VOID){}
};
```

Fig. 2 - Virtual PLC class hierarchy and part of the base class *VirtualPLC*

The *LadderProgram* object supports the ladder program inside the PC memory. Basically it is a two dimension array of pointers to ladder objects, but also contains the rules of ladder programming.

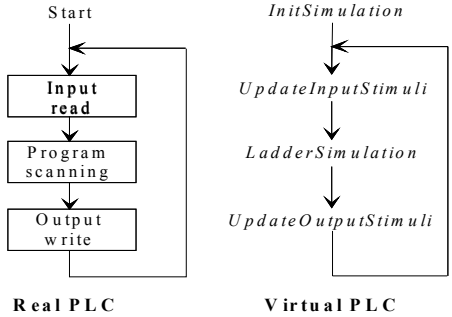


Fig. 3 - Real and Virtual PLCs in run mode

Being a cyclic machine, a PLC executes a program indefinitely, and each program execution is named as a scan. The virtual PLC simulates program execution in a similar way. Figure 3 represents the actions of a real PLC and of the virtual PLC during a scan. *InitSimulation*, *UpdateInputStimuli*, *LadderSimulation* and *UpdateOutputStimuli* are virtual methods defined in the base class *VirtualPLC*.

The virtual method *UpdateInputStimuli* reads the inputs defined in the stimuli editor and updates the respective inputs inside the virtual PLC. The virtual method *UpdateOutputStimuli* refreshes the output graphical windows, after each program scan. The virtual method *LadderSimulation* controls the simulation of a scan of the ladder program; it sends events to stimulate the objects that represent the ladder symbols (each object type contains its own simulation routine).

Fig. 4 presents a printout of a simple program to control a timed coffer. The order to open the coffer (output *Coffer*) only takes place after five clicks on the *Open* push button, followed by a 4 seconds wait period. To close the coffer the delay is 2 seconds after 3 clicks on the *Close* push button. The figure shows the ladder editor, with the ladder program; the stimuli editor, also showing the fast simulation results; the image of a real PLC with blinking lights; and an interactive window performing real time simulations.

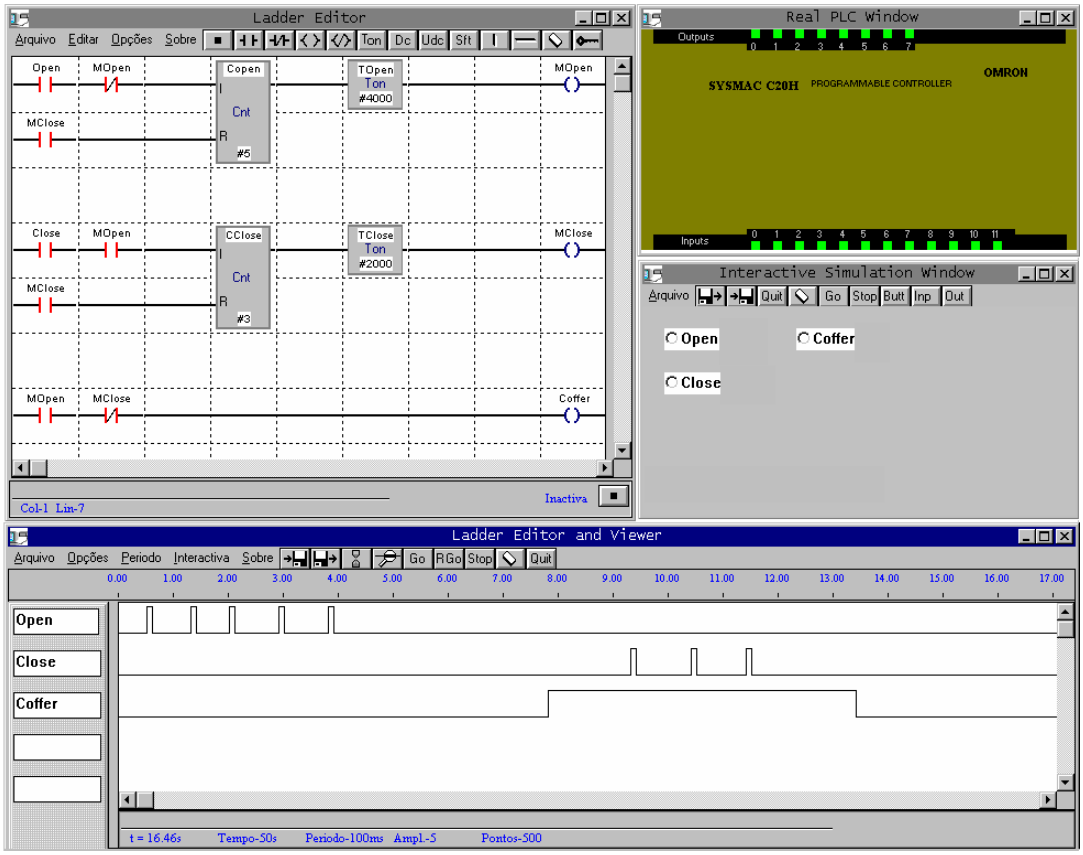


Fig. 4 - Layout of the virtual PLC system with the coffer example

3. RELAY LADDER SIMULATION ALGORITHMS

The advantages of the object oriented techniques in the implementation of simulation algorithms are already accepted. The design of a simulator in such a way involves the communication between objects along the time [5]. In the present case, the algorithm is based on an information passing scheme, using simulation events issued by objects lying in the left side of the ladder diagram and received by the objects located to the right. Simultaneously, the contents of the relevant memory positions are modified in accordance with the type and the action of the objects, which are interpreted in the context of the ladder program.

In the following example (figure 5), stating the logic equation $a.\overline{b} = c$, the execution of the instruction labelled **c (output)**, depends on the logic continuity from node 0 (the start of continuity) up to node **2**. This signal corresponds to the execution condition of the instruction **c**. But before this condition has arrived to node **2,** it must reach node **1**. Thus, the token results from the auto-simulation of the most-left ladder object. The execution condition of one instruction will be false if there is no logic continuity up to the left node of the object, and it will be true if there is logic continuity.

Another question to consider takes place when one or more objects are connected vertically. Considering the example of the figure 6, the logic continuity at node **(1,1)** depends on the objects **a**, **b** and **c**. This suggest that the nodes **(1,1)**, **(1,2)** and **(1,3)** are the same. In fact, the execution condition of **d** can be achieved by the logic continuity through any of the objects **a**, **b** or **c**.

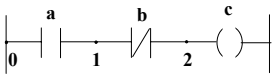


Fig. 5 - Ladder diagram representing the logic equation $a.\overline{b} = c$

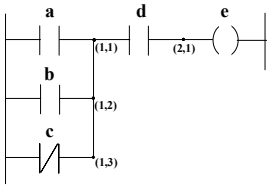
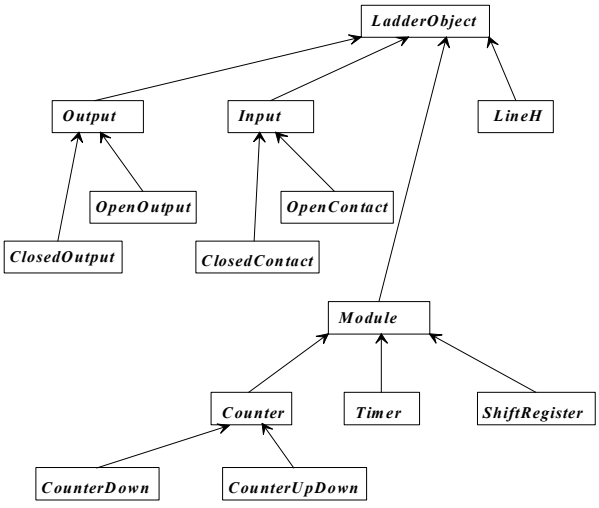


Fig. 6 - Ladder diagram representing the logic equation $(a + b + \overline{c}).d = e$

All the ladder symbols allowed by the ladder editor are objects (instances) of predefined C++ classes. These classes belong to a hierarchy of classes illustrated in figure 7. The base class *LadderObject* defines the functionality that descending classes should have. For instance, each ladder object has a name, a location in the programming editor, and the information about the connections to other symbols, as well as methods that are called to perform operations such as drawing (to draw itself) and simulation (invoked when the symbol behaviour is to be simulated).

The simulation of a ladder program is supported by the virtual methods *Simul* and *OwnerSimulation*, that are defined in the base class ladder objects, *LadderObject*. However, they can be redefined in other classes. The connection between the graphical symbols (and the corresponding ladder objects that represent those symbols) define the way as the information is exchanged between them.




```
class LadderObject {
protected:
   CHAR *objectName;
   CHAR **objectLabel;
   WORD *memLocation;
   Position gridPosition;
   LadderObject **leftObjects;
   virtual VOID OwnerSimulation()=0;
public:
   virtual BOOL Simul(VOID);
   virtual VOID ResetValuesToFirstScan(VOID);
   virtual VOID ResetToOtherScan(VOID);
   virtual VOID Draw(VOID);
   virtual VOID UnDraw(VOID);
};
```

Fig. 7 - Hierarchy of ladder classes and a part of the class LadderObject

For each ladder object, the virtual method *Simul* can be considered as the interpreter for the connections with other ladder objects, and the *OwnerSimulation* method executes the action of the instruction associated with the ladder object.

The goal of the method *LadderSimulation* is the stimulation of the simulation process; the stimulation is done by the invoking the *Simul* method for all terminal objects. This starts a navigation through the ladder editor grid, from the right to the left, followed by the return to the initial object, in a way similar to the simulation process implemented in [6]; this navigation takes advantage of the re-entering characteristics of the virtual method *Simul*. To simulate the action of one object, the input parameter (execution condition) should be known; this is obtained by sending a simulation event (calling the *Simul* method) to the left-connected objects. The right to the left navigation ends when the start of continuity is reached or when a ladder object, already simulated in the present scan, is found.

The left to the right navigation process results from the returning of the successive calls of *Simul* method from the different objects. For each ladder object, its *OwnerSimulation* method is also executed.

For all the ladder objects occupying one single cell (only one input), the interpretation of the program context is the same, thus the *Simul* method, defined in the *LadderObject* class, serves all the single cell objects.

Fig. 8 illustrates the sequence of the successive calls of the method *Simul* for the objects in a simple ladder diagram.

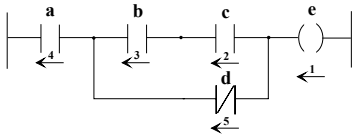


Fig. 8 - Sequence of the successive calls of the method *Simul*

For ladder objects with more than one input the *Simul* method should be redefined, because it needs to interpret adequately the signals connected to its inputs. As an example, the following list presents the differences between *Simul* methods for *LadderObject* and more complex *CounterUpDown* classes.

```
BOOL LadderObject :: Simul()
{
  if (flagSimulation)
    return (tokenSimulation);
  int i = 0;
  while (leftObjects[i++]) {
    tokenSimulation = tokenSimulation || leftObjects[i]->Simul();
    OwnerSimulation();
  }
  if( startContinuity ) {
      tokenSimulation = TRUE;
      OwnerSimulation();
    }
  return (tokenSimulation);
}
```

```
BOOL CounterUpDown :: Simul()
{
  if (flagSimulation)
    return (tokenSimulation);
  if (leftShape[0])
    counterUp = leftObjects[0]->Simul();
  if (leftShape[1])
    counterDown = leftObjects[1]->Simul();
  if (leftObjects[2])
    reset = leftObjects[2]->Simul();
  OwnerSimulation();
  return (tokenSimulation);
}
```

A ladder object, after reading its inputs (by sending simulation events to all its left-connections), must call its own *OwnerSimulation* method in order to simulate the instruction it represents. In this work, the *OwnerSimulation* methods simulate the behaviour of the corresponding instructions of a real PLC (OMRON C20H [7]). As an example, the following listing presents the *OwnerSimulation* methods for *CloseContact* and *OpenOutput* classes.

```
BOOL CloseContact :: OwnerSimulation()
```

```
{
```

```
flagSimulation = TRUE;
if( tokenSimulation )
  return (tokenSimulation = (*memLocation & maskBit) ? FALSE : TRUE );
  return (tokenSimulation = FALSE );
}
```

```
BOOL OpenOutput::OwnerSimulation()
{
flagSimulation = TRUE;
if(tokenSimulation)
   *memLocation |= maskBit;
   else
   *memLocation &= ~maskBit;
return (tokenSimulation);
}
```

With the methodology presented here, it is not difficult to add new instructions to the virtual PLC. To implement a new instruction it is necessary to derive a class from the hierarchy of ladder objects; the virtual method *Simul* should, if necessary, be redefined (except for single cell objects, because of the inheritance mechanism of object oriented techniques); the *OwnerSimulation* method must be redefined for each new ladder object since it is responsible for the particular action of each instruction.

4. CONCLUSIONS

It is well known that simulation is of vital importance in several distinct fields, namely in the industrial automation area. Also accepted are the advantages of object oriented techniques, in the development of simulation algorithms.

In this work, a generic and distributed algorithm was implemented to simulate relay ladder logic language. Generic, because it is independent of the ladder objects (instructions) to be simulated, and distributed, because each instruction, represented by an object, has its own simulation routine. This allows future upgrading without changes in the simulator, even with the introduction of new instructions in the language.

The validation of the simulation algorithm has been done by running several programs, with both the virtual PLC and a real PLC.

5. REFERENCES

1. Hughes, T. A., Programmable Controllers, ISA, 1989.

2. Picard, R. P. and Savage, G. J., “Instructing Industrial Controls Using Ladder Diagrams on an IBM PC”, IEEE Transactions on Education, vol. E-29, no. 1, 1986.

3. Ferreira, J. A., Virtualization of Programmable Logic Controllers, Master Thesis. U. Aveiro, Aveiro (Portugal), 1994.

4. Stroustrup, B., The C++ programming language, Addison-Wesley, 1986.

5. Bischak, D. P. and Roberts, S. D., “Object Oriented Simulation”, Proceedings of the 1991 Winter Simulation Conference, Phoenix (USA), 1991.

6. Estima, J. O., Azevedo, J. L., Ferreira, J. A. and Ferreira, P. J., “Software Development for Programmable Logic Controllers - a methodology and a system”, Proceedings of the IFAC Workshop on CIM in Process and Manufacturing Industries, Espoo (Finland), 1992.

7. OMRON Corporation, Mini H-type Programmable Controllers - Operation Manual, 1990.