A MIDDLEWARE FOR CONTROL APPLICATIONS.

D. Renaud, JM. Perronne, C. Petitjean, M. Hassenforder

ESSAIM/MIPS
Université de Haute Alsace
12, rue des frères Lumière
F-68093 Mulhouse Cedex
{D.Renaud, JM.Perronne, C.Petitjean, M.Hassenforder}@uha.fr

Abstract: This paper proposes an original Object Oriented architecture and an associated framework to set-up easily and safely control software. To improve the design process of such a software several propositions are given: a multi-application support, a layered model, a semantic variable abstraction, a synchronising mechanism and a factoring process. The example illustrates how the architecture can be used to build complex applications. A two tanks plant is simulated and a supervised controlled application illustrates a multi-worlds system with concurrent control and supervision algorithms.

I. INTRODUCTION

Building a Control-Command software application involves the design of a core software component which will be a controller. This controller has to provide specific control algorithms and to interact with physical systems through input and output devices. Often this kind of applications are also connected to supervisors and/or diagnosis processes.

Usually, setting up a Control Command process involves several steps. First, the system, which must be controlled, has to be well known. The model of the system can be either a knowledge model, or an identification model or a compound model [1]. Next, the controller has to be designed according to the system model and the constraints of the desired control strategy. Often, the controlled system is adjusted and validated using simulations. Finally, the controller is implemented in the final target.

In this process, two stages can be distinguished: an offline design stage using simulations and a real time design stage on final targets. These two stages need different kinds of skills and point of views and are realised by different type of designers with the help of different specific tools. The coherence of the designed system is hard to maintain; on the one hand it is important to reach optimal control performances and on the other hand, robust synchronous real time data stream must be provided.

From these considerations, three families of approaches can be noted.

- Designer uses simulation software (Matlab, MAPSim, …) to design the application and then to generate dedicated code for specific target.
- Designer uses simulation software and then makes sub-contract work with consultants.
- Unified environment, where designing and targeting are bound together. In this frame, several related works can be cited. Dias et al. propose a methodology to develop real time distributed applications [2]. Yacoub and Ammar present the benefits of pattern-oriented frameworks to develop closed-loop control systems [3]. Moore et al. promote intelligent controller architectures, which provide functions such as communication abilities, task decomposition, functional decomposition using sub-systems and modules [4]. Maffezzoni et al. demonstrate the suitability of an Object Oriented Modelling approach for control system in industrial areas [5]. This way of solutions shows that intermediate software between control algorithms and hardware is required; middleware are particularly suited to the design or to the implementation of applications in fields of: control-command, identification, diagnosis, supervision processes. Such middleware have to propose the unification of the design process in terms of concepts, semantics, point of views, tools, etc. They also have to hide non-interesting tricky aspects for the designer.

To respond to such considerations, the middleware CoCo'OS (COntrol COmmand Operating System) provides the following features:

- A timing sequence model where data are provided according to traditional sampling sequence at the right frequency.
- The ability to work with either real time clock or virtual time clock (simulated time).
- Data abstractions, which handle hardware or computed values in the sense of control command semantics.
- A control application can be evaluated on the real target or by simulation. In this case, the simulator is set-

up by another instance of the middleware. It simulates the real system and connects its inputs and outputs to the hardware of the control application.

- The ability to activate, to deactivate, and to replace hardware components (a feature quite useful in case of failure). Previous works on reconfiguration topics, Perronne and Hassenforder, propose a first approach in this domain [6].

II. PROPOSED ARCHITECTURE

Such architecture is designed around key concepts; they take into account notions described in this paragraph. Among them appear a layered architecture, several time independent application worlds, synchronous time scheduling, a semantic presentation layer providing variables, functions and synchronised entities. These synchronised entities allow the setting-up of concurrent algorithms and applications.

A. Architecture Layers

The CoCo'OS architecture can be separated in two fields. The first field is concerned by the hardware and the second by the implementation of the application (controller, simulator, supervisor …). Each field can be divided in two layers as shown in figure 1. Such a separation allows an easy replacement of each part. The logic in the upper layer (application) represents only what must be achieved by the application. The presentation layer proposes abstraction to store, to synchronise and to manipulate values in the application sense. The purpose of the port layer and the hardware layer is to propose supports to activate, to deactivate and to replace hardware components, in order to drive sensors and actuators. These two layers are fully described in [7].

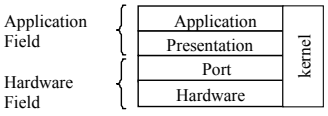

Fig. 1: CoCo'OS Architure layers.

The kernel part proposes registration services to the layers in order to register, to retrieve components and to link them together if necessary. A designed system is build with several instances of components belonging to the different layers and registered by the kernel and realizes what is called in CoCo'OS a *World*.

B. Structural view

Coco'OS allows a multiple *world* scheme, thus several applications, independently conceived, can run concurrently in the same *universe*. Each application, a designer build up, is materialized by a *world*. Moreover, a *world* can be made with one or several instances of layers; *worlds* can share layers or components. For example, a common control-command application requests the four layers (hardware, port, presentation, and application), while its supervisor application just uses two layers (presentation and application). The presentation is shared by the two applications as depicted in figure 2.

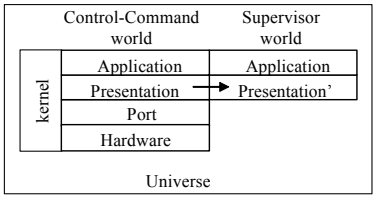

Fig. 2: An example of the universe architecture.

C. Time Scheduling

With a multi-worlds architecture, time scheduling has to be carefully designed. Each world can request different time schedules, however they have to be synchronised. The universe owns the master clock from which a slave clock per world is derived according a ratio factor.
The different scheduled tasks involved in an application are depicted in figure 3. Traditional control process is divided in three stages: sampling (read input data), synchronising (write output data) and solving (evaluate functions).

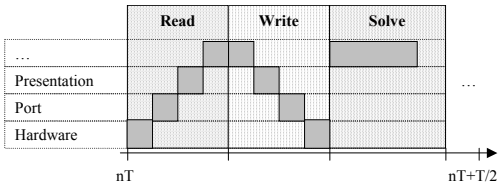

Fig. 3: Process scheduling.

In this layered architecture, each layer has to propose read and write operations in order to transport data from lower to upper layers and from upper to lower layers. Read and write operations have to be synchronised too across the layers. These tasks must have a reserved slot in the scheduled sequence; this is assumed by the world with an update operator. For convenience, each layer updates its data at its own cadence, via a ratio factor, based on the world scheduler frequency. For example, with a master clock at 10ms, the hardware can be updated every 10ms, whereas the port and presentation are updated every 30ms and 50ms respectively. A UML collaboration diagram, in figure 4, presents objects, of the different layers, involved in this scheme [8].

The MasterClock object is a component, which provides the time reference in the architecture. From a hardware clock tick, it generates wall clock time by a simulation process or by a real time clock observer according its

class. This fact allows checking real time behaviour in a simulation process without additional efforts.

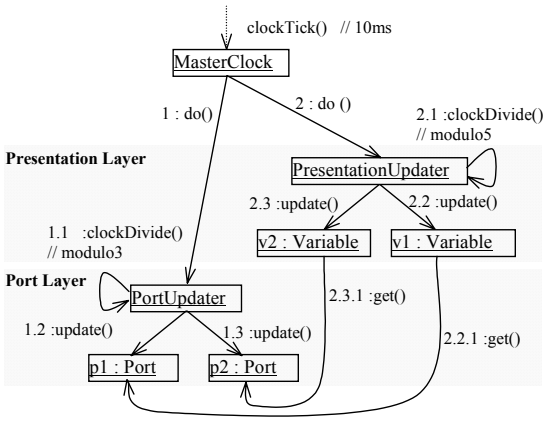

Fig. 4: objects scheduled in layers.

D. Application layer

Semantically, an application can be considered as a set of functions, using variables, evaluated at given times. The designer uses components proposed by the underplaying presentation layer to build up his applications.

E. Presentation layer

The presentation layer provides the abstraction of the variable, function and monitoring entities used by the application layer (figure 5).

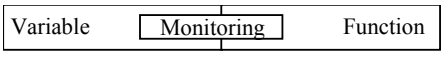

Fig. 5: Details of the presentation layer.

The abstraction named variable eases the handling of physical values or computed values with an application semantic. Three types of variable exist, input or output variables connected to input or output ports (relative to hardware) or non-connected variables (relative to computation).

The variable (figure 6) is able to maintain the current value and the history required by the application. So it is easy to design any kind of time-delayed system without boring with implementation details. The "getValue()" method gives the current value, which was sampled. This is a non-destructive function; each activation gives the same value until an update is achieved. The "getPreviousValue(int delay)" gives a previous value extracted from the history array. The "setValue(Value)" changes the current value. This is also a non-destructive operation; it only changes the internal representation, which will be applied on the port only after the activation of the update function. The private "shift()" function is a helper to shift all values in the history array and loses the oldest one. The “update()” function read from or write to the connected port and invokes shift().

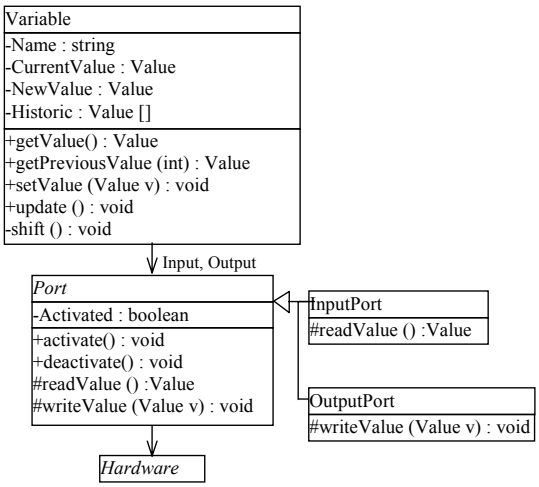

Fig. 6: Variable class diagram.

The abstraction named function eases the handling of relations between variables. A function allows by the intermediary of algorithms, the calculation of a set of variables from others. With such an abstraction, relations (values computations) can be handled as other entities of the architecture. A complex relation set can be broken up in several functions. Using the variable dependency graph through the functions, the architecture allows an optimal computation reorganisation.

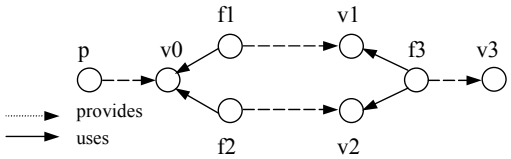

Fig. 7: dependency graph.

In figure 7, the variable v3 is computed by the function f3 which uses the two variables v1 and v2 ; v1 and v2 are respectively obtained via f1 and f2. These two last functions use v0, which has been updated from the port p. f1 and f2 can be computed as soon as v0 is updated and f3 must wait for v1 and v2. Considering the graph, the functions f1 and f2 can be evaluated in a concurrency way. By generalisation, each function in the architecture can be considered as a process waiting on an event. In the above example, f3 is waiting for the event due to the updates of v1 and v2.

At this time two kinds of abstraction must be provided by the architecture, a process one and a synchronisation one. The service abstraction eases the conversion of a function into a process. A service waits for a wakeup event and invokes the computation behaviour of a registered function. The monitor abstraction eases the synchronisation of services. A monitor takes in charge the registration, the blocking and the releasing mechanisms of dependent services. The UML collaboration diagram in figure 8 illustrates the entities

required by the synchronisation between f1 function and v0 variable.

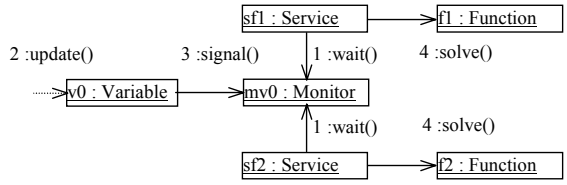

Fig. 8: collaboration diagram.

The two services are blocked by the wait function of the monitor. An update of the variable invokes the signal function of the monitor, which releases services. Further more, time and processor are allotted to all concurrent released services and finally allows the use of the solve function of the Function entities.

F. Application factoring

Setting-up an application involves a lot of components. A supervised control command application, for example, needs several application worlds, acting at different time. Each world manipulates variables, functions and monitoring entities. The input and output variables have to be linked to ports in order to access to the corresponding hardware. Two problems appear, in such a process, complex dependant components have to be created and then bound together.

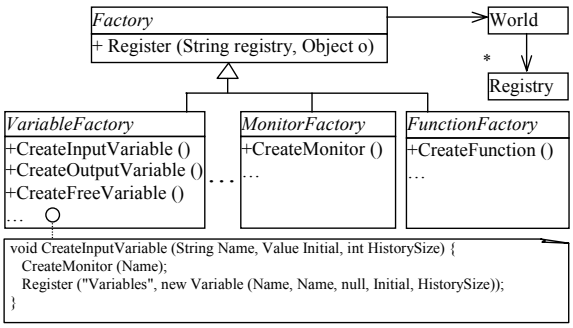

Fig. 9: Factory architecture.

The abstraction named factory eases the creation process of complex dependant components [9]. Sometimes, the creation of an object implies the creation of many others. For example, when a user instantiates a variable (such as v0 in figure 8), its associated monitor (mv0) is also instantiated. Each component is finally registered in the appropriate layer registry in the current world (v0 and mv0 are registered in the variable and monitor registries in the presentation layer). Another aspect of the factory is to provide a semantic close to that of the user. The use of factory is generalized to all components in the architecture as depicted in figure9.

The abstraction named universe eases the setting up of applications, which involve several complex worlds. For each world a step by step process is adopted, the universe abstraction:

1) instantiates the different worlds,
2) creates the different layers in each world,
3) populates the different layers with dedicated application components,
4) shares application components between worlds,
5) establishes links between components in each world,
6) starts the different tasks.

III. IMPLEMENTATION ASPECTS

The real time constraints involved in CoCo'OS architecture impose a careful implementation design and require synchronisation mechanisms and multi-tasks entities. The design of the architecture follows an object-oriented scheme, so the choice of an object oriented language seems particularly relevant. In a first approach, a validation prototype has been designed and realized. The Java language responds to the above constraints providing object oriented features, multithreading and synchronisation mechanisms. Java threads are used to implement concurrent entities: functions, applications. An embedded system, which supports a java runtime environment with real time features, is naturally adapted to the current implementation; in this case, no additional efforts must be achieved. In other situations, an underlying real time operating system has to be chosen; it usually proposes same interfaces: tasks or threads, blocking wait function. In this case, the architecture remains, the only additional effort resides in a transforming process; the java code must be translated in another target language.

IV. APPLICATIONS

The example is chosen (figure 10) to highlight the different features of CoCo'OS and is based on the benchmark described by Staroswieky and Gehin in the IFAC SafeProcess conference [10]. It is composed of two identical connected tanks. Each tank is cylindrical of section A. The inflow Qi is provided by pump Pi (continuous on a specific range) and is controlled by the signal Vi. The flow Qa between the two tanks is controlled by an on/off valve Va. Connecting pipe is at level 0.3m. The on/off valve Vo is an outlet valve located at the bottom of the tank T2.

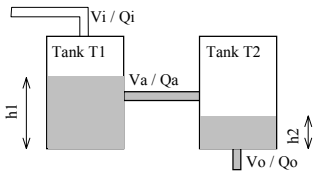

Fig. 10: Two tanks plant.

The two tanks plant will be realized by a simulator build with the CoCo'OS architecture. So, three worlds (figure 11) are going to be set-up in order to evaluate the characteristics of the architecture on a supervisor, a controller and a simulator. This example is a typical case where the design is simplified by the use of several independent applications (worlds). The controller is the main world. The simulator is connected to the controller through a bridge at the hardware level and the supervisor observes specific variables of the controller in the presentation layer.

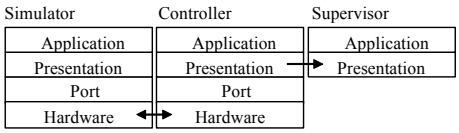

Fig. 11: Two tanks application worlds.

In fact, in the world of the controllers, there are two controllers; the aim of the first one is to maintain h1 level at 0.4m acting on Vi with a PI controller. The flow Qa is considered as a perturbation. The aim of the second controller is to open the valve Vo as soon as the level h2 equals 0.20m.

To set-up this world, the entities of the different layers of this world have to be created using the factory mechanism. The populate function takes in charge this process as presented in listing 1.

To work, this controller world needs h1 and h2 input levels and provides vi and vo valves commands as outputs. Naturally, h1, h2, vi and vo variables are created in the presentation layer as InputVariables or OutputVariables and then will be able to be used in the different control functions. In order to be connected to corresponding hardware associated entities are instantiated in the port and hardware layers with the same name. The internal mechanism of the architecture assumes that entities, which have the same names, are linked. In this example, the hardware created allows the writing of variables in files.

```
class Controller extends UserWorld {
  public void Populate () {
  // Hardware set-up
    hardFactory.CreateHardwareOutputFile ("vi", "vi.txt", 100, 50);
    hardFactory.CreateHardwareOutputFile ("vo", "vo.txt", 100, 50);
    hardFactory.CreateHardwareOutputFile ("h1", "h1.txt", 500, 50);
    hardFactory.CreateHardwareOutputFile ("h2", "h2.txt", 500, 50);
  // Input Ports set-up
    portFactory.CreateInputPort ("h1");
    portFactory.CreateInputPort ("h2");
  // Output Ports set-up
    portFactory.CreateOutputPort ("vi");
    portFactory.CreateOutputPort ("vo");
  // Free Variables set-up
    varFactory.CreateFreeVariable ("Kp", new Value (0.001));
    varFactory.CreateFreeVariable ("Ki", new Value (5e-6));
    varFactory.CreateFreeVariable ("h1ref", new Value (0.40));
    varFactory.CreateFreeVariable ("h2ref", new Value (0.20));
  // Input Variables set-up
    varFactory.CreateInputVariable ("h1", new Value (0.4), 20);
    varFactory.CreateInputVariable ("h2", new Value (0.1), 20);
  // Output Variables set-up
    varFactory.CreateOutputVariable ("vi");
    varFactory.CreateOutputVariable ("vo");
  // Functions set-up
    funcFactory.CreateFunction (new vi_PIController (), "h1");
    funcFactory.CreateFunction (new vo_OnOffController (), "h2");
  }
}
```

Listing 1: Controller world

According to the architecture, the control functions take into account the separation between the algorithm and the activation events. For example, vi_PIController function is called when variable h1 is updated (vo_OnOffController when h2 is updated). The listing 2 shows how the vi_PIController function object is implemented.

```
public class vi_PIController extends VariableFunction {
  private double Kp, Ki, h1ref, h1, vi, integral=0;
  public vi_PIController () {
    AddInputVariable ("h1"); AddInputVariable ("h1ref");
    AddInputVariable ("Kp"); AddInputVariable ("Ki");
    AddOutputVariable ("vi");
  }
  public void Solve () {
    GetValues ();
    double error = h1ref - h1; integral += error;
    vi = Kp * error + Ki * integral;
    SetValues ();
  }
  protected void GetValues () {
    h1 = GetInputVariable(0).GetValue().GetDoubleValue ();
    h1ref = GetInputVariable(1).GetValue().GetDoubleValue ();
    Kp = GetInputVariable(2).GetValue().GetDoubleValue ();
    Ki = GetInputVariable(3).GetValue().GetDoubleValue ();
  }
  protected void SetValues () {
    GetOutputVariable(0).SetValue (new Value (vi));
  }
}
```

Listing 2 : vi_PIController function object

The simulator world follows the scheme prescribed by the architecture. In this case, the hardware is imported from the controller world through the import mechanism. The only one task that remains is the computation of the different flows (Qi, Qa, Qo) and the computation of the two levels (h1, h2), this is achieved by the according functions. To calculate a level, the function needs the input and output variable flows, so the function must be synchronised by the two variable updates. Listing 3 illustrates these aspects.

```
class Simulator extends UserWorld {
  public void Populate () {
  // Input Ports set-up
    portFactory.CreateInputPort ("vi");
    portFactory.CreateInputPort ("va");
    portFactory.CreateInputPort ("vo");
  // Output Ports set-up
    portFactory.CreateOutputPort ("h1");
    portFactory.CreateOutputPort ("h2");
  // Free Variables set-up
    varFactory.CreateFreeVariable ("A", new Value (0.0154));
    varFactory.CreateFreeVariable ("S", new Value (3.6e-5));
    varFactory.CreateFreeVariable ("h0", new Value (0.30));
    varFactory.CreateFreeVariable ("qa");
    varFactory. CreateFreeVariable ("qi");
    varFactory. CreateFreeVariable ("qo");
  // Input Variables set-up
    varFactory.CreateInputVariable ("vi");
    varFactory.CreateInputVariable ("va");
    varFactory.CreateInputVariable ("vo");
  // Output Variables set-up
    varFactory.CreateOutputVariable ("h1", new Value (0.40));
    varFactory.CreateOutputVariable ("h2", new Value (0.10));
```

```
  // Functions set-up
    Vector v1 = new Vector (); v1.add ("qi"); v1.add ("qa");
    funcFactory.CreateEventFunction (
        new ComputeLevel ("Ch1", "h1", "qi", "qa"), v1);
    Vector v2 = new Vector (); v2.add ("qa"); v2.add ("qo");
    funcFactory.CreateEventFunction (
        new ComputeLevel ("Ch2", "h2", "qa", "qo"), v2);
    funcFactory.CreateFunction (new ComputeQa (), "va");
    funcFactory.CreateFunction (new ComputeQi (), "vi");
    funcFactory.CreateFunction (new ComputeQo (), "vo");
  }
  public void Import () {
    Add (new Import ("Controller", "Hardware", "vi"));
    Add (new Import ("Controller", "Hardware", "va"));
    Add (new Import ("Controller", "Hardware", "vo"));
    Add (new Import ("Controller", "Hardware", "h1"));
    Add (new Import ("Controller", "Hardware", "h2"));
  }
}
```

Listing 3: Simulator world

The supervisor world follows also the same scheme; the presentation layer is imported from the controller world. The supervisor observes the sign of the time derivative of the two tank levels (h1, h2) and defines and registers the appropriated functions.

The figure 12 shows the data written in files by the chosen hardware in the different worlds. So simulation results allow verifying that each world works well. During this simulation, valve Va is opened from 3s to 13s, the two controllers and the supervisor act as previously described.

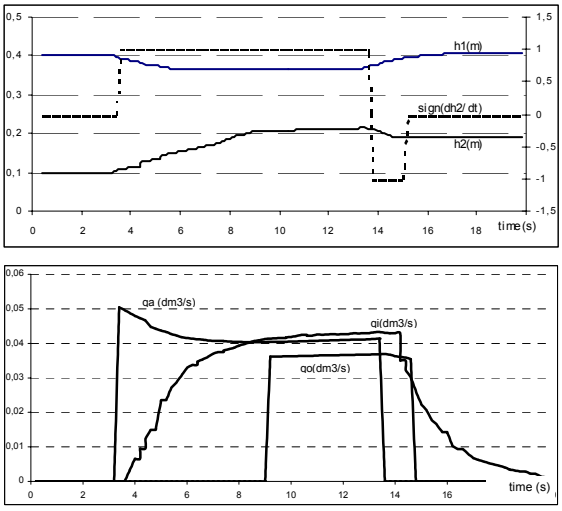

Fig. 12: simulation results.

V. CONCLUSION

A key problem in the implementation of applications in the fields of control-command, identification, diagnosis and supervision, is the complexity of the software design. This paper proposes a solution (CoCo'OS) based on a layered time independent architecture. An application can be break-up in several worlds, which can share components from different layers. This feature eases the design of complex systems where different processes are identified. Layered design brings simplicity and an obvious functional separation between components in a world. For time considerations, virtual or real schedulers and updaters synchronise entities across layers. The resulted architecture dramatically simplifies the tasks of a final designer. Moreover, the only tasks that remain are the applications and algorithms definitions. The data propagation from hardware to variables is hidden by the architecture. A use case application, build using the CoCo'OS architecture, shows how a simulated plant can be controlled and supervised. In case of component failures, no support is actually provided. Future works have to integrate fault detection and reconfiguration features such as selection mechanisms and replacement strategies. So, the robustness of the control-command application made with CoCo'OS architecture will be increased.

VI. REFERENCES

[1] Hassenforder M, Gissinger GL. "Relations between variable granularity and object oriented models. Application to a diesel engine". *In proceedings of IEEE-AVCS'98 conference,* pp 81-86.

[2] Dias O.P., Teixeiras I.M., Teixeiras J.P., Becker L.B., Pereira C.E. "On identifying and evaluating object architectures for real-time applications". Control Engineering Practice, 2001, v9, pp403-409.

[3] Yacoub S.M., Ammar H.H. Toward pattern oriented frameworks. *Journal of Object Oriented Programming*, 2000, v12, n8, pp25-35.

[4] Moore M.L., Gazi V., Passino K.M., Shackleford W.P., Proctor F.M. Complex control system design and implementation. *IEEE Control systems*, 1999, V19, n6, pp12-28.

[5] Maffezzoni C., Ferrarini L., Carpanzano E. "Object-Oriented models for advanced automation engineering". *Control Engineering Practice* **N7**, pp 957-968.

[6] Perronne J.M, Hassenforder. CoCa, object architecture for controller reconfiguration. *Proceedings of SafeProcess*, 2000, V2 pp751-755.

[7] Perronne J.M, Hassenforder. An object architecture for advanced controller software. *Proceedings of ACIDCA 2000*. 2000, Vol SE pp37-42.

[8] Muller P.A (1997), *Modélisation objet avec UML*, Eyrolles, Paris.

[9] Gamma E., Helm R., Johnson R., Vlissides J. *Design patterns, elements of reusable O.O. software.* Addison wesley.

[10] Staroswieki M. Gehin A.L. From Control to supervision. SafeProcess. *Proceedings of SafeProcess*, 2000, Volume 1 pp312-323.