A Survey of High-Level Programming Languages in Control Systems

Fernando Valles-Barajas
Faculty of Engineering, Universidad Regiomontana, México

Abstract: *This paper explains how advanced programming language concepts can be used to increase the readability and maintainability of control process software. The programming language concepts presented in this paper are: function pointers, variable argument functions and three concepts related to object-oriented programming: polymorphism, relationship of composition between classes and class methods. The advantage of every one of these concepts is demonstrated by using control applications. The paper also demonstrates that intelligent control algorithms can be improved by using these concepts. C and C++ programming languages are used to implement the code of the control systems.*

Keywords: *Kalman filter, intelligent control, adaptive control systems, polymorphism, function pointers, and object-oriented programming.*

Received March 6, 2009; accepted November 5, 2009

1. Introduction

Today, many systems are controlled by a control system. Examples of control applications are: robotics, manufacturing processes, military applications and medical devices among others. To control a process, control engineers must obtain a model of the process to be controlled [13]. With this model a control law is then designed in such a way that the entire control system fulfils the requirements of the system users. This control law is later implemented using a programming language. To get robust software the programming language must be carefully chosen by control engineers. If this aspect is not considered when the control system is constructed, the software will be difficult to maintain and understand.

It is important for control engineers to have some knowledge of the advantages of the different programming paradigms. This will help them in the choosing of the right language.

Motivation of the paper: there have been many control applications reported in control engineering literature that show a poor use of the concepts of programming languages. Some of the applications using some of the concepts presented in this paper do not formally explain them. The main goal of this paper is to explain these concepts to the control engineering community so they are able to obtain better software.

Related works: in [11] graphical variant modelling is introduced. This modelling technique is a kind of object-oriented modelling where the classes are graphically specified and the inheritance between classes is defined by stating the differences between the derived classes and the base classes in the subclasses. This novel modelling technique is applied to build an experimental tool that models systems. The tool is based on Simulink, the graphical modeller of MATLAB. The authors present an interesting example which shows the benefits of the approach; an abstract inverted pendulum is defined and then two inverted pendulums (one linear and one non-linear) are defined. With this example the authors demonstrate that the common behaviour of the inverted pendulum does not have to be repeated in the more specific cases and only the differences must be specified.

Modelica is a modelling language designed to specify mathematical models of complex systems [6]. Every model is represented as a class. The variables defined inside the classes represent data related to the system model. In object-oriented languages the behaviour is usually specified using methods (as in Java or member functions such as in C++); in Modelica the behaviour is defined using equations. Once the system is specified using variables and equations that relate these variables, the user can perform simulations to have a better understanding of the system being modelled.

In [9] an exhaustive study is found on the impact of object-oriented programming in Computer-Aided Control System Design (CACSD). The benefits of object-oriented programming (encapsulation, reuse of data and classes among others) in CACSD are demonstrated by the use of examples.

Papers [4, 5] review programming languages in robotics. The author reviews general-purpose languages, like Java. Specific-purpose languages such as the Forth language are also studied. The advantages and disadvantages of these approaches are analyzed in the paper.

Outline of the paper: in this section the motivation has been indicated. The following section provides an explanation of the programming concepts that will be used to implement control systems. Section 3 applies these concepts to control systems. The last section contains concluding remarks.

2. Concepts of Programming Languages

2.1. Function Pointers

A function pointer is type of pointer that exists in programming languages that are based on the C programming language [10]. Even though pointers to functions are one of the most difficult uses of pointers (this could be the reason why they are frequently underutilized), they provide an effective form of subprogram generality. By using function pointers, a function to be executed can be selected on run-time from a set of functions, thus decreasing software complexity. The usefulness of functions pointers is illustrated in Figure 1.

```
#include <math.h>

double get_integral(double a, double b,
double (*fp)(double)){

double sum = 0.0, x;
int n;

for(n = 0; n <= 100; n++){
x = a + n*(b-a)/100.0;
sum += fp(x) * (b-a)/101.0;
}

return sum;
}

int main(int argc, char **argv){

double (*fp)(double);
double a, b, result;

fp = &cos;

// get the values of a and b from the user

result = get_integral(a, b, fp);

return 0;
}
```

Figure 1. Getting the defined integral using function pointers.

The first line of the main function (**double** (*fp)(double);) declares a function pointer called fp. The third line of the main function (fp = &cos;) initializes the pointer *fp*. To achieve this task the address of some function using the reference operator *&* must be given; in this case the address of the cosine function is given. The *get_integral* function, which implements the equation $\int_b^a f(x)dx$, is called in the *main* function by passing the address of the function pointer *fp*. Any function with one *double* parameter and returning a *double* type can be passed as the third argument of the function *get_integral*. This function can be made in a module and distributed to potential users who do not have to change any line of code of this function. This means that by using function pointers the level of reusability can be increased.

2.2. Variable Argument Functions

Variable argument list functions (also known as variadic functions) are functions that can accept a variable number of arguments. Even the data type of the arguments is unknown. This type of function is useful when the number of parameters to be passed is unknown.

Figure 2 shows an example of variadic functions. To declare that a function has a variable list of parameters, fixed parameters are written as usual and then three dots are written to indicate the possibility of variable parameters.

```
#include <stdargs.h>

void variadic_function(int a, int b,...){

   va_list l_arg;
   //b is the last fixed parameter
   va_start(l_arg, b);

double d1 = va_arg(l_arg, double);
int    i1 = va_arg(l_arg, int);

// processing of d1 and i1

   va_end(l_arg);
}
```

Figure 2. Example of a variadic function.

To access the list of variable parameters, variable argument macros defined in the *stdarg.h* library are used. These macros are: *va_arg, va_start* and *va_end*. The macro *va_start* initializes the list pointer *l_arg* to the beginning of the list of variable arguments. Then the macro *va_arg(l_arg, data_type)* is called every time one element of the variable argument list needs to be recovered. The data type of the list of variable arguments should be known prior to using it in a variable argument function, as the second argument of this macro implies. After getting the elements from the variable list, the macro *va_end* is used which ends the use of the list pointer *l_arg*.

2.3. Object-Oriented Programming

Object-Oriented Programming (OOP) is a programming technique in which entities that exist in a system are modelled by using classes [9, 16]. Inside a class, attributes and behaviour common to all the elements that belong to a class are specified. One of the advantages of using a class is the protection of attributes; these are only accessed using the methods defined inside a class. This characteristic of classes is called encapsulation and it is used to maintain the consistency of class data.

The following relationships between the entities of a system can be modelled: composition, association, dependency and generalization.

The concept of polymorphism, which is one of the fundamental concepts of OOP, is explained as follows.

2.3.1. Polymorphism

Polymorphism means multiple implementations one interface [3]. This is a mechanism used in programming languages to provide more readable code. There are two types of polymorphism, polymorphic types and polymorphism related to functions. Operator overloading and function overloading are both types of function polymorphism.

By using operator overloading the meaning of one operator can be expanded. As an example, let us analyze the sum operator + which is defined in programming language to work with integer and real numbers.

It is well known that a valid operation for complex numbers is the sum of complex numbers, but in many programming languages this data type is not included and therefore this operation is not defined.

```
class Complex{
private:
 float real, imag;
public:
  Complex();
  Complex(float, float);
  Complex operator+( Complex );
  // other operations
};

int main(int argc, char **argv){

  Complex c1(1, 2), c2(2, 4), c3;

  c3 = c1 + c2;

  return 0;
}
```

Figure 3. An example of operator overloading.

By using operator overloading, not only the sum operation for complex numbers can be defined but also other valid operations of the complex numbers. Fig. 3 presents one example that implements the operations of complex numbers. Without using this characteristic the operations would have being implemented with functions, resulting in a more difficult code to read.

2.3.2. Instance and Class Methods

A class can be defined as a template to create objects. All the objects created from a template share attributes and behaviour. The behaviour of objects is implemented by using methods. To perform an operation on an object, a method has to be defined. Sometimes an operation is not related to one specific object. Methods that only apply to one object are called instance methods and methods that apply to the whole class are called class or static methods. An example of a class method is a method that returns the number of objects created from a class.

3. Application of Advanced Programming Concepts to Implement Control Algorithms

In this section the advanced concepts of programming languages described in the previous section are applied to build software related to control systems. The following list explains how these concepts are applied to control systems:

- Operator overloading, which is a kind of polymorphism, is explained using the Kalman filter. It will be shown that this concept is not exclusive of object-oriented programming; it also exists in procedural programming.
- Function pointers will be applied to neural networks and genetic algorithms. These algorithms were included in this paper because of their strong application in control engineering. For example, genetic algorithms have been applied in control engineering to identify processes and to tune controllers [12]. On the other hand, the use of neural networks in control engineering is also intensive. Neural networks have been applied to detect faults in control systems, to model processes and to control non-lineal processes [14].
- Variable argument functions will be applied in the calculation of the control signal u of a PID controller.
- The concept of inner classes is applied to model the composition relationship between an adaptive control system and its components (supervisor, process, controller, parametric adaptation algorithm and controller design).
- The concept of class methods will be illustrated by building a class assigned to check the stability of processes.

3.1. Implementing Kalman Filters Using Operator Overloading

To illustrate the concept of operator overloading, which is a kind of polymorphism, let us suppose that the states of a process needs to be estimated. Also, suppose that the states are not directly available but can be inferred from noisy measurements. A solution for this problem is the Kalman filter. A Kalman filter is an online-recursive algorithm that estimates the states of a system based on noisy measurements [8]. The system is modelled by the discrete-time linear equation:

$$x_k = Ax_{k-1} + Bu_k + w_{k-1} \tag{1}$$

using noisy measurements represented by the equation.

$$y_k = Hx_k + v_k \tag{2}$$

where:

- $x \in R^n$ is a vector containing the state of the process.
- $A \in R^{n \times n}$ is the state transition matrix and it relates the state of the system at a instant *k-1* to the state at instant *k*.
- The $B \in R^{n \times l}$ matrix relates the control input $u \in R^l$ to the state x.
- $w \in R^n$ represents the uncertainty in the process and is modeled as white noise having a normal probability distribution $p(w) \sim N(0,Q)$. v models the noise in the measurement and is also modeled as white noise $p(v) \sim N(0,R)$.
- $y \in R^m$ is the process measurement vector.
- $H \in R^{m \times n}$ is the measurement matrix and it relates the state of the system with the measurements.

In the Kalman filter, $\hat{x}_k^-$ is a priori state estimate at time *k* and is calculated based on past values of the output *y*. $\hat{x}_k$ as a posteriori state estimate at time *k* and is calculated based on past and current values of the output *y*.

```
initialize k←0
set initial estimates for $\hat{x}_{k-1}$, $P_{k-1}$
loop
    I.- predictor step

        1.- project the state ahead
        2.- project the error covariance ahead

    II.- corrector step

        1.- compute the Kalman gain $K_k$
        2.- measure the process to obtain $y_k$
        3.- update the a posteriori state estimate $\hat{x}_k$
        4.- update the a posteriori error covariance $P_k$

    increase k
end loop
```

Figure 4. *N*-dimensional Kalman filter algorithm.

Using $\hat{x}_k^-$ and $\hat{x}_k$ a priori estimate error covariance P_k^- and a posteriori estimate error covariance P_k can be obtained. This is done by first getting a priori and a posteriori estimate error ($e_k^- \equiv x_k - \hat{x}_k^-$ and $e_k \equiv x_k - \hat{x}_k$). The a priori estimate error covariance is then defined as $P_k^- = E[e_k^- e_k^{-T}]$ and the a posteriori estimate error covariance is defined as $P_k = E[e_k e_k^T]$.
A Kalman filter algorithm has two steps. A predictor step and a corrector step. Fig. 4 contains the algorithm to implement the *N*-dimensional Kalman filter.

where steps I-1, I-2, II-1, II-3 and II-4 are given respectively by equations 3-7.

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_k \tag{3}$$

$$P_k^- = AP_{k-1}A^T + Q \tag{4}$$

$$K_k = P_k^- H^T \left[HP_k^- H^T + R\right]^{-1} \tag{5}$$

$$\hat{x}_k = \hat{x}_k^- + K_k\left(y_k - H\hat{x}_k^-\right) \tag{6}$$

$$P_k = \left(I - K_k H\right)P_k^- \tag{7}$$

To implement the Kalman filter algorithm a matrix library can be made as shown in Figure 5.

In the main function of Figure 5 the code to implement equation 5 is presented. As the reader may notice, this implementation is cumbersome and does not resemble equation 5. It will be difficult to understand what this program does.

```
// matrix.h

float** multiply(float **, float**);
float** inverse(float **);
float** transpose(float **);
float** sum(float**, float **);

// definition of other prototypes

#include matrix.h

int main(int argc, char** argv){

 float **P, **H, **R;

 // other definitions and operations
 PHt = multiply( P, inverse(H) );
 HPHt = multiply( H, PHt );
 // compute the Kalman gain
 K = multiply( PHt, inverse(sum( HPHt, R ));

 return 0;
}
```

Figure 5. Implementation of equation 5 of the Kalman filter algorithm without using operator overloading.

Another way to implement eq. 5 is to use operator overloading. To do this, a class Matrix is defined in Figure 6.

```
#ifndef MATRIX H
#define MATRIX H
using namespace std;
class Matrix{
 friend Matrix operator*(float, Matrix);
 friend Matrix operator*(Matrix, float);
 friend ostream&
 operator<<(ostream&, const Matrix&);
private:
 float **data;
 int n, m;
public:
  Matrix();
  Matrix(const Matrix&); // copy constructor
  ~Matrix();
  Matrix operator^(char);// transpose
  Matrix operator^(int); // inverse
  Matrix& operator=(const Matrix&);
  Matrix operator+(const Matrix&) const;
  Matrix operator*(const Matrix&) const;
};
#endif

#include "Matrix.h"

int main(int argc, char **argv){

  Matrix K, // Kalman gain
         P, // priori estimate error covariance
         H, // measurement matrix
         R;

  char  T; //dummy

  // definition of other variables

  // operations to fill the necessary variables

  K = P*(H^T)*((H*P*(H^T)+R)^-1);

  return 0;
}
```

Figure 6. Implementation of equation 5 of the Kalman filter algorithm using operator overloading.

This example shows that by using operator overloading, more readable code can be obtained; the code of fig. 6 closely resembles eq. 5 so it is very easy to understand this version of the Kalman filter algorithm. And 2 are changed respectively to eq. 8 and 9 Here, one question may arise: is operator overloading a new concept developed in the object-oriented theory?. Amazingly the answer is negative. Let us demonstrate this statement.

Let us modify the previous problem in such a way that only one state is estimated as in equation 1.

$$x_k = ax_{k-1} + bu_k + w_k \tag{8}$$

$$y_k = hx_k + v_k \tag{9}$$

The Kalman filter algorithm is changed to

```
initialize  k←0
set an initial estimates for $\hat{x}_{k-1}$, $p_{k-1}$
loop
  a. predictor step
     1. project the state ahead
     2. project the error covariance ahead

  b. corrector step
     1. compute the Kalman gain $k_k$
     2. measure the process to obtain $y_k$
     3. update the a posteriori state estimate $\hat{x}_k$
     4. update the a posteriori error covariance $p_k$
     increase k
end loop
```

Figure 7. Scalar Kalman filter algorithm.

where steps I-1, I-2, II-1, II-3 and II-4 are given respectively by equations 10-14.

$$\hat{x}_k^- = a\hat{x}_{k-1} + bu_k \tag{10}$$

$$p_k^- = a^2 p_{k-1} + Q \tag{11}$$

$$k_k = \frac{hp_k^-}{h^2 p_k^- + R} \tag{12}$$

$$\hat{x}_k = \hat{x}_k^- + k_k\left(y_k - h\hat{x}_k^-\right) \tag{13}$$

$$p_k = p_k^-\left(1 - hk_k\right) \tag{14}$$

The scalar Kalman filter algorithm can be implemented in a procedural language, for example in the C programming language. If equations 13 and 14 are carefully examined, it can be seen that the minus operator is overloaded; this operator works with two real numbers in equation 13 (y_k and the result of $h\hat{x}_k^-$) and in equation14 this operator also works with one integer (the number 1) and the result of hk_k.

The reader is referred to [8] to have a more complete description of the Kalman filter.

3.2. How Intelligent Control Software Can be Improved by Using Function Pointers

3.2.1. Genetic Algorithms

Genetic algorithms are search based algorithms used in optimization [7]. These algorithms generate a set of possible solutions represented as strings of bits. Every string represents a chromosome of a particular individual. The algorithm selects the best individuals based on the strength reflected in the chromosomes. This reveals that genetic algorithms are inspired in natural selection.

Because the process of getting a model process and designing a controller can be represented as optimization problems, genetic algorithms have been used to attack these problems [12].

```
struct individual{
/* chromosome string for an individual */
  String chromosome;
/* fitness of an individual */
  double  fitness;
/* definition of other variables */

};

void obj_fun(struct individual *i,
  double (*fp)(double)) )

  /* definition of variables */
  float x;

  /*
   * code to convert a string to a
   * number between [0,1].
   */

   i->fitness = fp(x);
}
```

Figure 8. Part of the implementation of a genetic algorithm by using a function pointer.

The implementation of a genetic algorithm can be improved by using function pointers. To understand how function pointers can be used in genetic algorithms, let us define the representation of an individual using a structure and the function that evaluates this individual, as is shown in Figure 8

The *obj_fun* converts an individual chromosome from a string into a number between 0 and 1 (this is stored in variable *x*). Then variable *x* is passed as an argument to a function that evaluates the strength of the individual. This function has to be specified by the user and it depends on the problem to be optimized. In the case of Figure 8, the function is passed as an argument to the *obj_fun*. The main advantage of using a function pointer in the *obj_fun* is that the user does not have to modify this function to specify which function she/he wants to optimize.

As can be seen from this example, function pointers are used to select a function at run-time.

3.2.2. Neural Networks

Neural Networks (NN) are algorithms that imitate the learning process of the human brain [15]. The most important applications in control engineering are identification and control of processes [14]. Let us suppose that it is required to model a process which can be modelled by using a first-order differential equation. The transfer function of the equation is:

$$G_p(s) = \frac{Ke^{-\theta s}}{\lambda s + 1} \tag{15}$$

The discrete version of this equation is given by the equation.

$$G_p(z^{-1}) = \frac{z^{-d}B(z^{-1})}{A(z^{-1})} = \frac{z^{-d}(b_1 z^{-1} + b_2 z^{-2})}{1 + a_1 z^{-1}} \tag{16}$$

where:

$$a_1 = -e^{-T_s/\tau} \tag{17}$$

$$b_1 = K\left(1 - e^{L - T_s/\tau}\right) \tag{18}$$

$$b_2 = Ke^{-T_s/\tau}\left(e^{L/\tau} - 1\right) \tag{19}$$

Equation 16 can also be represented using the recurrence equation:

$$y(k) = -a_1 y(k-1) + b_1 u(k-d-1) + b_2(k-d-2) \tag{20}$$

As can be observed, the value of output variable *y* at instant *k* depends on its previous value and it also depends on the values of the input variable *u* at the instants *k-d-1* and *k-d-2*. With this information a first-order model can be obtained by using an NN. Fig. 9 shows the structure of the NN to model a first-order process.

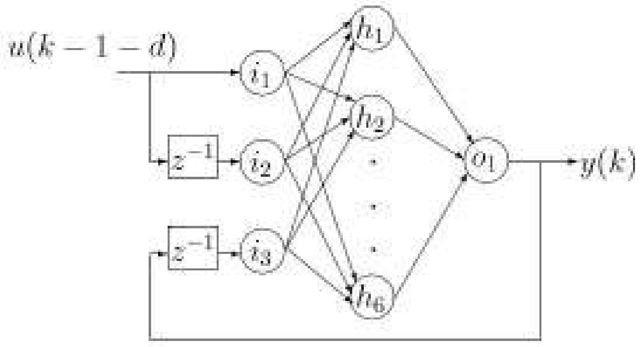

Figure 9. A neural network modelling a first-order process.

Each of the circles in this figure represents a processing unit. The behaviour of this processing unit depends of an activation function. Examples of activation functions are: linear, log-sigmoid, step and saturating linear.

The function that evaluates the output of a processing unit could receive as an argument a function pointer that points to an activation function; in this way the code that evaluates a processing unit can be hidden to the user. This would increase the level of usability.

3.3. Getting the Control Signal *u* by Using Variable Argument Functions

A PID controller is one of the most widely used controllers in industry [1]. This controller has the following structure:

$$U(s) = K_c E(s)\left(1 + \frac{1}{s\tau_i} + s\tau_d\right) \tag{21}$$

where e_t is the control error and u_t is the control signal and $K_{c,}$ τ_i and τ_d are respectively the proportional gain, the integral time constant and the derivative time constant. $K_{c,}$ τ_i and τ_d are the controller parameters.

A PID controller is often called a three-term controller because it consists of three terms: a proportional term *P*, an integral term *I* and a derivative term D.

In practical situations, the structure of equation 21 is not used; instead the following structure is used:

$$U(s) = K_c\left(bR(s) - Y(s) + \frac{E(s)}{s\tau_i} - \frac{s\tau_d}{1 + \frac{s\tau_d}{N}} Y(s)\right) \tag{22}$$

This structure helps to mitigate strong variations in the control signal due to noise and peaks in the control signal are also avoided when a change in the set-point occurs. To implement this control law it is necessary to approximate the derivative and integral parts. The proportional term $P = K(br - y)$ is implemented by replacing the continuous variables by their sample versions

$$P_k = K(br_k - y_k) \tag{23}$$

where *k* denotes the sampling instants.
The integral term is approximated by the equation

$$I_{k+1} = I_k + \frac{Kh}{T_i} e_k \tag{24}$$

in which the derivative is approximated by a forward difference.

In equation 24 *h* represents the sampling time. The derivative term is obtained by approximating the derivatives by a backward difference:

$$D_k = \frac{T_d}{T_d + Nh} D_{k-1} - \frac{KT_d N}{T_d + Nh}(y_k - y_{k-1}) \tag{25}$$

Figure 10 shows a way to implement the control law of a PID controller using variable argument functions. An identification of the process is made in the *main* function before getting the control signal *u* and then with the parameters of the process the PID controller tuning is done. Here, the interesting point is that the user can select the kind of controller; the user can select a P, PI, PD or PID controller. The *get_u* function only receives the necessary parameters to obtain the control signal *u*. This function is made in such a way that the necessary parameters are passed as fixed parameters and the parameters that are not always needed are passed using the variable argument function syntax [10].

3.4. Implementation of an Adaptive Control System Using Object-Oriented Programming

In this section two concepts of object-oriented programming languages are used to implement an adaptive control system: inner classes and class methods.

The configuration of an indirect adaptive control system is shown in Figure 11. In this kind of adaptive control, a model of the process $G_p(z^{-1})$ is obtained based on a set of input-output measurements ($u(k)$, $y(k)$) and then the controller $G_c(z^{-1})$ is designed with this model [2]. The Parameter Adaptation Algorithm (PAA) block is responsible for obtaining the parameter vector of the process (see $\hat{\theta}_{PAA}(k)$ in fig. 11).

The controller design block specifies the parameters of the controller $G_c(z^{-1})$ based on the model obtained by the PAA and on the desired performance specified by the system operator.

An adaptive control system must control a process in spite of disturbances $d_1(k)$, $d_2(k)$, noise $n(k)$ and parametric variations of the process. The supervisor block is in charge of detecting any event that may provoke a decrease in the performance of system.
In fig. 11, $y_{ref}(k)$ is the reference, $e(k)$ is the control error, $u(k)$ is the manipulated variable and k is the k_{th} sampling time.

```
#define P_Gc   0
#define PI     1
#define PD     2
#define PID    3
struct GcPID{
  double Kc, Ti, Td;
  double b, N;
};
double get_u(int PID_type, double Kc, double b,
     double y, double r, ...){
  va_list ap;
  va_start(ap, r);
    double T, Ti, P;
  static double I = 0.0, D = 0.0, yold =0.0;
  double a1, a2;
   switch(PID_type){
    case P_Gc:
      P = Kc*(b*r-y);
      I = D = 0.0;
      break;
    case PI:
      Ti = va_arg(ap, double);
      T  = va_arg(ap, double);
      P = Kc*(b*r-y);
      I = I + ((Kc*T)/Ti) * (r-y);
      D = 0.0;
      break;
    case PD:
      a1 = va_arg(ap, double);
      a2 = va_arg(ap, double);
      P = Kc*(b*r-y);
      D = a1*D-a2*(y-yold);
      break;
    case PID:
      Ti = va_arg(ap, double);
      T  = va_arg(ap, double);
      a1 = va_arg(ap, double);
      a2 = va_arg(ap, double);
      P = Kc*(b*r-y);
      I = I + ((Kc*T)/Ti) * (r-y);
      D = a1*D-a2*(y-yold);
      break;
    default:
      break;
  }
   yold = y;
   return (P + I + D);}
int main(int argc, char **argv){
  struct GcPID pid;
  double y, u, r;
  double T;
// tune the controller to get its parameters
// (Kc, Td, Ti)
// the user selects the type of controller
// calculate auxiliar variables for the D term
  double a1 = pid.Td/(pid.Td + pid.N*T);
  double a2 = pid.Kc*pid.Td*pid.N/(pid.Td +
     pid.N*T);
    // read y from the process and r form the user
  u = get_u(P_Gc, pid.Kc, pid.b, y, r);
  return 0; }
```

Figure 10. Implementing the control law of a PID controller using variable argument functions.

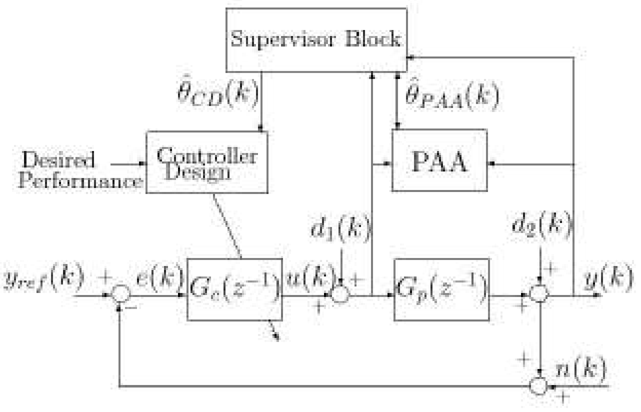

Figure 11. Configuration of an adaptive control system.

3.4.1. Inner Classes

When a system is modelled using the object-oriented paradigm, the first step for doing this is to identify the entities in the system and then the relationships between these entities must be established.

```
class ACS{

 class Gc{
  private: float *R, *S, *T;
 };

 class PAA{
  private: float forgeting_factor;
 };

 class Gp{
  private: float *A, *B, d;
 };

 class GcDesigner{
  public: tuneGc(){}
  };

 ACS(){} // constructor
};
```

Figure 12. Using inner classes to implement the adaptive control system of Figure 11.

Five entities were identified in the control system of Figure 11the supervisor, the controller, the process, the controller design and the parameter adaptation algorithm.

The first relationship established between these entities is the composition relationship which is used to model the relation between one component and the parts that compose this component. In this case the component is the Adaptive Control System (ACS) and its parts are the blocks of Figure 11. In Figure 12 the concept of inner classes was used to model the relationships between the ACS and its parts.

3.4.2. Class Methods

Usually the methods defined inside a class are designed so that the status of an object is recovered or modified. For example the method *tuneGc* defined inside the class *GcDesigner* of fig. 12 designs the controller *Gc*. Sometimes a method that does not necessarily apply to a particular object is needed; this type of method is called class method. For example, in fig. 13 a class to check the stability of a transfer function is defined. m_1 and m_2 are methods to check the stability of a process. These methods could be based on the Liapunuv stability or on the poles position of a characteristic equation, for example.

```
class Gp{}

class Stability{
private:
 // definition of variables
public:
 static void m1(Gp);
 static void m2(Gp);
 // other declarations
};

int main(int argc, char **argv){

 Gp gp;

 Stability::m1(gp);

 return 0;
}
```

Figure 13. Using class methods to create a stability class.

4. Conclusions

In this paper five programming concepts were used to implement control systems. The concept of polymorphism was used to implement the *n*-dimensional Kalman filter. By using the scalar Kalman filter it was demonstrated that polymorphism also exists in procedural languages like the C programming language. Function pointers were used to make an efficient implementation of genetic algorithms and NN. A variable argument function was used to implement a PID controller. The last concepts presented in this paper were composition and class methods which are two of the fundamentals of object-oriented programming. The author of this paper believes that the understanding of these concepts will help to develop a code that is more readable and easier to maintain.

References

[1] Astrom J. and Hagglund T., *PID Controllers: Theory, Design and Tuning*, Hagglund Publisher, 1995.

[2] Astrom J. and Wittenmark B., *Adaptive Control*, Dover Publications, 2008.

[3] Eckel B. and Allison C., *Thinking in C++, Volume 2: Practical Programming*, Prentice Hall, 2003.

[4] Frenger P., "Robot Control Techniques, Part One: A Review of Robotics Languages," *Computer Journal of SIGPLAN Notices*, vol. 32, no. 4, pp. 27-31, 1997.

[5] Frenger P., "Robot Control Techniques, Part Two: Forth as a Robotics Language," *Computer Journal of SIGPLAN Notices*, vol. 32, no. 6, pp. 19-22, 1997.

[6] Fritzson P., *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, Wiley-IEEE Press, 2003.

[7] Goldberg E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Professional, 1989.

[8] Grewal M., *Kalman Filtering-Theory and Practice Using MATLAB*, Press Wiley, 2001.

[9] Jobling W., Grant A., Barker W., and Townsend P., "Object-Oriented Programming in Control System Design: A Survey," *Computer Journal of* Automatica, vol. 30, no. 8, pp. 1221-1261, 1994.

[10] Kernighan W. and Ritchie M., *The C Programming Language*, Prentice Hall, 1998.

[11] Kinnucan P. and Mosterman P., "A Graphical Variant Approach to Object-Oriented Modeling of Dynamic Systems," *in Proceedings of the Summer Computer Simulation Conference*, USA, pp. 513-521, 2007.

[12] Kristinsson K. and Dumont A., "System Identification and Control Using Genetic Algorithms," *Computer Journal of IEEE Transactions on Systems, Man, and Cybernetics*, vol. 22, no. 5, pp. 1033-1046, 1992.

[13] Landau D. and Zito G., *Digital Control Systems: Design, Identification and Implementation. (Communications and Control Engineering)*, Springer, 2006.

[14] Narendra S. and Parthasarathy K., "Identification and Control of Dynamical Systems Using Neural Networks," *Computer Journal of IEEE Transactions on Neural Networks*, vol. 1, no. 1, pp. 4-27, 1990.

[15] Norgaard M., Ravn O., Poulsen K., and Hansen K., *Neural Networks for Modelling and Control of Dynamic Systems: A Practitioner's Handbook*, Springer, 2008.

[16] Stroustrup B., *The C++ Programming Language: Special Edition*, Addison-Wesley, 2000.

Fernando Valles-Barajas obtained a graduate degree in computer science at Center for Research and Graduate Programs of La Laguna Institute of Technology in 1991. He received an MS in control engineering in 1997, and a PhD in artificial intelligence in 2001, from Monterrey Institute of Technology (ITESM) campus Monterrey. He was a research assistant at Mechatronics Department of ITESM Campus Monterrey (1997-2001). He received certification as a PSP developer from the Software Engineering Institute of Carnegie Mellon University in 2008. He is member of the IEEE and ACM. His research interests include topics in software engineering and control engineering. Currently, he is full time professor in the Department of Information Technology at Universidad Regiomontana, Monterrey, Nuevo León, México.