OBJECT-ORIENTED APPROACH AND PAC MODEL: DESIGN OF THE GReAt (Graph Researcher Assistant) ENVIRONMENT

F. LOSAVIO, A. MATTEO, O. ORDAZ, O. MEZA, W. GONTIER
Centro de Ingeniería de Software Y Sistemas (ISYS)
Escuela de Computación, Universidad Central de Venezuela
Apdo. 47567, Caracas, Venezuela

ABSTRACT
The main goal of this work is to present a systematic development process of a graphical interactive application for researchers and students in the field of graph theory, emphasizing the use of the PAC (Presentation, Abstraction and Control) multiagent model. This model, inspired on the MVC model, has been used to design the system's architecture. A general implementation schema, integrating the object-oriented paradigm and preserving the PAC model communication principles is proposed. This implementation favors maintenance and reuse of the resulting code, facilitating also the work of programmers' teams.

KEYWORDS: design of interactive environments, user-interface development, multiagent model, PAC model, PAC agent implementation, object-oriented programming, digraph family, digraph editors.

1. INTRODUCTION
The main goal of this paper is to present the systematic development process for building an interactive environment for students and researchers in the specialized graph theory domain. We essentially emphasize the transition from the PAC agents constituting the system's architecture to the classes implementing them. The GReAt environment integrates a set of tools focused on one hand towards the drawing of objects such as digraphs and families of digraphs and on another hand, on the formulation of conjectures and theorem proving, mostly based on these drawings. Several systems have been already constructed with this goal in mind, providing functionalities for graph edition and properties computing [Bau 90], [Bau&Al 90], [Fou 87], [Him 89], [LMO 90], [Los&Al 91], or in some cases supplying support on conjectures' formulation [Faj 87], [DB 83]. The GReAt environment integrates both features of digraph edition and conjecture's support, dealing with complex objects, the digraph families, defined as sets of digraphs. A multiagent model, the PAC (Presentation, Abstraction, Control) model [Cou 90], inspired on the MVC (Model-View-Controller) approach for user-interfaces development [Gol 84], has been used in order to identify, structure and design the GReAt application. We have chosen to implement the PAC agents following object-oriented technics [Boo 86], [Mey 88], proposing a general implementation schema, which preserves the communication principles of the PAC multiagent model.

GReAt is being developed in C++ [Str 93], at the Universidad Central de Venezuela, on a Unix platform, using X-Window and Motif graphical development tools.

Besides this introduction and the conclusion, this paper is structured in four main sections. Section 2 presents the PAC model, focusing on the systematic process followed to develop an interactive application. Section 3 describes the main functionalities of the GReAt environment. Section 4 presents the architecture of the GReAt environment expressed in terms of PAC agents. Finally, in Section 5 a general structure is given for the object-oriented design of a PAC model based application, showing the C++ class patterns implementing the Presentation, the Abstraction and the Control perspectives of the PAC agents.

2. THE PAC MODEL
2.1. GENERALITIES
The PAC model, a communicating agents based model, allows to structure recursively the architecture of an interactive system. These agents are organized according to three basic formalisms or perspectives:
i. The **Presentation**, defining the image of the system, reflecting its behavior with respect to the user input/output.

ii. The **Abstraction**, defining the concepts and functionalities of the system.

iii. The **Control**, maintaining the coherence and communications between the Presentation and the Abstraction perspectives since these are not allowed to communicate with each other. Moreover, the communications among the PAC agents are only performed by means of their respective controls.

The PAC diagram of the application is recursively defined in a tree-like fashion from elementary PAC agents. Each tree level corresponds to a semantical level of the application. On one hand, the highest level of the tree (root) contains the semantical features of the application; on the other hand, the lowest level (leaves) contains elementary graphical features of the application. PAC is a multiagent based model [Cic 84], [Gol 84], [LVC 89], where each PAC agent is considered as a complete system. Hence PAC adds another dimension, the parallelism, to the existing models used in the construction of interactive systems (language based model [F&D 84], input/output based model [Lan 86]). PAC agents work in parallel (parallelism), and each agent is organized as three parallel processors (micro-parallelism) represented by the abstraction, control and presentation formalisms, with respectively three memories: an abstract state, a control state and a graphical state. The control instructions communicate with the other PAC agents, eventually performing translations, hence satisfying the requirements of these objects and the user's actions by calling the abstraction and presentation processors. As a consequence of its functions, the control may also work as an event recognizer. The instructions of the abstraction processor are functions working on the abstract state memory. The presentation processor contains instructions applied on the graphical state memory to display the objects on the screen and to receive the user's actions.

It has to be pointed out that the main difference between the PAC and the MVC similar approaches is the fact that MVC agents may indistinctly communicate among them, while PAC agents can only communicate through their respective control perspective. The main advantages of the PAC approach are presented below.

### 2.2. ADVANTAGES OF THE PAC MODEL

We can resume the advantages of this model, focusing four main points:

i. It is a systematic methodology applicable to all the abstraction levels of an interactive system.

ii. The control notion is introduced for communicating two different perspectives (abstraction and presentation), and for managing the interaction by means of cooperating local controls, facilitating the communications among agents. The control concept allows to maintain context depending information at different abstraction levels, facilitating general services such as help and histories.

iii. It favors the distribution of semantical and syntactical actions. Particularly, the application may express itself in its own terms, independently from the presentation aspects, allowing the cohabitation of semantically distinct domains. In this way, functions which are usually mixed into the application domain, as for example the editing functions, are now kept only at interface level. These aspects facilitate the system maintenance (extendibility and modifications).

iv. A PAC agent may hold different presentations with the same abstraction.

### 2.3. DESIGN WITH THE PAC MODEL

In what follows we will detail the process used in order to apply the PAC methodology to design the architecture of an interactive application:

i. Formulation of the requirements for the application.

ii. Study of the functionalities of the system with respect to user-interface, on the basis of the requirements formulated in step i.

iii. Identification of the graphical objects and the PAC agents based on the above results. A general PAC diagram of the system is constructed. In parallel with the identification of the PAC agents, the classes implementing the abstraction and presentation perspectives may also be identified.

iv. Syntactical analysis on the PAC diagram, introducing the user external actions, for validating in a certain measure the architecture model. The structure and operations of the corresponding classes may be used in this sense, allowing also the semantical validation of the operations identified in the classes. This process points also out the control functionalities, in order to identify the corresponding classes and operations.

v. Implementation of the classes corresponding to the PAC agents which have been identified in step iii and iv.
Notice that the structure of the PAC agents may be influenced, according to the choices of the platform and the graphical tools. Moreover, PAC does not impose any particular design methodology, but PAC agents seem naturally adapted to be implemented using object-oriented programming techniques (see section 5).

3. THE GREAT ENVIRONMENT
GReAt offers two working modes: the family mode, which is the system default mode and the digraph mode. There are specific functionalities for each mode. In this work we will be dealing essentially with the family functionalities, because it is the main goal and interest of our system.

3.1. GREAT MAIN FUNCTIONALITIES
The main innovation of the GReAt application is to permit the treatment of digraph families at two levels: the syntactical or graphical level, concerned with the edition (drawings) of families and the semantical or conceptual level, supporting the interpretation of the drawings. The family notion is an important unifying concept used in graph theory for formulating theorems and conjectures, but there are no standard notations for handling it. At graphical level, GReAt offers a set of family constructors for edition purposes constituting the Family Description Language [FLM&A 93]. The interactive edition of a family is done using the constructors. At syntactical level, the meaning of the different constructors used is not relevant. Hence, the user is free to build his families, regardless of semantical control. The family semantics is relevant at the moment of analyzing the graphical object produced; an instantiation operation is then introduced at conceptual level. This operation helps to determine if a family is empty. If this is not the case, an instance or digraph representing the family, can be randomly generated. Instantiation plays an important role in the construction of digraphs with given properties. The features of conjecture formulation covered by GReAt are performed by the Agora tool. It performs feasibility tests for deducing or rejecting conjectures, from a specialized knowledge base in the domain of path and circuits in digraphs, with the main purpose of measuring the level of difficulty of conjectures. Agora should complement the specialist's feeling on how "strong" or how "weak" looks the conjecture he is formulating. The Agora user interface functionalities will not be discussed here, since this tool is a complex system that will be presented in future works.

3.2. BRIEF DESCRIPTION OF THE GREAT INTERFACE
The main feature of the GReAt interface is the Family Window for drawing and manipulating digraph families, which is shown when the GReAt application is started. This window presents:

- The Family General Options containing buttons to access the usual file, edition and configuration functionalities.
- The Family Constructors palette, holding the buttons associated to the family constructors and operations.
- The Family Edition, that holds the family drawing area.

In the digraph mode, a digraph window similar to the family window, specific to digraph treatment, will be created. Several family and digraph windows may be simultaneously present on the screen. Figure 1 shows a family constituted by six families. A digraph family is represented by a hierarchy of related rectangular frames, each frame represents a subfamily and the top level frame corresponds to the whole family. For example, in the family denoted by 1.1 we can observe an embedded family, namely family 1.1.1.

435
Figure 1: Example of family drawing and its hierarchy tree.

Besides the family windows and digraph windows, the GREAt window is displayed, showing the GREAt headings and general information such as hour and date of the session. It also holds the family and digraph mode buttons corresponding to the two GREAt working modes and the buttons to access the Tutorial, Help and Agora tools.

4. ARCHITECTURE OF THE GREAT ENVIRONMENT USING THE PAC MODEL

The architecture of the GREAt system is presented in this section making use of the hierarchical diagram of the PAC agents, identified in step iii of the development process (see Section 2.3). According to this process, the first level corresponding to the conceptual domain of the application treats all the aspects relevant to the abstract manipulation of digraphs and digraph families. The system functionalities at user-interface level, discussed in Section 3, must be taken into account in order to establish the PAC inferior levels of abstraction. In what follows we will briefly present the agents of these levels, making reference to Figure 2. Each oval shape in the diagram contains a PAC perspective: abstraction (A), presentation (P) and control (C), in which we place the name of the respective PAC object. The arcs indicate the communication links among the control components of the PAC agents.

4.1. THE APPLICATION LEVEL

This level is constituted by the conceptual domain of the application and it treats all the aspects relevant to the abstract manipulation of digraphs and digraph families. The abstraction of the GREAt agent holds all the application's functionalities related to the treatment of digraphs and digraph families at conceptual level, independently from any graphical representation; it is the conceptual heart of the environment. The control of the GREAt agent allows the communication between the application and the interface levels.
4.2. THE INTERFACE LEVEL

It corresponds to the GReAt user-interface and it is constituted by the following PAC agents:

- **Interface Manager**: the Interface Manager presentation shows the application window which consists of the mode buttons to create a new Family Window, a new Digraph Window, and to access the tools: Agora, Help and Tutorial. The Interface Manager abstraction holds the Family and Digraph Window lists and the Interface Manager control establishes the link between its abstraction and presentation, it manages the Family and Digraph Window lists, and its different child PAC agents.

- **Family Window**: corresponds to a composed agent holding three child agents: the Family General Options menu, the Family Edition drawing area and the Family Constructors palette. There exists a Family Window agent for each family being constructed at a time, and for each Family Window agent there exist a Family Constructors agent and a Family Edition agent.

- **Family General Options**: containing all the file operations, edition and configuration functionalities for the currently edited family.

- **Family Constructors**: corresponding to the palette which is used for building digraph families. It has a presentation constituted by the constructors' buttons and their related menus. The menus allow to select a specific configuration for the constructors which is stored in the abstract memory. The characteristics of the constructor expressed by its configuration will be taken into account in the presentation of the Family Edition agent by means of the Family Window agent's control.

- **Family Edition**: corresponding to the drawing area for digraph families, is the most important and interesting agent of the GReAt interface because it is responsible for the family display and direct family manipulation on the screen. The basic semantics of this agent is the following: it manages the family drawing area. Its abstraction holds the data structure of the family drawing, in order to check the drawing constraints. The operations of select, move and resize are effected directly on the presentation of the Family Edition agent, since they are external events which are produced directly on the family drawing area. Hence, the Family Edition control communicates with its abstraction in order to authorize the operations of move and resize, determining a new location for the family object.
5. IMPLEMENTATION OF THE PAC AGENTS APPLYING OBJECT ORIENTED
TECHNIQUES

We have seen in section 2.3 that the agents of the PAC model may be implemented using object-oriented programming techniques. According to step v, we present a possible implementation of the PAC agents. In our implementation we have considered for each PAC agent, the three perspectives as separate classes (see Figure 3). The advantages of this approach is to enhance maintenance and reuse of the code, facilitating also its manipulation by the programmers' team. Moreover, we have used the tree structure of the Unix file system in order to organize the files of the classes in accordance to the tree structure of the PAC diagram.

Figure 3: Class diagram pattern expressed in Booch notation [Boo94] for the implementation of the PAC agents.

The diagram shown in Figure 4 presents a basic component used in the definition of a PAC diagram, where the oval shapes represent the Abstraction, Presentation and Control perspectives of the PAC agents. The arrows show the only communications allowed by the PAC model.

Figure 4: Communications permitted among the PAC agents and its perspectives.

---

- **Digraph Window** agent: behaves similarly to the Family Window agent. It is a composed agent (Digraph General Options menu, Digraph Constructors palette and Digraph Edition area) for digraphs. It communicates with its child agents, according to the functionalities of the digraph mode.
- **Agora, Help and Tutorial** agents will not be described in this paper.
Notice that on one hand, all the connections must be bi-directional and on the other hand, the communications between the abstraction and presentation of an agent must not be allowed. Hence the notion of inheritance cannot be used to relate the classes implementing the PAC agents. In view of the data heterogeneity involved in the communications, the definition of general communication methods seems difficult. A use relation is then established in order to allow communication among the classes implementing each agent. Therefore, in order to simplify our implementation, we have chosen to implement the required communications by function calls (public methods). Hence, we do not exploit the parallelism (not essential in our application) implicit in the PAC model, avoiding synchronization problems. Taking account of these aspects, we propose the following general C++ class pattern for the definition of the PAC agent:

```cpp
/* Classes' definition */

class Control_W {
   // Class implementing the Control perspective of the PAC agent W.
   private:
      Control_X *x;    // pointer to the Control perspective of the PAC agent X.
   public:
      ... // pointer to the Control perspective of the father PAC agent
      ...f(...);   // Function implementing one of the possible communications.
   ...};

class Control_X {
   // Class implementing the Control perspective of the PAC agent X.
   private:
      Control_W *w;    // pointer to the Control perspective of the PAC agent W.
      Abstraction_X *abstraction;  // pointer to the Abstraction perspective.
      Presentation_X *presentation; // pointer to the Presentation perspective.
      Control_Y *y;    // pointer to the Control perspective of the PAC agent Y.
      Control_Z *z;    // pointer to the Control perspective of the PAC agent Z.
   public:
      Control_X(Control_W *control_w); // Called at initialization of the object Control_X.
      ~Control_X(); // Called at the destruction of the object Control_X (destructor).
      ...f(...);    // Function implementing one of the possible communications.
   ...};

class Abstraction_X {
   // Class implementing the Abstraction perspective of the PAC agent X.
   private:
      Control_X *control;    // pointer to the Control perspective of the PAC agent.
   public:
      Abstraction_X(Control_X *control_x); // Constructor of the object Abstraction_X.
      ~Abstraction_X(); // Destructor of the object Abstraction_X.
      ...f(...);    // Function implementing one of the possible communications.
   ...};

class Presentation_X {
   // Class implementing the Presentation perspective of the PAC agent X.
   private:
      Control_X *control;    // pointer to the Control perspective of the PAC agent.
   public:
      Presentation_X(Control_X *control_x); // Constructor of the object Presentation_X.
      ~Presentation_X(); // Destructor of the object Presentation_X.
      ...f(...);    // Function implementing one of the possible communications.
   ...};

class Control_Y {
   // Class implementing the Control perspective of the PAC agent Y (or Z).
   private:
      Control_X *x;    // pointer to the Control perspective of the PAC agent X.
   public:
      ...f(...);    // Function implementing one of the possible communications.
   ...};
```

439
/* Methods' code */

/* Control perspective's Constructor of the PAC agent X */
Control_X::Control_X(Control_W *control_w)
{
    w = control_w; // Save the link to the Control perspective of the PAC agent W.

    /* Create an instance of the Abstraction perspective of the PAC agent, and save the link to this instance. */
    abstraction = new Abstraction_X(this);
    /* Create an instance of the Presentation perspective of the PAC agent, and save the link to this instance. */
    presentation = new Presentation_X(this);

    Create the instances of the Control perspectives of the child PAC agents, and save the links to these instances. This creation involves by recursion, the creation of the perspectives Presentation, Abstraction and Control of the children of these agents.
    y = new Control_Y(this); // PAC agent Y.
    z = new Control_Z(this); // PAC agent Z
}
/* Control perspective's Destructor of the PAC agent X */
Control_X::~Control_X()
{
    /* Destruction of the class instances implementing the Control perspective of the child agents. This destruction involves by recursion the destruction of the perspectives Presentation, Abstraction and Control of the children of these agents. */
    delete(y); // PAC agent Y.
    delete(z); // PAC agent Z
    /* Destruction of the class instance implementing the Abstraction perspective for the PAC agent X. */
    delete(abstraction);
    /* Destruction of the class instance implementing the Presentation perspective for the PAC agent X. */
    delete(presentation);
    /* Destruction code of the Control perspective (PAC agent X). */
}

Abstraction_X::Abstraction_X(Control_X *control_x)
{
    /* Save the link to the instance of the Control perspective (PAC agent X). */
    control = control_x;

    /* Initialization code of the Abstraction perspective (PAC agent X). */
}
Abstraction_X::~Abstraction_X()
{
    /* Destruction code of the Abstraction perspective (PAC agent X). */
}
Presentation_X::Presentation_X(Control_X *control_x)
{
    /* Save the link to the instance of the Control perspective (PAC agent X). */
    control = control_x;
    // Initialization code of the Presentation perspective (PAC agent X).
}

Presentation_X::~Presentation_X()
{
    // Destruction code of the Presentation perspective (PAC agent X).
}

Where:
- *this* means a pointer to the class instance of the member function ("pointer-to-self").
- *new* is the dynamical memory allocation operator of C++.
- *delete* is the dynamical memory desallocation operator of C++.

This implementation ensures that the PAC model principles with respect to communications among the agent's perspectives and the agents of the PAC diagram, are not violated by programmers. This consideration is reflected in Figure 5, where the f() function implements a communication between two perspectives.

<table>
<thead>
<tr>
<th>From / To</th>
<th>Control of W</th>
<th>Abstraction of X</th>
<th>Presentation of X</th>
<th>Control of Y</th>
<th>Control of Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control of W</td>
<td>Local access</td>
<td>Impossible</td>
<td>Impossible</td>
<td>x-&gt;f()</td>
<td>Impossible</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(com. E2)</td>
<td></td>
</tr>
<tr>
<td>Abstraction of X</td>
<td>Impossible</td>
<td>Local access</td>
<td>Impossible</td>
<td>control-&gt;f()</td>
<td>Impossible</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(com. E12)</td>
<td></td>
</tr>
<tr>
<td>Presentation of X</td>
<td>Impossible</td>
<td>Local access</td>
<td>Impossible</td>
<td>control-&gt;f()</td>
<td>Impossible</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(com. E14)</td>
<td></td>
</tr>
<tr>
<td>Control of X</td>
<td>w-&gt;f()</td>
<td>abstraction-&gt;f()</td>
<td>presentation-&gt;f()</td>
<td>Local access</td>
<td>y-&gt;f()</td>
</tr>
<tr>
<td>Control of Y</td>
<td>Impossible</td>
<td>Impossible</td>
<td>Impossible</td>
<td>local access</td>
<td>z-&gt;f()</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(com. E5)</td>
</tr>
<tr>
<td>Control of Z</td>
<td>Impossible</td>
<td>Impossible</td>
<td>Impossible</td>
<td>x-&gt;f()</td>
<td>Impossible</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(com. E4)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5: Possible direct communications among the different perspectives of the PAC agents shown in Fig. 4.

The creation of the PAC agents of the application is recursively performed, beginning with the PAC agent at the root of the PAC diagram. The Control class of each agent dynamically creates an instance of the Abstraction and Presentation classes. It also creates dynamically a Control class instance of each child agent. Similarly, the destruction of the PAC agents is recursively performed. Moreover, the pointers to the created instances are saved in order to allow the function calls implementing the communications. The declaration of the pointers in the private part of the classes preserves the PAC model principles with respect to communications.

As we have seen, the inheritance is not helpful for implementing the communications among the different PAC perspectives. Nevertheless, multiple inheritance is used to avoid code repetition while constructing the classes constituting the PAC agents (see Figure 6). We have developed reusable PAC agents which are independent from the application's PAC diagram. These agents are used to manipulate each of the commonly used elementary objects (for example a push-button). The classes implementing the PAC agents of the PAC diagram are partially constructed by multiple inheritance from the corresponding classes' perspectives of the independent PAC agents (implementing the elementary objects).
Moreover, in order to respect the PAC model concept, the functions which receive the external user events must be located in the classes implementing the Presentation perspectives of the PAC agents. Nevertheless, since the use of the execution stack of the program is different between the C++ compiler and the X-window system (pointer this implicitly passed to each method of a class in the case of C++), we have defined classes permitting the interface between the X-Window system and the methods which are called by this one (reception of external events). The classes requiring to use the functions of the X-Window system inherit the methods of these predefined classes, hence providing a correct interface.

6. CONCLUSION

We have presented the GReAt application, a graphical interactive environment for researchers and students in the field of graph theory, focusing its user-interface functionalities. Moreover we have proposed a systematical approach for the design of graphical environments based on the PAC model, and a possible implementation of this model using object-oriented programming. This implementation strictly preserves the principles of the PAC model. From this experience, we have observed that the tasks of the programming team can be facilitated, distributed and standardized according to the class pattern provided. Moreover, the proposed organization for the classes implementing the application using the PAC approach, facilitates the code documentation. A first prototype of the GReAt environment has been already completed using our approach. Actually, only the abstraction perspective of the GReAt agent and the agents corresponding to the Agora and the Tutorial subsystems (see Figure 2) are now under development. We feel that the development process with the PAC model discussed here is an important step towards interactive software construction. In the near future we think to improve our development process integrating the use case approach [Jac&Al 92], for building interactive applications.

7. ACKNOWLEDGMENTS

We gratefully acknowledge the creative programming efforts of A. Durán, A. Guia, F. Marchena, V. Roa, J. Infante, O. Dominguez, G. Schoeberg and C. Urrecheaga, in the implementation of the GReAt environment.

8. REFERENCES


[LMO 90] F. LOSAVIO, O. MEZA, O. ORDAZ. "Un AMbiente interactivo de trabajo para un investigador en teoría de Dígrafos ADMI". Acta Científica Venezolana, Enero 90


Keywords: Interface design, Model, GraphEd.
Francis LOSAVIO received the Doctor degree in Computer Science in 1991 and a 3ème. Cycle Doctor degree in Computer Science in 1985 from the Univ. Paris-Sud, Orsay, France. She also received a MSC degree in Computer Science from the Univ. Simón Bolívar, Venezuela in 1983. At present, she is a Titular Professor at the School of Computer Science, Faculty of Science, Univ. Central de Venezuela and coordinates the ISYS Research Center. Her research includes software engineering environments, methodologies for software development, automatic program construction and formal specifications.

E-mail: flosavio@conicit.ve.

Alfredo MATTEO received the Doctor degree in Computer Science from the Univ. Paul Sabatier, Toulouse, France in 1984. At present, he is an Aggregate Professor at the School of Computer Science, Faculty of Science, Univ. Central de Venezuela and is member of the ISYS Research Center. His research includes software engineering environments, methodologies for software development, formal specifications and algorithmic complexity.

E-mail: amatteo@conicit.ve.

Oscar ORDAZ obtained the Doctor degree in Algebraic Geometry from the Univ. Paris-Sud, Orsay, France in 1973 and a MSC in Computer Science from the Univ. Simón Bolívar, Venezuela in 1983. At present he is a Titular Professor at the School of Physics and Mathematics, Faculty of Science, Univ. Central de Venezuela, and is member of the ISYS Research Center. His current research is the graph theory.

E-mail: oordaz@conicit.ve.

Oscar MEZA received the Doctor degree in Operation Research form the Univ. of Grenoble, France in 1983. He is an Associated Professor at the Computer and Information Science Department, Univ. Simón Bolívar, Venezuela. His research interests include programming environments, software engineering environments and methodologies for software development.

E-mail: meza@usb.ve.

William GONTIER received the Maitrise in Computer Science form the Univ. Paris Nord, France, in 1991 and the DEA in Computer Architecture from the Univ. Paris-Sud, France, in 1992. At present working at the ISYS Research Center within the PCP-Informatics program of the CEFI-CONICIT. His research interests include programming environments, software engineering environments and methodologies for software development.

E-mail: william@lri.fr.