Atlas, a platform for distributed graphics applications

M. Fairén and À. Vinacua

Department of Software
Institute of Robotics and Industrial Informatics
U.P.C.
Diagonal 647, 8^{ena} planta
E08028 Barcelona, Spain
{mfairen,alvar}@turing.upc.es

Abstract: ATLAS, a platform for developing distributed applications by splitting them into several collaborating processes scattered in a local area network is presented. Although of general use, it has features especially designed for supporting graphics applications. We present its architecture and some aspects of its implementation, and discuss design criteria.

1 Introduction

ATLAS is a platform for the development of advanced interactive graphics applications that distributes the application processes in a network. Its purpose is to make some techniques that would require a lot of specialized programming available to every application built over it with the least possible hassle for the programmer. These techniques include the distribution of the application processes (problems involved in distribution were focused in [1] and [2]); a journaling mechanism allowing the final user to do arbitrary replays of his session; fault-tolerance (see [3], for instance) in case of communication failures of the network or some of the processes; a powerful control language allowing the developer to describe his application and aspects of its user interface and allowing the final user to introduce his own macros; and mechanisms for parametric and constrained-based design. This platform can also be extended to support CSCW in applications developed over it as we will explain in section 5.

ATLAS is an evolution of a system described in [4]. It inherits some aspects of the architecture of the previous one, and adds robust and fault-tolerant network distribution transparently, a meta-journaling system, a much more flexible control language and an orthogonal design which affords much more flexibility to the applications built in it. In section 2 we can see an overview of the ATLAS architecture that also explains its main components.

Each process involved in an ATLAS application is described by its interface defined in a module written in ATL language (the ATLAS control language [5]), and its implementation (written in C++) which defines the external routines described in the ATL module. These processes, then, are seen by ATLAS as objects, but these objects don't support several capabilities offered by Object-

Oriented architectures (like inheritance, for example). We will explain it in more detail in section 3.

Atlas applications often dissociate the input processes from the others. This dissociation is desirable in order to achieve the maximal reusability, but it is not compulsory (some modules require very special interface elements that are usually incorporated into the same process – e.g. virtual reality applications). In computer graphics applications the meaning of an input data is only known by the process dealing with it and to make the system aware of this information is a tedious and unnecessary work. Section 4 explains the mechanism that Atlas uses to solve this aspect, arising from data that interactive graphics applications usually manage. This problem may not be aparent at first glance but is extremely important.

2 Overview of Atlas architecture

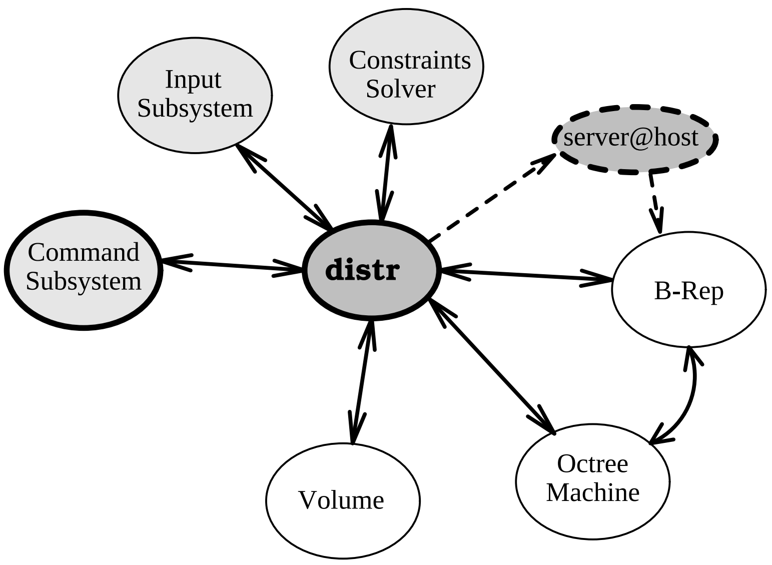

Fig. 1. A sample execution of an Atlas application.

The Atlas architecture is represented by figure 1, where the ovals denote processes and the arrows represent communications between them. It is a centralized architecture where the process `distr` acts as the master process and is the center of each Atlas application. This architecture allows an intelligent distribution to be managed, i.e. the `distr` process decides the processes distribution dynamically depending on availability, load and aptitude of each host in the network to run each application process.

Figure 1 shows a typical ATLAS application. The processes depicted with a thick line represent the main components of ATLAS. All the others are regarded equally by the system, and they don't need to know about each other.

Of these three main processes, the most crucial one is the master process `distr`. This is the process that the user starts up to invoke the application. It acts as a communications center for the duration of the execution, and provides some essential services to all other modules. It is also responsible of the *journaling* mechanism, the *fault-tolerance* of the system and the central mechanism to assign an input datum, provided by an input system, to the corresponding request, normally issued by another process.

The process `server` is a daemon that runs on all hosts configured to run ATLAS applications in the network. A user can select a specific list of hosts via variables in his environment, or else ATLAS attempts to use all resources configured by the administrator (depending on their load). Each time a new process needs to be loaded and connected to the rest of the application, `distr` connects to the `server` on the chosen target machine and requests that such a process be started for him. `server` then forks a copy of itself, makes the appropriate verifications, loads the adequate environment and execs the desired process. Figure 1 only shows one such `server` for legibility, although one such server will be running on every node available to ATLAS.

The third of the ATLAS main processes is the `command subsystem` which guides the application behavior by interpreting programs and instructions written in a language (ATL) designed for ATLAS and described in [5]. When the `distr` is started it starts the `command subsystem` and feeds it the initialization file which determines which application is being started. This initialization file includes instructions to load all the (initially) necessary modules (a module is a file written in ATL language which describes the interface of a process –see section 3), and also defines commands adequate to that application.

Other two standard components offered by ATLAS are the `input subsystem` and the `constraints solver`. The first one is a generic input handling process. It provides a window in which all the textual interactions occur (issuing commands or entering numerical data), but can also be instructed to capture events from other windows (owned by the rest of the processes in the application), and it is also an interface between ATLAS and an extended Tcl/Tk [6] engine, so scripts in Tcl can be sent to it to instantiate new interface components. The constraints solver rely on the global identifiers assigned to external data by the applications (see section 4) to convey information on the appropriate new values that entities within the system should take to the true owners of those data. This is the only mechanism where a "sort-of-pass-by-reference" paradigm is used (the global identifiers acting as a kind of reference to the concrete datum). ATLAS uses elsewhere a "copy-in copy-out" parameter passing convention, as opposed to applications supported on CORBA (see [7]), for example. In this way, objects are managed by the user applications directly, and data is shared only between applications that agree on their type. We have found this scheme to better support the applications we intend to develop, and to favor the portability of modules from one application to another, as they are more loosely coupled. Although type agreement is required, the modules become totally encapsulated and independent. The generic input system can input data that will be used to construct those objects without further knowledge of them, and in the applications considered, less network traffic is produced.

3 Processes as heavy objects

Each user process in an ATLAS application can in fact be seen as a "heavy" object. The ATLAS module describes its interface, whereas the process itself provides an implementation. The parallel cannot be pushed all the way through. These objects cannot be instantiated (in the present version), and more importantly, the main watermarks of object orientation are missing: there is no scheme for inheritance, no scheme for delegation, and no polymorphism. However, they do provide a very strong encapsulation, and each defines a very precise collection of messages to which it shall respond.

They are also heavy, as seen on the example of section 2, because they represent not the basic building blocks of the application but rather complete components with very sophisticated behavior, like a complete B-Rep modeler. The proposed scheme would not be efficient at a finer granularity, but seems to fare rather well in this setup.

Our experience so far is positive also in that transforming other user applications to this environment is usually quick. A prototype solid modeler with ca. 15.000 lines of code was attached to this platform in a couple of days' work, for example.

A benefit of this architecture is that intricate and large data structures managed by an application remain strictly local to it, and others need not know the details of them to deal with them. When this data must travel through the net, ATLAS provides an external version of them, deprived of methods and local information (as ATLAS's language is not itself object oriented). The development platform provides code to automatically transfer the user's data into ATLAS' variables, and backwards, through *bridge types* used to isolate the user from the details of ATLAS's variables (which an advanced user can use directly if he wishes to).

Figure 2 shows a portion of a module's interface definition in ATL. The `USE` clause at the beginning of the file denotes a dependency. When the ATL compiler sees it, it compiles a module called `se.atl` if it has not already done so. Towards the end of the example the procedure `se::Output` —defined in that module— is invoked. The compilation of `se.atl` will have introduced the corresponding entry in the system's symbol table for it to know what to do at this point.

In the interface definition, types functions and procedures preceded by `EXPORT` are visible to other modules, and the rest are local. Functions and procedures listed in the `PROT` section with an `EXTERN` modifier denote user routines exported by an ATLAS process. If an `.atl` file includes `EXTERN` procedures or functions,

```
USE se;
           ...
EXPORT #deftype Solid
   STRUCT
      faces_number -> integer;
      vertices_number -> integer;
      faces -> Listfaces;
      vertices -> Listvertices;
      material -> Material;
      label -> type_identifier;
   ENDSTRUCT
           ...
PROT
  EXTERN FUNCTION SolidIntersection (Solid a, Solid b) RETURNS Solid;
  EXTERN PROCEDURE DisplaySolid (Solid s);
  EXTERN FUNCTION IsVertexOrEdge (BRep_Button_pressed_event ev,
                                  Solid &s) RETURNS boolean;
           ...
ENDPROT
           ...
FUNCTION GetSolid (BRep_Button_pressed_event ev) RETURNS Solid IS
  integer i; i = 0;
  Solid solid;
  BRep_Button_pressed_event eaux;
  WHILE ((i<MaxAttempts) && (!IsVertexOrEdge(ev,solid))) DO
    eaux = GETDATA ("Click on a vertex or edge!");
    i = i + 1;
  ENDWHILE
  RETURN solid;
ENDFUNCTION

EXPORT PROCEDURE Intersec2Solids () IS
  DisplaySolid (SolidIntersection(
                GetSolid(GETDATA("Press button on a vertex or an
                                  edge of the first solid")),
                GetSolid(GETDATA("Press button on a vertex or an
                                  edge of the second solid"))));
  se::Output ("Intersection completed","m");
ENDPROCEDURE
```

Fig. 2. Portion of the interface definition in Atlas for the BRep modeler process.

then it is regarded as the interface definition of a process by the same name, and the command system requests from `distr` that the corresponding process be started. Functions and procedures like those in this example, written in ATL itself, are usually used to define the dialogues with the user or other aspects of the interface. In that area the lower efficiency of Atlas's interpreted code is largely outweighed by the ease of development and testing, plus the flexibility.

When building an application, the programmer will hand out these interface definition files to Atlas for it to construct appropriate drivers for each of the user's processes. Figure 3 shows how this is done. Files in the leftmost column

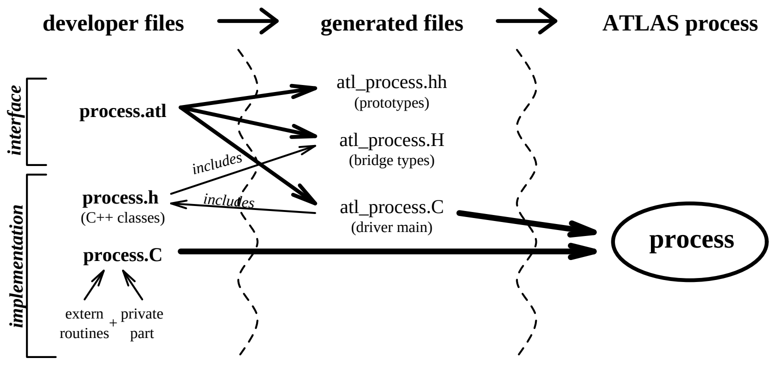

Fig. 3. Building an ATLAS process.

are written by the programmer, and ATLAS generates automatically those in the middle column. Figures 4 through 6 show portions of the automatically generated code stubs to link with the user code, generated from the interface code in figure 2.

The file `atl_BRep.H` (in Figure 4) includes the definitions connected with the *bridge type* corresponding to `Solid`. Notice how methods to convert to and from objects of type `Variable` are built, and the use of *atl_Solid* in figure 6 to isolate the user's class (*Solid*) from its ATLAS external representation.

4 Global identification of data

In text-oriented applications the input data that the user introduces have directly the correct meaning for the application, but in interactive graphics applications this is not so. The user frequently uses *points* to input data that must be changed to other geometric entities by the application, e.g. selecting a vertex, an edge or a face of a previously defined object. Moreover that *point* refers normally to an extremely unreliable reference. It is referred usually to the device coordinates of the selected *pixel* and therefore it depends on the current visualization and the size of the window where the pixel has been selected.

This is important because this input data must be recorded in the *journal* to be reused if it is needed. And simple configuration changes (resizing of windows, for example) may cause the absolute pixel address to mean something different to the same process in a different execution.

ATLAS solves this problem with a very simple mechanism. Input data are attached a unique tag used to identify that datum globally, and both tag and datum are recorded in the *journal* together. (A similar mechanism but with a different meaning and motivation is proposed in [8]. Tags are used there to define topologic relationships in order to be able to compute the results in a model where its parameters have been changed).

```
     ...
namespace brep {
struct atl_Solid {int faces_number; int vertices_number; Listfaces faces;
                  Listvertices vertices; Material material;
                  type_identifier label;
atl_Solid() {}
atl_Solid(Variable &v) {
   if (v.Tree()==NULL) atl_exit(-1); // Invalid variable
   faces_number = ((nodeint *) (*(v.Tree())).accede(0))->Getvalue();
   vertices_number = ((nodeint *) (*(v.Tree())).accede(1))->Getvalue();
   faces = ((nodelist *) (*(v.Tree())).accede(2))->Getvalue();
   vertices = ((nodelist *) (*(v.Tree())).accede(3))->Getvalue();
   material = ((nodeint *) (*(v.Tree())).accede(4))->Getvalue();
   label = ((nodeint *) (*(v.Tree())).accede(5))->Getvalue();
   }
operator Variable() {
   Type t("brep::Solid","S(faces_number integer,vertices_number integer,
                          faces Listfaces,vertices Listvertices,
                          material Material,label type_identifier)");
   Variable v(t,"");
   v.build_tree();
   *((*(v.Tree())).accede(0)) = faces_number;
   *((*(v.Tree())).accede(1)) = vertices_number;
   *((*(v.Tree())).accede(2)) = faces;
   *((*(v.Tree())).accede(3)) = vertices;
   *((*(v.Tree())).accede(4)) = material;
   *((*(v.Tree())).accede(5)) = label;
   return (v);
   }
   };
}
      ...
```

Fig. 4. Portion of the generated atl_BRep.H file.

```
#ifndef __ATL_brephh__
#define __ATL_brephh__

#ifndef NOHEADER
#include "brep.h"
#endif
#include "atl_brep.H"
Solid SolidIntersection(Solid,Solid);
void DisplaySolid(Solid);
bool IsVertexOrEdge(BRep_Button_pressed_event,Solid &);

#endif
```

Fig. 5. The generated atl_BRep.hh file.

```
        ...
   void aux_IsVertexOrEdge(String codi,DLList<Variable *> &parameters) {
      Pix p=parameters.first();
      atl_BRep_Button_pressed_event ptp0(*(parameters(p)));
      BRep_Button_pressed_event par0(ptp0);
      parameters.next(p);
      atl_Solid ptp1(*(parameters(p)));
      Solid par1(ptp1);
      parameters.next(p);
-->   bool res=IsVertexOrEdge(par0,par1);
      Variable *vr=new Variable(restp);
      ReturnValue *rv=new ReturnValue(codi,vr);
      distrib.send(rv);
      ptp1=par1.conversion_to_bridge_type();
      Variable *rp1=new Variable(ptp1);
      ReturnParam *retpar1=new ReturnParam(codi,rp1);
      distrib.send(retpar1);
      }
        ...
   void main(int argc,char **argv) {
        ...
      ini_process();
      driv.Dispatch();
      close(CANAL_COMUNIC_DISTR);
      exit(0);
      }
```

Fig. 6. Portion of the generated atl_BRep.C file. The arrow has been added pointing to the point where user code is actually invoked.

The ATLAS API offers some routines which help the developer to use this mechanism in the easiest way. For example, an application process can ask for some of these tags (new tags) to be attached to its own data, or the process receiving this input datum can request that an annotation be made in the *journal* that his interpretation of that datum corresponds to some other (previously obtained) datum. This new tag will be used in replays of the *journal* instead of the datum, speeding up replays and making them more robust. The effort the process must do to tell the system (`distr`) this association of tags is minimum, and also minimum is the information that goes through the network.

This simple global identification mechanism (described in more detail in [9]) achieves a total robustness in the reexecution of the *journal*, and is also useful internally to control the consistency of this *journal* after being edited and modified. Other important utility of this mechanism for ATLAS applications is the ability to share data identity among several processes in the application (for example with a constraints solver).

Following an example, the simplicity of using this mechanism and also the achieved level of transparency for the developer can be seen clearly:

Figure 7 shows the wrapper classes interface for ATLAS input data which manage the association of a datum and its tag.

```
class io_base {
  protected:
    long ticket;
    char id_type[MAXNAME];
  public:
    long TK ()
      { return ticket; }
      ...
};
```

```
template <class T>
class io : public io_base {
    T datum;
  public:
    io ()
      { ticket = 0;
        id_type = atl_type_name(T); }
    io (T d)
      { ticket = atl_ask_for_ticket();
        id_type = atl_type_name(T);
        datum = d; }
       ...
 };
```

Fig. 7. Wrapper classes interface for ATLAS data

We want to implement a routine to select an edge from a 3D point (if this point is near the edge). If we design the classes *"Point3D"* and *"Edge"* as in figure 8 and considering we have an array of edges as:

```
io<Edge> Table[MAXEDGES];,
```

we can implement the routine *"do_anything"* (figure 9) taking a *"Point3D"* as a parameter and searching for the corresponding edge in the array. If the corresponding edge is found, the routine asks the system to make an annotation in the *journal* to replace the *"Point3D"* tag for the selected *"Edge"* tag for future reexecutions.

```
class Point3D {
    float x,y,z;
  public:
     Point3D (float xx, float yy, float zz)
       { x = xx; y = yy; z = zz; }
           ...
};
class Edge {
    io<Point3D> &fvert, &svert;
  public:
     Edge (io<Point3D> &fv, io<Point3D> &sv)
        { fvert = fv; svert = sv; }
        ...
};
```

Fig. 8. Interfaces of classes "Point3D" and "Edge"

In order to reexecute the *journal* successfully the routine *"do_anything"* must also accept an *"Edge"* tag as a parameter. This can be done in two ways:

– using overloading with this routine and passing as a parameter an *"io_base"* reference which contains the ticket for the *"Edge"*

```
        void do_anything (io_base &id1)
```

```
void do_anything (io<Point3D> &input)
{
  int i=0;
  while ((i<N) && (!near_edge (input, Table[i])))
    { i++; }
  if (i<N)
    { atl_substitute_ticket (input.TK(), Table[i].TK());
      Compute_with_edge (Table[i]);
  else error ("there aren't edges near that point");
}
```

Fig. 9. Routine searching the edge to compute with it

```
{ Compute_with_edge (search_for_edge(id1)); }
```

– using a routine with a generic parameter and calling always this one. This possibility is even more flexible because we can use it also passing an edge if the application allowed the user to input edges. (figure 10)

```
void generic (io_base *arg)
{
  if (strcmp (arg->Type(), "Point3D") == 0)
    do_anything ((io<Point3D>) *arg);
  else if (strcmp (arg->Type(), "") == 0)
         Compute_with_edge (search_for_edge(*arg));
       else if (strcmp (arg->Type(), "Edge") == 0)
              Compute_with_edge ((io<Edge>) *arg);
            else error("type error");
}
```

Fig. 10. Routine with a generic parameter

5 Extension for CSCW

The architecture of ATLAS lends itself easily to a simple approach to transparently implementing primitive means of computer supported collaborative work with ATLAS applications. Its attractive lies especially in the transparency. Once an application has been built upon ATLAS, it can be turned into a CSCW-supporting application by running a copy of it for each user, but with additional connections established between the `distr` of one process (the master session) with those of the rest (the slaves)—see figure 11. By forwarding the relevant messages between these processes, they can be made to share the same data and state, although slave processes are granted access only to limited commands, by decision of the owner of the master session.

In this scheme, the user processes attached to each of the running ATLASes ignores if it belongs to a master or to a slave session. All it sees are messages

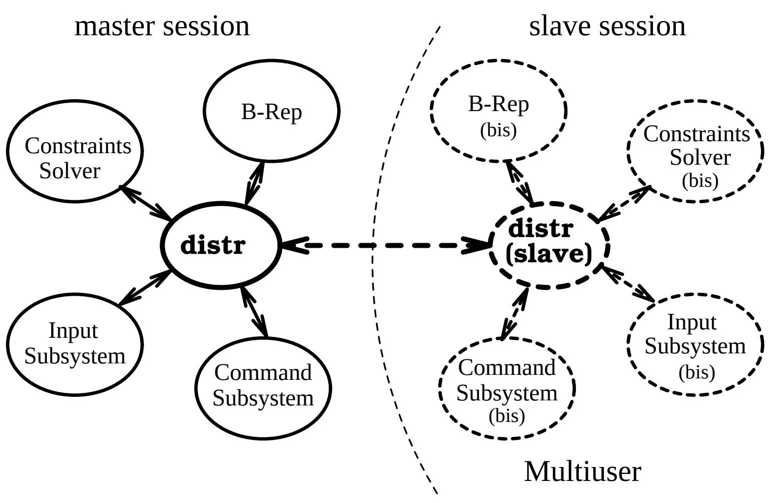

Fig. 11. ATLAS scheme for multiuser architecture

from its copy of `distr` which it heeds, be they generated by user interactions or by synthetic messages imported from a master session. Therefore the developer of an ATLAS application need not be concerned with the collaborative side. All he needs to do is decide whether this behavior of `distr` will be allowed or not for his application.

Different users may change local aspects of the session, like the layout of the windows, or even the point of view from which the scene is rendered. The global identification of data covered in section 4 will ensure that messages across the collaboration links are understood uniformly by all parties.

Upon this primary structure, we purport to build a simple collaborative environment. It remains to design mechanisms for displaying on each party's screen the activity of the other parties, plus mechanisms for controlling ownership of an execution (who is the master), validation of newcomers (how to ensure that they are who they say they are before granting the connection) and to delegate (i.e. turn one of the slaves into the new master). Other tools for such an environment will probably need to be developed as ATLAS modules, for example a whiteboard tool, and even some sort of teleconferencing support. This is a direction in which we will extend the features of ATLAS once it is consolidated as an application development tool.

6 Conclusions

The work presented here is part of an ongoing effort. The components presented here are already in a test phase, and several applications are presently being built upon it at our labs. Other components not described here (like the meta journal) are still in a development stage. The presentation omits also some relevant aspects like the architecture of the virtual machine executing the ATLAS code, which have an impact on the system's features but are not within the focus of this conference.

The architecture of ATLAS —where users add components as independent processes with an external interface visible essentially as a remote procedure call— is proving effective in the construction of applications and in the reusability of those components between similar applications.

7 Acknowledgements

ATLAS development has been greatly facilitated by the use of the ACE-Wrappers (see [10]) library to build the communications drivers, and also by the PCCTS (see [11]) compiler construction tool.

This research has been supported partially by grants TIC-92-0605 and TIC-95-0630-C05-01 of the CICYT

References

1. Gregory R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1), March 1991.
2. Roger S. Chin and Samuel T. Chanson. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1), March 1991.
3. Shrivastava, Mancini, and Randell. The Duality of Fault-Tolerant System Structures. *Software Practice and Experience*, 23(7), July 1993.
4. Antoni Soto, Sebastià Vila, and Àlvar Vinacua. A Toolkit for constructing command driven graphics programs. *Computer & Graphics*, 16(4):375–382, 1992.
5. Marta Fairén and Àlvar Vinacua. ATLAS. Sistema de Comandes: Manual tècnic (in Catalan). *Report LSI-95-11-T*, 1995. http://www.lsi.upc.es/~ mfairen.
6. Brent B. Welch. *Practical Programming in Tcl and Tk.* Prentice Hall PTR. Upper Saddle River, New Jersey 07458, 1995.
7. Jon Siegel. *CORBA Fundamentals and Programming.* Jon Siegel. OMG, 1996.
8. Jiri Kripac. A Mechanism for Persistently Naming Topological Entities in History-Based Parametric Solid Models. In Chris Hoffmann and Jarek Rossignac, editors, *Third Symposium on Solid Modeling and Applications*, pages 21–30, Salt Lake City, Utah, May 1995.
9. Marta Fairén and Àlvar Vinacua. Interacción Gráfica en ATLAS (in Spanish). In *Proceedings of CEIG'97*, 1997.

10. Douglas C. Schmidt. The ADAPTIVE communication environment: Object-oriented network programming components for developing client/server applications. In *12th Sun Users Group Conference*, 1994.
11. Terrence J. Parr. Language Translation Using PCCTS and C++ (A Reference Guide), June 1995. Address: http://www.parr-research.com/ parrt.