ATLAS: a platform for transparently developing distributed applications

M. Fairén and À. Vinacua
Department of Software,
Institute of Robotics and Industrial Informatics, U.P.C.
Diagonal 647, 8ena planta
E08028 Barcelona, Spain
+34 3 401 6739
{mfairen,alvar}@lsi.upc.es

ABSTRACT
We discuss the design and implementation of a software development platform that allows unsophisticated programmers to include advanced features to their applications with no or very little extra information and effort. These features include the splitting of the application in distinct processes that may be distributed over a network, a powerful configuration and scripting language, and several tools including an input system to easily construct reasonable interfaces. We attempt to describe both the techniques used to achieve transparency for the programmer and what exactly a user must do to build new ATLAS modules.

KEYWORDS
Distributed applications, Software development tools, External data representation

1 INTRODUCTION

ATLAS represents an attempt to facilitate to programmers in our lab the incorporation to their applications of certain services at a minimum cost. In fact, ATLAS helps them build distributed applications (problems involved in distribution were focused in [2]), provides them with powerful configuration mechanisms and a macro language and include a journaling facility, among other things. All these aspects require a substantial amount of familiarity by the programmer with the said techniques, and a fair amount of programming for them on each application.

In a research lab like ours a large amount of software is constantly being developed to demonstrate ideas, but eventually has to be rewritten when certain functionalities need to be combined. It also remains in an unfinished status and is usually hard to use or demonstrate by anybody except the author, because of the high cost of "finishing up" the details. We then realized that a development platform that facilitated the construction of this software giving easy access to many sophisticated resources and establishing basic rules that eased the joint operation of modules was valuable to us, since it enabled experimental software to grow in a more harmonious way, and facilitated even the reuse of old components.

Evolving from the ideas originally presented in [9] we designed ATLAS for this purpose. Since our work is centered in Computer Graphics applications we included certain mechanisms especially designed for that domain which we've discussed in specialized forums [6]. However by far the largest portions of effort and of code involved very general tools usable by interactive applications in most other domains. Therefore ATLAS can also be viewed as a general programming tool that greatly simplifies the construction of fairly sophisticated applications.

In this paper we shall present a view of ATLAS from this last perspective, discussing the general aspects not present in other publications about it and showing how and in which ways it can enhance the software development process. In the next section we shall enumerate more concrete objectives and criteria used in the design of ATLAS. Section three will briefly present the solution adopted for the interprocess communication, and section four will describe aspects of the design and implementation of an ATLAS application from the point of view of the user —the programmer building the application. Finally we will turn briefly to conclusions and future work.

2 OBJECTIVES AND DESIGN CRITERIA

The first priority in the design of ATLAS has been to make its inner workings as transparent to the user as possible. To this end, some aspects may not have the intrinsically best or most powerful or most flexible solution, but users can build applications on ATLAS without almost any concern about it, yet getting substantial benefits from its presence.

In terms of functionality, ATLAS includes these objectives:

- *Low level of parallelism.* ATLAS applications feature several distinct processes running concurrently in the same or different machines in a network; processes encapsulate ATLAS components or user modules. The user implements routines that are

accessible to other processes as remote procedures.

- *Interprocess communication.* Since the application is split in several processes, these need to communicate over the network and exchange data between possibly different architectures. This gives rise to many different problems that need to be addressed [1].

- *Standardized input model.* A great deal of effort in an application's development is spent in its user interface. ATLAS provides a uniform but flexible view of inputs that allows many different dialogue modes in a uniform way, somewhat related to Abstract Data Views [3].

- *Configuration and macro language.* A flexible yet powerful way must be provided for a programmer to describe what is in each of his modules, and how it should relate to others, and also to define the dialogues of the application and its behavior in a simple way.

- *Journaling mechanism.* This is not yet implemented and only mentioned here in passing. It is partially designed and will be at the heart of other services offered by ATLAS.

- *Fault tolerance.* Since the application is spread out among several hosts, it becomes more exposed to transient or permanent failures (of the communications or of any of the hosts involved). Fault tolerance is transparently provided based on the journaling mechanism and on heartbeat messages sent by all processes so that their status can be assessed.

- *Reusability.* Each user module is completely isolated from others (in a separate process) except through a well defined interface described in ATLAS's programming language. Thus new components can incorporate and use reliably old ones.

Not all these aspects can be discussed here, as they would take an inordinate amount of space. The following sections center around aspects of the present beta version concerning the housekeeping of processes and communications, and how ATLAS helps users build applications with these features easily.

3 IMPORTANT ISSUES IN THE ATLAS COMMUNICATIONS MECHANISM

3.1 Problems That Need To Be Addressed

Since ATLAS first priority is to offer the maximum transparency to the developer, the design of ATLAS architecture must hide the intricacies of the interprocess communications from the programmer.

The communications mechanism has to address problems like how to start a process in an application, how to manage the interchange of information between processes and also how to detect failures in the application communications in order to know when the *fault-tolerance* mechanism should be activated.

To start a process in a different machine it is necessary to have a process listening for a connection in the chosen host (this requires to have knowledge of low level connections in the operating system), and it would be also desirable that the process to be started inherits the application environment.

The interchange of information between processes has also an added difficulty when they are running in an heterogeneous network because data can be interpreted with different meaning depending on the architecture where they are used.

Next subsection explains how these problems have been addressed in ATLAS.

3.2 The Approach Used In ATLAS

The ATLAS architecture is represented in 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 is able to decide the processes distribution dinamically depending on the aptitude of each host in the network to run each application process.

Therefore, this master process is the most crucial one in this architecture and also in the communications mechanism because it is the communications center for each application. It is also the one to take care about the status of each process in execution at any time. This is easy because of the *heartbeat* mechanism designed in ATLAS.

The *heartbeat* mechanism makes every process being executed in the application send a short message periodically to the master process giving the required information to control the global status of the execution. This mechanism is very useful to detect if a process or the communication with it fail, therefore it will become necessary to the *fault-tolerance* mechanism which, although it is not available yet, has been almost completely designed. This mechanism will be shortly discussed in the future work section below.

Server Process Design
The ATLAS server process is a simple but very important process in the ATLAS architecture. Its role, in the

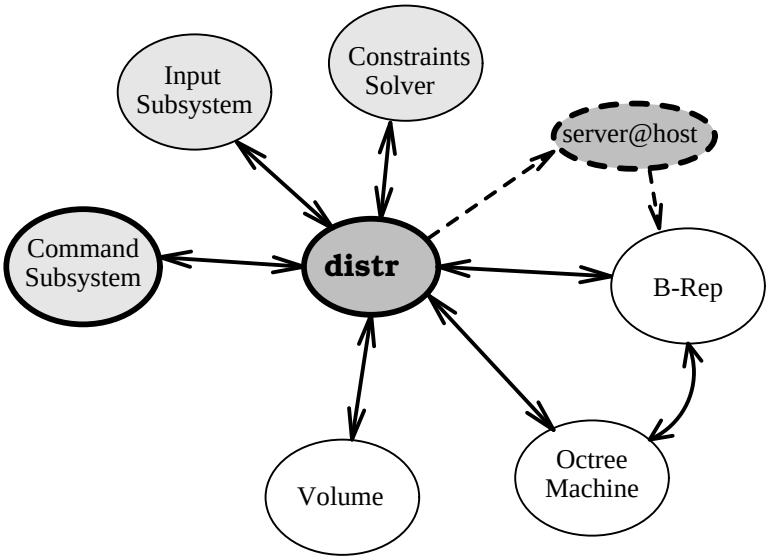

Figure 1: A sample execution of an Atlas application.

current version, is to accept connections from the master process `distr` and run the application process requested by it. In future versions it will also implement a handshake with `distr` to achieve a certain degree of load-balancing.

As the server daemon is normally run at boot time with *root* permissions, it must change its permissions to the user's permissions before executing the user's process in order to protect the system. Moreover, to do that it must be sure of the identification of the user and also inherit the user environment to achieve a successful execution. This environment information is sent by the `distr` process in the connection message. The user identification mechanism is based on a special 32 bytes long user identifier that every Atlas user must have generated before he executes Atlas for the first time.

Communications Drivers Design
An Atlas application process execution is based on a *remote-procedure-call*-like paradigm. A process can be then considered as a set of routines to do the process related work plus a communications driver to manage the interchange of messages with the rest of the application (see section 4).

The communications driver is the main program of the process, and its role is to listen requests or messages from the master process and other connections added and dispatch them as needed. The most relevant requests or messages sent by the master process can be related with a routine call, data answering a request, or an Atlas event notification (see section 4.3).

The main execution of an Atlas process is then the dispatch routine of its communications driver. It enters in a loop listening in the group of channels actived by this process (the default is only the connection with the master process), and when it receives a message it dispatches it and keep on listening for another one (see figure 2). Apart from dispatching messages the driver can also dispatch interruptions (like signals). In fact the *heartbeat* mechanism uses the SIGALRM interruption.

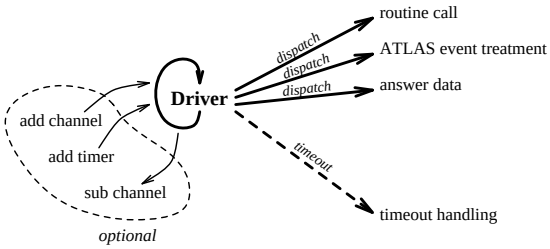

Figure 2: Driver role scheme.

The default treatment of an Atlas process driver is managing the messages comming from the master process and the interruption of SIGALRM. But the actual driver is much more flexible (as can be seen in figure 2). It also allows the process to add channels to listen to, add a timer treatment or remove channels added before. This flexibility only requires to have implemented the treatment for messages being received by these new channels.

The Atlas communications mechanism is implemented using sockets, and its implementation takes profit of the wrapper classes for sockets offered by the public domain package ACE_Wrappers (see [7] and [8]).

The biggest problem to solve in communications between ATLAS processes is the interchange of data. Since these processes may be running in different architectures, data must be sent through the network using a standard representation in order that they have the same meaning to the different processes.

Furthermore, these data need to be –at least very often– accessible to ATLAS itself, which includes a programming language to define user-machine dialogs or otherwise interconnect to different modules (see section 4).

The problem of actually transfering the data robustly has long since been solved. Indeed we just rely on XDR [10] for that purpose. To attain these data sharing with the maximum of transparency for the developer, ATLAS includes a mechanism based on our data structures (called `Variables`) used to wrap user data in each process (this mechanism is thoroughly explained in [5]). These structures offer access methods used by the interpreter of the command language, and also encapsulate methods to encode and decode XDR streams transparently. Using this mechanism and adding also the automatic code generation explained in the next section, the developer needs not be aware of XDR and indeed almost not be aware of our interchange method at all.

4 DESIGNING AN APPLICATION. BENEFITS TO THE DEVELOPER.

4.1 Transparency To The Developer

The most important aim that ATLAS wants to achieve is a high level of transparency to the developer. Almost all services that ATLAS offers to applications developed over it are techniques that normally require a lot of specialized programming. The emphasis in ATLAS's design is to relief the programmer from this effort.

Many facilities offered by ATLAS, like distribution or the communications mechanism, must be totally managed by ATLAS in order to achieve the maximum transparency to the developer. The programmer need not know about these mechanisms and can concentrate on the objectives of his application.

Automatic Code Generation
The ATLAS process communications require quite a bit of code in each process devoted to handshaking with `distr`, generating the *heartbeat* messages at the adequate rate, preparing the arguments for process routines or collecting results and encoding them for being transported over the network, and dispatching calls to process routines. To handle this, ATLAS automatically generates code stubs that the developer must link with his program. These stubs are constructed from the interface declaration of the process (like in figure 3), which contains the type definitions used for variables to be exported and the prototype definitions of the process extern routines.

The generated code also includes stubs to automatically transfer the user's data into ATLAS `Variables` and backwards, through *bridge types* used to isolate the developer from the details of the ATLAS `Variables` (which an advanced programmer can use directly if he wishes to).

The *bridge types* are used to build intermediate objects with the data structure of the process objects (as per their ATLAS declarations) but without the methods of the process objects (which remain unknown to ATLAS). Each *bridge type* has also methods to translate to and from ATLAS `Variables`, making both translations transparent to the developer. The only burden on the developer is then to provide his classes with conversion methods to and from these *bridge type* objects which is usually trivial (unless the developer choses to have a very different structure for the ATLAS data that the one used internally by his program).

As an example we can see some relevant portions of this automatically generated code in appendix.

4.2 Design Process Of An ATLAS Application

An ATLAS application is a set of processes which can communicate between them through the ATLAS communications mechanism. Each process can be seen as a module offering some public methods that can be used by the other processes. Therefore, it must be designed as a set of exported routines that may be called by other processes. This should be the general view of a process belonging to an ATLAS application.

From the developer point of view, a process consists of two parts: its interface and its implementation.

The Process Interface
The process interface is a module written in ATL language which defines the prototype of the public routines of the process and the needed types for their parameters and return results.

The ATL language (described in detail in [4]) is an imperative and modular language designed for ATLAS applications. It allows to define types, variables, functions and procedures that can be exported (visible to the others) or local. It also accepts the most common control structures inside functions and procedures (conditionals, loops, etc.) and routine calls both synchronous and asynchronous.

Although only its prototypes and types are needed for a process interface, the module defining the process interface can also include functions or procedures defined in ATL which describe the interaction with other processes

in the application or with the user (e.g. asking for input data). ATL modules which are not the interface of any process but define the execution coordination and interaction between processes can also be defined in the application, and users may dynamically add their own.

An example of an ATL module with part of the interface of a process called `volum` can be seen in figure 3. Since it is an example it is not complete, but it shows the definition of a set of types exported by the module (some of them are needed as parameters of extern routines), the prototypes of two extern routines (these prototypes and the types of its parameters would form the interface of the process), and the description of a procedure (being also exported to be visible to other modules) that combines the execution of the process routines, asks for an input datum (through `GETDATA`) and also calls to a procedure of another module (`se::Output`).

```
USE se;
EXPORT #deftype point STRUCT
                        x -> real;
                        y -> real;
                        z -> real;
                      ENDSTRUCT
EXPORT #deftype face VECTOR [3] OF STRUCT
                                    p1 -> point;
                                    p2 -> point;
                                    id -> integer;
                                  ENDSTRUCT
EXPORT #deftype simplex STRUCT
                          name -> string;
                          sides -> VECTOR [4] OF face;
                        ENDSTRUCT
EXPORT #deftype scene VECTOR [100] OF simplex
EXPORT #deftype property integer

EXPORT scene totalsc;
       ...
PROT
  EXTERN FUNCTION segmentation (scene sc, property p)
           RETURNS scene;
  EXTERN PROCEDURE display_scene (scene sc);
     ...
ENDPROT
      ...
EXPORT PROCEDURE SegmentSimplex () IS
  display_scene (segmentation(totalsc,
           GETDATA("Input the property value")));
  se::Output ("Segmentation completed","m");
ENDPROCEDURE
```

Figure 3: Portion of the interface definition in Atlas for the volume modeling process ("volum").

The Process Implementation
Since giving the process interface in the ATL module allows Atlas to generate code stubs to implement the communications driver for this process (see subsection 4.1), from the developer point of view the process implementation consists of the set of C++ routines declared as externals in the ATL module plus the definition of the C++ classes used by their parameters. This implementation can also include whatever the developer wants as a private part of the process. This part won't be visible outside the process.

Figure 4 shows how an Atlas executable process is generated from its source files. The files depicted on the left most column are those that the developer must implement.

The automatically generated code is divided in three files: the `atl_process.hh` file defines the C++ prototypes for the process routines declared as extern routines in the ATL module; the `atl_process.H` file has the *bridge types* implementation for that types used by the extern routines of the process; finally the `atl_process.C` file implements the main code for the communications driver and also auxilary routines which convert Atlas `Variables` to the process C++ classes and backwards in order to be able to call the corresponding process routine with the correct parameters and result variables.

4.3 Other Benefits To The Developer

There are also other benefits provided to the developer that maybe are interesting to be mentioned:

- At run time, the Atlas kernel keeps some structures containing information about the status of the application (what processes are in execution, if there is some request waiting for an input data, etc). To give the opportunity that an application process be informed about changes in this internal structures, Atlas offers the Atlas *events mechanism* that allows the process to ask for a subscription to a particular Atlas event (ADD_PROCESS when a new process starts in the application or ADD_INPUT when an input is produced by the end user, for example). Whenever an Atlas event is produced the `distr` process sends the corresponding event message to every process subscribed to that event, and the driver of the process, when it receives this event message, calls the routine attached to this event at subscription time.
- In order to address the *standardized input model* presented as an Atlas objective in section 2, Atlas also offers 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 consider them input data to be channeled to those processes. The input system is also extensible. In fact it is also an interface between Atlas and an extended Tcl/Tk [11] engine, so scripts in Tcl can be sent to it to instantiate new interface components.

  Using this Atlas component the developer can prepare the user interface for his application almost trivially, and dedicate most of his time to the proper subject of his application.

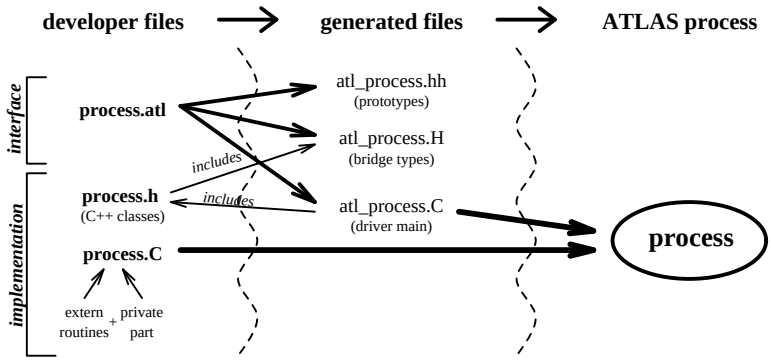

Figure 4: Creating an Atlas process.

- The virtual machine that interprets ATL code allows both synchronous and asynchronous calls to external routines. Some processes may therefore act as large batch processes that are executed concurrently with the application. To rendezvous with these asynchronous calls, Atlas uses a simple device: the virtual machine tags all output parameters or return values of an asynchronous call as "dirty", and any attempt to use one of them as an r-value freezes the executing thread. Thus asynchronous calls may be issued and other portions may properly await their completion in a transparent manner.

5 CONCLUSIONS AND FUTURE WORK

We have presented a software platform designed to allow developers to incorporate advanced features in their development with the least hassle. It is presently being used within our lab to port several packages developed here, and also to build new ones. It's design favors the construction of reusable modules that can relatively easily be combined with each other. This seems very desirable, especially in an environment like ours, where large portions of code are constantly being generated by students which later depart.

The users may just as easily add processes to implement a new application or to extend Atlas itself. Presently, for instance, a menu-handling module is being constructed, that will then be available for all other Atlas applications to define their own, very flexible, menus.

We are currently porting the current version (0.2) of Atlas to different platforms, currently including Suns under both Solaris 1.x and 2.x, and HPUX, but soon to include also SGI's IRIX 6.x and Windows NT. We are also completing the journaling mechanism, which will not only allow replays of sessions, but will support the fault-tolerance within Atlas, and will provide unlimited (albeit expensive) undo's and redo's through a "commit" blocking instruction within the journal, and support for inverse functions.

Among the near future projects, we plan to add some support for CSCW by the simple device of cloning the application for the different users collaborating, and establishing special connections between the corresponding `distr` processes, only one of which acts as master. This, although limited, would turn essentially every Atlas application into an CSCW-capable application, with no or extremely little effort by the developers, as per Atlas's requirements.

REFERENCES

[1] G. R. Andrews. Paradigms for Process Interaction in Distributed Programs. *ACM Computing Surveys*, 23(1), March 1991.

[2] R. S. Chin and S. T. Chanson. Distributed Object-Based Programming Systems. *ACM Computing Surveys*, 23(1), March 1991.

[3] D. D. Cowan and C. J. Lucena. Abstract Data Views: An Interface Specification Concept to Enhance Design for Reuse. *IEEE Transactions on Software Engineering*, 21(3), March 1995.

[4] M. Fairén and A. Vinacua. ATLAS. Sistema de Comandes: Manual tècnic (in Catalan). *Report LSI-95-11-T*, 1995. http://www.lsi.upc.es/~ mfairen.

[5] M. Fairén and A. Vinacua. Interprocess data transfer in Atlas, a platform for distributed applications. 1997. Submited to the OPENARCH'98 conference.

[6] M. Fairén and A. Vinacua. Atlas, a platform for distributed graphics applications. 1997. To appear in the proceedings of Eurographics Workshop on Programming Paradigms in Graphics.

[7] D. C. Schmidt. The ADAPTIVE communication environment: Object-oriented network programming components for developing client/server ap-

plications. In *12th Sun Users Group Conference*, 1994.

[8] D. C. Schmidt. Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching. In *Proceedings of the 1st Pattern Languages of Programs Conference*, August 1994.

[9] A. Soto, S. Vila, and A. Vinacua. A Toolkit for constructing command driven graphics programs. *Computer & Graphics*, 16(4):375–382, 1992.

[10] R. Srinivasan. Rfc 1832: Xdr: External data representation standard, August 1995.

[11] B. B. Welch. *Practical Programming in Tcl and Tk*. Prentice Hall PTR. Upper Saddle River, New Jersey 07458, 1995.

APPENDIX: AN EXAMPLE OF THE AUTOMATICALLY GENERATED CODE

Using the portion of the `volum` process interface shown in figure 3 the ATLAS code generator makes automatically the files `atl_volum.hh`, `atl_volum.H` and `atl_volum.C` which are partially depicted in figures 5 through 7.

```
#ifndef __ATL_volumhh__
#define __ATL_volumhh__

#ifndef NOHEADER
#include "volum.h"
#endif
#include "atl_volum.H"
scene segmentation(scene,property);
void display_scene(scene);

#endif
```

Figure 5: The automatically generated atl_volum.hh file.

Figure 5 shows the code generated to define the C++ prototypes for the two extern routines declared in the interface. This file also includes the `volum.h` file implemented by the developer because the prototypes use process types only known by the developer code.

In figure 6 we can see two of the five *bridge type* definitions corresponding to the exported types defined in the interface: the most simple one is the one corresponding to a type definition which depends on another type defined before, and the other one is the one corresponding to the most complex one whose contents depend on another type defined before and its methods show how this *bridge type* is made from an ATLAS `Variable` and backwards. These two methods make possible the automatic translation between the *bridge type* and the corresponding `Variable`, isolating the developer from this ATLAS external representation.

The last one, figure 7, shows a portion of the main code of the driver. This code includes an auxiliary routine for each one defined as an external routine in the interface (in the figure only the one for `segmentation` routine is shown), and the main routine of the communications driver. The auxiliary routine is the one to translate the parameter types from `Variables` to the process types (through *bridge types*) in order to call the process routine in the correct way, and also to translate back the result of the process routine to have a `Variable` to go through the network. The main routine of the driver only have to make some initializations for it and enter in the dispatching loop.

```
       ...
namespace volum {
typedef atl_pyramid atl_simplex;
}

namespace volum {
struct atl_scene{
   atl_simplex cont[100];
   operator Variable() {   // automatic translation to a Variable
      Type t("volum::scene",
             "V[100]S(name string,base V[3]S(p1 S(x real,y real,z real),
                                             p2 S(x real,y real,z real),
                                             ident integer),
                      sides V[3]V[3]S(p1 S(x real,y real,z real),
                                      p2 S(x real,y real,z real),
                                      ident integer))");
      Variable v(t,""); v.build_tree();
      for (int i1=0;i1<100;i1++)
        { *((*(v.Tree())).accede(i1)) = *(((Variable)cont[i1]).Tree()); }
      return (v);
      }
   atl_scene() {}
   atl_scene(Variable &v) {     // constructor from a Variable
      if (v.Tree()==NULL) atl_exit(-1); // Invalid variable
      for (int i1=0;i1<100;i1++) {
         Variable v2("S(name string,base V[3]S(p1 S(x real,y real,z real),
                                               p2 S(x real,y real,z real), ident integer),
                        sides V[3]V[3]S(p1 S(x real,y real,z real),
                                        p2 S(x real,y real,z real), ident integer))","");
         v2.build_tree();
         *(v2.Tree())=*((*(v.Tree())).accede(i1));
         atl_simplex tpaux(v2); cont[i1]=tpaux;
         }
      }
   };
}
       ...
```

Figure 6: Portion of the automatically generated atl_volum.H file.

```
   Comunic_Distr distrib(CANAL_COMUNIC_DISTR);
   String nameprogram;
   Driver driv(distrib);
       ...
   void aux_segmentation(String codi,
                         DLList<Variable *> &parameters) {
      Pix p=parameters.first();
      atl_scene ptp0(*(parameters(p)));
      scene par0(ptp0);
      parameters.next(p);
      property ptp1(((nodeint *)parameters(p)->Tree())->Getvalue());
      parameters.next(p);
-->   scene res=segmentation(par0,ptp1);
      atl_scene restp;
      restp=res.conversion_to_bridge_type();
      Variable *vr=new Variable(restp);
      ReturnValue *rv=new ReturnValue(codi,vr);
      distrib.send(rv);
      }
       ...
   void main(int argc,char **argv) {
      nameprogram=argv[0];
      ini_to_calls();          // some inicializations for the driver
      driv.set_name_program(nameprogram);
      ini_process();           // inicializations of the process itself
      driv.Dispatch();         // loop
      close(CANAL_COMUNIC_DISTR);
      exit(0);
      }
```

Figure 7: Portion of the automatically generated atl_volum.C file. The arrow has been added pointing to the point where user code is actually invoked.