UNIVERSITAT POLITÈCNICA
DE CATALUNYA

On the design and implementation of flexible software platforms to facilitate the development of advanced graphics applications

Marta Fairén

June, 2000

PhD Thesis
Supervised by Dr. Àlvar Vinacua

Software PhD program
LiSI Department,
UPC

Chapter 9

Using the system

9.1 Design process of an ATLAS application

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

9.1.1 The process interface

The process interface is a module written in ATL language (see chapter 4) which defines the prototype of the public routines of the process and the types needed for their parameters and return results.

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).

An example of an ATL module with part of the interface of a process called `volum` can be seen in figure 9.1. 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 external routines), the prototypes of two external routines (these prototypes and the types of its parameters would form the interface of the process), and the description of a procedure (also exported to make it visible to other modules) that combines the execution of the process routines, asks for an input datum (through `GETDATA`) and also calls a procedure of another module (`se::Sortida`).

9.1.2 The process implementation

Giving the process interface in the ATL module allows ATLAS to generate code stubs to implement the communications driver for this process (see chapter 7). 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

```
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::Sortida ("Segmentation completed","m");
ENDPROCEDURE
```

Figure 9.1: Portion of the interface definition in ATLAS for the volume modeling process ("volum").

of the C++ classes used by their parameters[1]. This implementation can also include whatever the developer wants as a private part of the process. This part will not be visible outside the process.

Figure 9.2 shows how an ATLAS executable process is generated from its source files. The files depicted on the leftmost column are those that the developer must implement.

The automatically generated code consists of three files and a dynamic library created from other temporary generated files. All of these files of generated code have been explained in section 7.5.1.

The developer must implement the header file `process.h` where the C++ classes corresponding to the ATL exported types and the prototypes for the exported routines must be declared, and the file `process.C` with the implementation of the exported routines (declared as external in the ATL module); in fact this last file may be given any other name, and can be also split in more than one C++ file, but must include `process.h`.

We want to remind also that inside the C++ classes corresponding to the ATL exported types the translation methods from and to the *bridge types* must also be implemented (this has been explained in chapter 7). These are:

[1]This dependency of C++ is only because of our current code generator; but this might be another language as stated in chapter 1

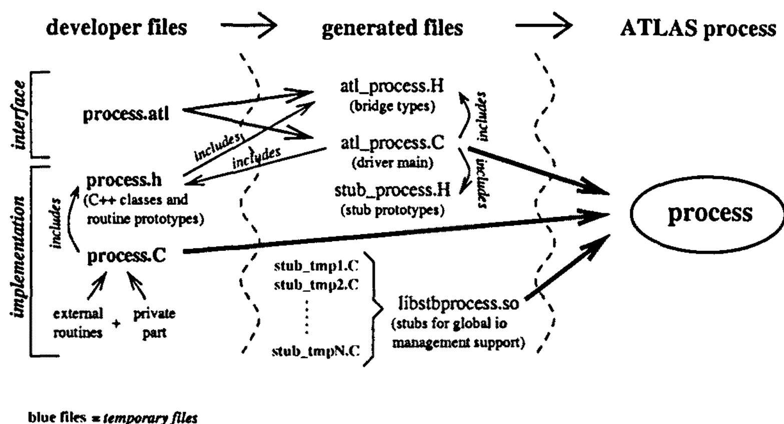


Figure 9.2: Creating an ATLAS process.

- a constructor from the corresponding bridge type. The name of the bridge type is the name of the type prefixed by *atl_* (e.g. for the *Point2D* type the bridge type would be *atl_Point2D*);
- a translating method to the bridge type called `conversio_a_tipus_pont()` without any parameter and returning a bridge type object.

The name given to the ATL exported type must be the same as the C++ class for this type because part of the code using this class is automatically generated. The header file `process.h` must also include the generated header file which declares the bridge types, `atl_process.H`.

The code of an example of a toy application implemented over ATLAS can be seen in section 9.4.

9.2 The API library

The API library includes the routines offered by ATLAS to be used by the developer in the application processes. In order to use the API library the process must include the file `inc_atlas.H` which declares the prototypes of the API routines.

The routines offered by the ATLAS API library are:

- "void **atl_subscribe** (char*ev, typfunc f, bool upd, Comunic_Distr &d);"

  This routine is used to *subscribe the process to the* ATLAS *event* given as the first parameter. The ATLAS events mechanism is fully explained in section 6.4. The ATLAS events a process can subscribe to are:

event identifier		produced when...
ADD_DEMAND	⟶	a data request is added to the list
SUB_DEMAND	⟶	a data request is removed from the list
MOVE_DEMAND	⟶	a re-ordering is produced in the requests list
ADD_PROCESS	⟶	a new process is added to the application execution
SUB_PROCESS	⟶	a process is removed from the application execution
IMALIVE	⟶	a *heartbeat* message has arrived to `distr`
ADD_INPUT	⟶	an input datum is added to the list
SUB_INPUT	⟶	an input datum is removed from the list
MOVE_INPUT	⟶	a re-ordering is produced in the input data list

The first parameter, ev, is the event identifier to which the process wants to be subscribed.

The second parameter, `f`, represents the function to be called when the event message arrives to the process. The type of this parameter, *"typfunc"*, is defined as "`typedef void (*typfunc)(Param *);`" and this `Param` is an abstract class which has three derived classes:

- *"ParamDemand"* contains an object *"Demanda"*. This type is defined as

```
class Demanda
{
    String ident;           // data request identifier
  · String nom_proces;      // name of the process requesting it
    String tipus;           // name of the data type
    String nom_dades;       // string shown to the final user
    fourbytes timeout;
    unfourbytes thread;     // thread of the execution who produced it
    void omplir (String id, String np, String t, String nd,
                 fourbytes tm);
    int llargada ();
  public:
    Demanda ();
    Demanda (RequestData missdemand, String proces);
    String Ident () { return ident; }
    String Tipus () { return tipus; }
    String Nom_proc () { return nom_proces; }
    String Nom_dades () { return nom_dades; }
    fourbytes Timeout () { return timeout; }
    unfourbytes Thread () { return thread; }
    ~Demanda ();
};
```

  and is the information received when an ADD_DEMAND event is produced. The *"Demanda"* can be accessed by using the method "`cont ()`" of *"ParamDemand"* which returns a *"Demanda &"*.

- *"ParamInput"* contains an *"Input"* object. This type just consists of an identifier given to the input datum and the type name of this datum. The *"Input"* class is defined as:

```
class Input
{
    int identr;            // input datum identifier
    String tipus;          // type name of the input datum
  public:
    Input (int id, String tip);
    int Identr () { return identr; }
    String Tipus () { return tipus; }
};
```

This *"Input"* object is received when an ADD_INPUT, SUB_INPUT or MOVE_INPUT event is produced. The *"Input"* can be accessed by using the method "`cont ()`" of *"ParamInput"* which returns an *"Input &"*.

- *"ParamString"* contains a *String* object which is the information associated to all events except those receiving a *"Demand"* or an *"Input"* (ADD_DEMAND, ADD_INPUT, SUB_INPUT, MOVE_INPUT). The *String* can be accessed by using the method "`cont ()`" of *"ParamString"* which returns a *String &*.

Since the *"typfunc"* wants a "Param *" as a parameter, and the actual functions defined by the developer will usually receive a pointer to one of the derived classes (ParamDemand, ParamInput or ParamString) as a parameter, when the subscription API routine is used, the second parameter requires an explicit cast to `typfunc`. An example of using this routine can be:

```
// prototype
Adding_process (ParamString *parst);
   ...
// Call to subscription
atl_subscribe (ADD_PROCESS, (typfunc)&Adding_process, true);
```

The third parameter, upd, is optional and indicates whether an update of the list associated to the event is required or not. Passing *true* to this parameter only makes sense for events ADD_DEMAND, ADD_PROCESS and ADD_INPUT, and its effect is that the system tries to give the current state of the list to the process by sending to it an event of this type for each element currently in the list in the appropriate order. This allows processes being started in the middle of an execution to subscribe to a list and get its current state. The default value for this parameter is *false*. If given as *true* for an event other than the three just mentioned, the system will give a warning message.

The last parameter, d, is also optional and represents the object wrapping the communication of the process with `distr`. The default value for this parameter is the communication object each process has by default (see the code generation in section 7.5.1).

- "void **atl_endsubscribe** (char ev, Comunic_Distr &d);"

  This routine causes the *removal of the subscription* of the calling process to the given event.

  The first parameter, ev, is the ATLAS event identifier, and the second, d, is optional and represents the communication with `distr` (as in the routine `atl_subscribe`).

- "void **atl_send_input** (Variable &var, fourbytes tm, Comunic_Distr &d);"

  This routine *sends an input datum* to `distr`.

  The first parameter, `var`, is the datum to be sent. The *"Variable"* type is used by the system to pass data through the network (see also chapter 7), but the developer does not have to know this type in order to use the routine. The automatic code generation implements the *bridge type* and a casting operator from it to *"Variable"*, so as this first parameter the developer may pass a *bridge type* object and the cast will be done automatically.

  The second parameter, `tm`, is optional and represents the timeout value for the input datum (see section 6.3).

  Here we only remind the possible timeout values, which in case of input data can be:

  - **timeout > 0** ⟶ the datum waits to be used until *timeout* seconds have passed, if this time expires, it is removed from the list without being used.
  - **timeout = 0** ⟶ the datum is immediate, i.e. if there is no request waiting for this datum, it is not appended to the list but directly removed.
  - **timeout = -1** ⟶ the datum does not have *timeout*, it waits in the list until it is used or the application execution finishes. This is the default case.

  The `fourbytes` type is an integer number always represented by 4 bytes independently of the architecture where it is used (in SUN workstations, for example, it is defined as an 'int').

  The third parameter, `d`, is also optional and represents the communication with `distr` (as in the routine `atl_subscribe`).

- "int **atl_send_request** (String typ, String cursor, fourbytes tm, Comunic_Distr &d);"

  This routine *sends a data input request* to `distr`.

  The first two parameters are those containing the data needed to build a request: the first, `typ`, is the name of the type requested, and the second, `cursor`, is the prompt message to be shown to the user for the request.

  The third parameter, `tm`, is optional and represents the timeout value to be given to the request. The possible timeout values for requests are (see also section 6.3):

  - **timeout > 0** ⟶ the request waits for the input datum until *timeout* seconds have passed. When this time expires, the request is answered with a null response (see section 6.3).
  - **timeout = 0** ⟶ the request has to be served immediately, so if an input datum was waiting in the list for this request it is served, but if not the request is immediately answered with a null response.

- **timeout = -1** ⟶ the request does not *timeout*, it waits in the list until it is served or the execution ends. This is the default value.
- **timeout = -2** ⟶ a request with this *timeout* value is identified by `distr` as a reentrant request. When this request is served with an input datum `distr` automatically puts the request back in the list, effectively producing another identical request.

The fourth parameter, `d`, is also optional and represents the communication with `distr` (as in routine `atl_subscribe`).

The integer returned by this routine identifies the request uniquely in the process. This integer is incremented automatically by the library.

- "void **atl_send_modifreq** (String ident, Comunic_Distr &d);"

  This routine asks the `distr` process for a *re-ordering of the requests list.* It causes the request with the given identifier (`ident`) to become the first on the list to be served when an input data of the same type arrives (see also section 6.3).

  The identifier of the request (the first parameter) consists of the name of the process which generated the request followed by a colon and the order number of this request in that process (formatted as a decimal integer with no leading zeros or spaces).

  The second parameter, `d`, is optional and represents the communication with `distr` (as in the routine `atl_subscribe`).

- "void **atl_send_modifinput** (int ident, Comunic_Distr &d);"

  This routine sends a *re-ordering request for the input data list* to the `distr` process. It causes the input datum with the given identifier to become the first on the list of pending inputs (see also section 6.3).

  The first parameter, `ident`, is the identifier of the input data to be moved to the first place.

  The second parameter, `d`, is optional and represents the communication with `distr` (as in the routine `atl_subscribe`).

- "void **atl_ask_input** (int ident, Comunic_Distr &d);"

  This routine asks for the value of the input datum identified by `ident`. Since the ADD_INPUT event message does not contain the datum, the process subscribed may want to access its value, and can ask for it using this routine.

  The second parameter, `d`, is optional and represents the communication with `distr` (as in the routine `atl_subscribe`).

- "void **atl_send_command** (String comm, Comunic_Distr &d);"

  This routine sends a *command message* to `distr`. This message will be sent by `distr` directly to the `Command Subsystem` in order to be interpreted by it.

  The first parameter, `comm`, is the command string to be sent. It can be anything the ATL compiler is able to accept at this point of execution (see

also chapter 4). Other values will –of course– trigger an error message by the compiler.

The second parameter, `d`, is optional and represents the communication with `distr` (as in the routine `atl_subscribe`).

- "void **atl_send_error** (String message, char level, Comunic_Distr &d);"

  This routine sends a message to be treated by the *errors management of the system* (see next paragraph). The process sends this message to `distr` who treats the message as an error message of the system.

  The ATLAS errors management is offered because in an ATLAS application the error channel of a process may not be visible to the final user. It uses a command offered by the `Input Subsystem` "`Sortida (String miss, String level)`", handing the message to be shown to the user and also a parameter `level` which selects one of three different levels of importance: **error** → `level` = "e", **warning** → `level` = "w" or **information message** → `level` = "m". More information about this command can be found in chapter 5.

  The two first parameters of the `atl_send_error` routine are those to be sent as parameters for the `Sortida` command of the `Input Subsystem` just mentioned.

  The third parameter, `d`, is also optional and represents the communication with `distr` (as in routine `atl_subscribe`).

- "void **atl_exit** (int ret, Comunic_Distr &d);"

  This routine is used to *end the process execution*, but advising `distr` first and passing to it the exit value of the process.

  The first parameter, `ret`, is the exit value of the process. This value will be sent to `distr` and next the process will exit with this value.

  The second parameter, `d`, is also optional and represents the communication with `distr` (as in routine `atl_subscribe`).

- "void **atl_high_freq** (int factor);"

  This routine *increments the interruption frequency* of the SIGALRM signal by the factor given as a parameter. This will be useful to redefine the method of the class *"Comunic_Distr"* managing this interruption and allows other time-dependent processing to be supported. Although this interruption treatment can be redefined, the *heartbeat* message must still be sent at least with the same frequency as before. We can see an example of its use in the utility process "`processos`" in section 9.3.1.

- "int **atl_compute_factor** (long microsecs);"

  This routine *computes the factor* needed to be the parameter for the `atl_high_freq` routine from the number of micro-seconds desired for the interruption interval.

  The parameter `microsecs` is the number of micro-seconds, and the routine returns the increment factor needed as parameter to `atl_high_freq`, which depends on the frequency the system uses.

- "int **atl_substitute_ticket** (atl_tkt idnow, atl_tkt idafter, Comunic_Distr &d);"

  This routine *asks for a substitution of tickets in the journal.* It is used to identify new input data as data already known by the process. See more details of its usefulness in section 8.1.3 – *Global data identification.*

  The two parameters are respectively the identifier to be substituted and the substitutor.

  The third parameter, `d`, is optional and represents the communication with `distr` (as in routine `atl_subscribe`).

- "atl_tkt **atl_get_ticket** (int howmany);"

  This routine *asks for one or more new atl_tickets* to the system. It is used by a process to identify its own data with the global data identifiers offered by the system.

  The parameter `howmany` is optional (its value is 1 by default) and it indicates the number of new atl_tickets the process wants to get from the system.

  The returning atl_ticket corresponds to the first of the consecutive `howmany` atl_tickets assigned to the process for this demand.

- "String **atl_get_local_path** ();"

  This routine *asks for the absolute path* where the process has been executed. Since the process is executed by the server daemon it may have to know its own directory in order to use some files with paths relative to the installation directory.

  The returning String is the absolute path for this process.

- "void **atl_dontrecover** ();"

  This routine indicates to `distr` that *the process does not need to be recovered* in case it or its communication with `distr` crashes (see also section 8.1.3 – *Process recovery*). It is used usually in the initialization of the process.

- "void **atl_checkpoint** (String filename, atl_ckpcall routinename);"

  This routine indicates that *the process has done a checkpoint at this time* which must be recorded in the journal (see section 8.1.3 – *Checkpoints*).

  The first parameter, `filename`, is the name of the file where the checkpoint has been saved.

  The second parameter, `routinename`, is the name of the routine to be called in case the system asks the process to recover that checkpoint. The type used for this parameter, `atl_ckpcall`, is the type of the routine which returns void and has a String as a parameter which will be the file name where the checkpoint is saved.

There are other routines added to the library doing nothing by default, but available as hooks that the developer can redefine them in the process implementation to add some functionality. These routines are:

- "void **ini_process** ();"

  This routine is called by the communications driver to initialize the process (see chapter 7). By default it does nothing but the developer can redefine it to initialize the process before the driver starts to manage messages coming from `distr`. (A typical use is adding channels that the driver should listen to).

  The utility processes explained in section 9.3 can be examples of the use of this routine.

- "void **treat_before_exit** ();"

  This routine is called also automatically when the process driver receives from the `distr` an order to finish its execution. It can be useful if the process has to do some housekeeping of external data before exiting its execution.

- "void **treat_data** (AnswerData miss);"

  This routine is called when the driver receives an input data requested by the process by using the API routine `atl_send_request`. The `AnswerData` type is a sort of message used in ATLAS to wrap the input data answering a request. Its interface is:

```
class AnswerData : public Message
{
    fourbytes demnd;          // data request identifier
    Variable *varbl;

    void construir (String miss);
    String Buffer ();
  public:
    AnswerData (char c, fourbytes l, String miss);
    AnswerData (fourbytes ident, fourbytes dem, Variable *v);
    AnswerData (const AnswerData &ad);
    int envia_xdr (SOCK_Stream canal_com);
    int envia_xdr (int canal);
    fourbytes Demnd () { return demnd; }
    Variable &Varbl () { return (*varbl); }
};
```

  The only possibility in the current version of ATLAS for a process to manage an input requested by the process is in an asynchronous way. The process does not have the possibility to stop its execution waiting to this input datum to come. Some initial ideas about how to extend ATLAS to allow this synchronous management have been designed and presented as an extension in chapter 10.

- "void **treat_answer_data** (AnswerInput miss);"

  This routine is called when the driver receives a message answering to the process question `atl_ask_input`. The `AnswerInput` type is a sort of message similar to the `AnswerData` which does not have identifier of a request since it is not produced by the matching of input data with a request (see section 6.3).

```
class AnswerInput : public Message
{
    fourbytes ident;          // input data identifier
    Variable *varbl;

    void construir (String miss);
    String Buffer ();
  public:
    AnswerInput (char c, fourbytes l, String miss);
    AnswerInput (fourbytes id, Variable *v);
    int envia_xdr (ACE_SOCK_Stream canal_com, FILE *fstream);
    fourbytes Ident () { return ident; }
    Variable &Varbl () { return (*varbl); }
};
```

9.2.1 Miscelaneous

It is also possible to *add and remove a communications channel to and from the process driver.* These possibilities have been already mentioned in chapter 7 but they have not been explained yet in detail.

To add a channel to the process driver the method `Add_handler` of the class *"Driver"* must be used. The channel must be wrapped in a class derived from *"Event_Handler"* and must redefine the methods `get_fd`, returning the file descriptor of the channel, and `handle_input`, which is called when data are available on the channel.

The prototypes of these methods are:

```
int get_fd () const;
int handle_input (int fd);
void Add_handler (Event_Handler &e,
                  Reactor_Mask mask=Event_Handler::RWE_MASK);
```

An example of this can be seen in the utility process "**demandes**", explained in section 9.3.2.

To remove a channel from the process driver the method `Remove_handler` of the class *"Driver"* must be used. Its prototype is:

```
void Remove_handler (Event_Handler &e,
                     Reactor_Mask mask=Event_Handler::RWE_MASK);
```

The developer may also *change the routine to treat an* ATLAS *event message* at any time, even if the process is already subscribed to the event. This can be done by using the method `ini_event_function` of the class *"Comunic_Distr"*. This method receives as parameters the event identifier and the address of the routine to be assigned. The prototype is:

```
void ini_event_function (int ev, typfunc f);
```

There are also other routines in the API directly related to the journal edition and translation offered by `distr` to be used by a Meta-journal editor. These are explained in section 8.2.

9.3 Utility processes

9.3.1 Process "processos"

This utility process is a clear example of the *heartbeat* mechanism (see chapter 6). It shows the list of the active processes at any time, and a big bullet next to each process name. This bullet changes its color (from green through white to red) while time is passing and changes to the initial color when the process sends a *heartbeat* message to `distr`. This utility is useful to know which processes are working well in the application (a snapshot can be seen in figure 9.3).

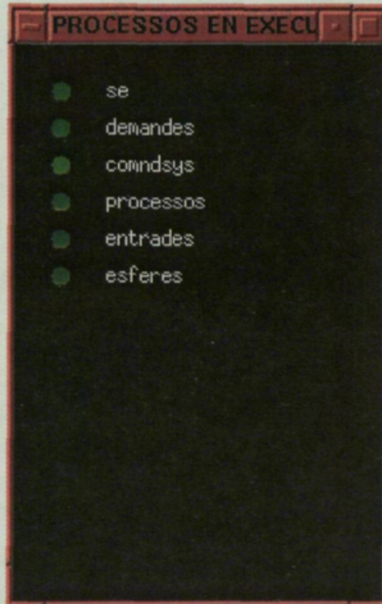


Figure 9.3: Snapshot of the "processos" utility process

The code of this utility process can be used as an example to see how the frequency of the SIGALRM interruption can be increased by deriving a class from *"Comunic_Distr"*. Note how the original routine is called at the appropriate times (`Comunic_Distr::handle_signal()`) to keep sending the *heartbeat* messages at the correct rate.

To the file `processos.h` we add the class deriving of *"Comunic_Distr"*:

```
class Prove : public Comunic_Distr
{
    int compt;
  public:
    Prove (int fd): Comunic_Distr(fd) { compt = 0; }
    int handle_signal (int signum) {
      NewStatus ();
      if (compt++ == atl_compute_factor (200000))
        { compt = 0;
          return (Comunic_Distr::handle_signal(signum)); }
      else return 0;
    }
};
```

where NewStatus () is the treatment to be done each time the SIGALRM interrupts.

The initialization of the process then has to increase the frequency of the signal interruption:

```
void initiate ()  // routine called by the "main" procedure in the ATL module
{
  // Asking for the subscription to the required ATLAS events
  atl_subscribe (ADD_PROCESS, (typfunc)&afegir_proces, true);
  atl_subscribe (SUB_PROCESS, (typfunc)&eliminar_proces);
  atl_subscribe (IMALIVE, (typfunc)&proces_viu);

  // Increasing the frequency of the SIGALRM interruption indicating
  // it must interrupt each 0.2 seconds (200000 microseconds)
  atl_high_freq (atl_compute_factor (200000));
}
```

Since we changed the class of the communication with distr, the automatically generated code must recognize this change. This can be done by defining the preprocessor symbol COMUNIC_DISTR with the name of the new class at compilation time (i.e. we need the flag "-DCOMUNIC_DISTR=Prove" in the compilation command). If this preprocessor symbol is not defined in the compilation command, the generated code will use the "Comunic_Distr" class for this communication as usual (see the generated code in section 7.5.1).

9.3.2 Process "demandes"

The "demandes" process is a tool to complement the ATLAS functionalities. This process shows in a window the list of requests waiting in distr to be served by input data. It also allows to re-order these requests, choose the method to use to input the data of a given type and to associate input methods to the type of this datum.

To explain it with an example, if we have a type Point, which is a 3D point used to select vertices of a tridimensional model, it could be interesting to have different possibilities to input this Point. For example:

- The point can be input by pushing the mouse button in a pixel of the window in the screen where the data we want to select are drawn. This pixel will be translated to a 3D point to be identified with the vertex we want to select.
- The point can be input as a set of three real numbers describing respectively the coordinates x, y and z of the desired vertex.

Having the two possibilities, the user can choose the way to introduce the point when the application needs this point as an input data from the user. It is then a very useful utility specially for interactive graphics applications.

The user point of view

The process shows (as we already said) the list of requests that ATLAS has in its internal structure at any time. To achieve this, the process is subscribed to the events of add, remove and re-order requests, so the process will be notified when this ATLAS events are produced.

The possibilities for the user are:

- If the requested type does not have any special input methods, the programmer must have designed-in some process that will eventually produce this input. This is the most common case for data of basic types (produced by `se`).
- If the request type has different ways to input data associated, the request appears with an arrow at the end which indicates that there is a sub-menu in order to be able to select one of the ways to input the datum.
- It is also possible to have a default input option which will be activated automatically when the request is produced. This can be changed later for one input by asking for a substitution of the command automatically started by default (see chapter 10).

The use is always based on pushing the different buttons of the mouse. With the right one we just re-order the list, putting the one clicked upon at the top of the list. With the left one we can open the sub-menu if it exists and choose the preferred method to input this datum (see figure 9.4).

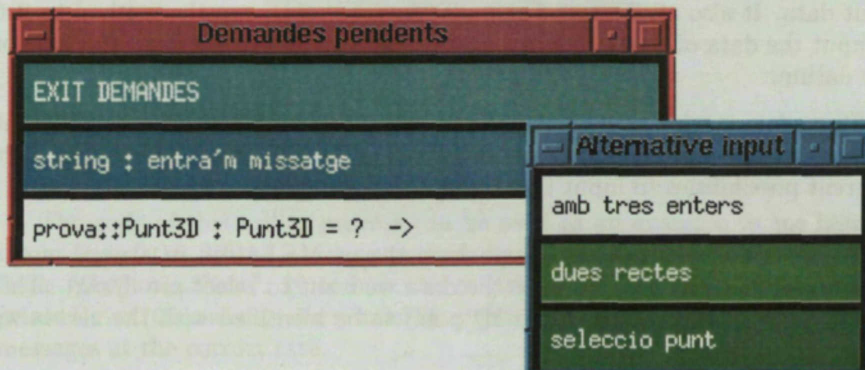


Figure 9.4: Windows with the information of the requests list and the different ways to input a Punt3D.

The developer point of view

In order to be able to add alternative input methods to be associated to a data type, the process offers three exported commands defined in the ATL module:

- PROCEDURE Add_Input_Method (string type, string proc, string text, boolean default)

  This command adds a new way to input data of type type. proc is the command associated to this method (the string is exactly the command to be executed), the text to appear in the sub-menu item, text and a flag indicating if this method must become the default one or not.

- PROCEDURE Delete_Input_Method (string type, string proc)

  This command removes the existing command proc associated to the type type as a possible way to input this type of data.

- PROCEDURE ChgDflt_Input_Method (string type, string proc, string text)

  This command changes the default input method for type type. If proc did not exist it is added.

As an example of an added channel

This process can also be an example of how the developer can add a new channel to the process communications driver. In this case the channel is the XConnectionNumber that connects the process to the window manager (in order to listen to events of the X window).

It defines a derived class of *"Event_Handler"* in order to add this handler to the driver:

```
#include "ComunicXevent.H"
#include "Driver.H"

extern Driver driv;
Comunic_Xevent eventfin;
void ini_process ()
{
  // Inicialitza el menu.
  ini_menu ();
  // Demanda de subscripcio als events que requereix
  atl_subscribe (ADD_DEMAND, (typfunc)&afegir_opcio, true);
  atl_subscribe (SUB_DEMAND, (typfunc)&eliminar_opcio);
  atl_subscribe (MOVE_DEMAND, (typfunc)&modifica_opcions);
  int X_cn = XConnectionNumber (menu->ObtDisplay());
  eventfin.set_fd(X_cn);
  driv.Add_handler(eventfin);
}
```

where the prototypes associated to the functions are:

```
void afegir_opcio (ParamDemand *dem);
void eliminar_opcio (ParamString *opc);
void modifica_opcio (ParamString *opc);
```

and the handler of Comunic_Xevent is:

```
class Comunic_Xevent : public Event_Handler
{
    int filedesc;
  public:
    Comunic_Xevent () {}
    Comunic_Xevent (int fd) : filedesc(fd) {}
    void set_fd (int fd) { filedesc = fd; }
    int get_fd () const { return filedesc; }
    int handle_input (int fd) { TractarEvent (); return 0; }
};
```

9.3.3 Process "entrades"

The "`entrades`" process is a tool for the final user. It shows the input data list kept in `distr` with input data which are still not used. It also allows to re-order the list as in the "`demandes`" process (subsection 9.3.2), to eliminate an input datum from the list and also to ask for the contents of the input datum independently of its type (except for unknown types –see section 4.1.3).

The removing possibility gives the user the possibility of removing a datum introduced by mistake.

Using it

The process shows (as we already said) the list of input data that ATLAS has in its internal structure at any time. To achieve this the process is subscribed to the events of add, remove and re-order input data, so the process will be notified when this ATLAS events are produced. The process stores the identifier of the data and writes in the window its type name. Figure 9.5 shows a snapshot of the process.

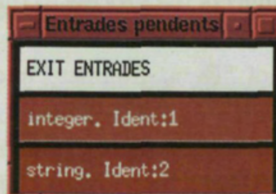


Figure 9.5: Snapshot of the process "entrades".

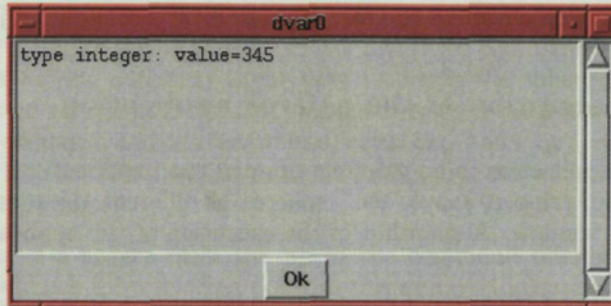
The use is always based on pushing the different buttons of the mouse. With the right one we just re-order the list, putting the one clicked upon at the top of the list. With the middle button the data is removed from the list. With the left button we ask to see the contents of this datum, it will open a window with hypertext and the user can go through the different fields or elements in structures or vectors.

Interpreting the contents

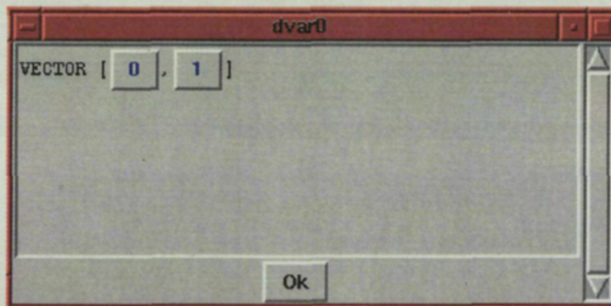
When the contents of an input datum are shown and it is of a basic type (integer, real, boolean, string or any re-definition of them) the information shown will be the name of the type and the value of the datum (see figure 9.6(a)).

When the datum is a vector (seen in figure 9.6(b)) it shows the positions of the vector as buttons that can be clicked to open another window with the contents of the corresponding element.

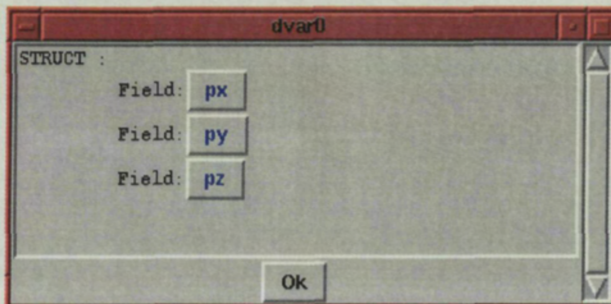
And finally when the datum is a structure (seen in figure 9.6(c)) it shows the name of the fields in the structure as buttons that can be clicked like the vector elements.



(a) Window with the contents of an integer input datum.



(b) Window with the contents of a vector input datum.



(c) Window with the contents of a structure input datum.

Figure 9.6: Windows showing different input data contents.

9.4 A toy example

In this section we describe a complete example application which shows some aspects of the use of ATLAS. It is not meant to be useful or relevant, but complete and simple enough to grasp it as a whole, hence the adjective "toy".

9.4.1 Description of the *esferes* application

The *esferes* application is an example of an ATLAS application. It allows the user to create, remove, move, etc. spheres of different sizes and colors in a graphical 3D window. A snapshot of the execution of the application is shown in figure 9.7.

The example consists of only one user process and uses the functionalities offered by other utility processes in ATLAS, like the demandes process.

Figure 9.7: Snapshot of the *esferes* toy application.

9.4.2 Tiny user manual

The application, when started, creates a graphical 3D window and adds a default sphere in it. The default window's size is fixed (600x600 pixels) and it shows the part of the 3D world contained in the parallelepiped with vertices PMin=(-20,-20,-30) and PMax=(20,20,29). The user is located in the position

(0,0,30) and he looks at the coordinate origin (0,0,0) which will be at the center of the graphical window. The default sphere drawn in the starting command is centered in the origin with radius 1 and color RGB=(0.5,0.5,1).

The description (in a nutshell) of the user commands is:

- To add a sphere to the world the user has three commands: `Add()` which will ask for the necessary input data to create the sphere (center point, radius and color); `AddDefault()` which directly creates the default sphere described above; and `AddCenterRad(real cx, real cy, real cz, real rad)` which creates a sphere centered in `(cx,cy,cz)` with radius `rad` and the default color.
- To remove a sphere from the world the command offered is `Remove()` which will ask for a selected point in the window indicating which sphere to remove.
- To change the color of an already created sphere, there are two commands: `ChangeColor()` which will ask both the sphere to which to change the color and the new color for it; and `ChangeToColor(real r, real g, real b)` which will ask only for the sphere and will change its color to the `(r,g,b)` color.
- To move a sphere the command `Move()` will ask for the sphere to move and also for the new position of its center.
- Finally there is the command `Distance()` which given two spheres (selected as before) calculates the distance between them, giving both the distance between centers and the distance between surfaces. The second one can be negative, meaning the two spheres intersect.

When the application asks for a point or a color as input data it offers two ways to input them: by entering the corresponding textual coordinates or by selecting on the window. The selection on the window for a point gives as a result the corresponding 3D point with the Z component set to zero, and the selection for a color gives as a result the color of the sphere selected.

9.4.3 Technical documentation and code

The interface

The interface of the application is given by the ATL module. In this module the external routines and the exported types needed by these routines are declared.

In this example the exported types are: Sphere, ColorRGB and Point. Lines 3–30 of the `esferes.atl` code listed below are the interface declaration of the application.

Other aspects to be emphasized of this code are:

- The control is given to the `se` process (the ATLAS input subsystem –see chapter 5) to manage the button press events over the X-window created by the process (lines 141–145).

- The use of the Add_Input_Method procedure of the demandes module in order to offer more than one way to input data of types Point or ColorRGB (lines 146–153). The commands to be called when these different input methods are selected are defined in lines 110–137.

- The use of mathematical functions in the ATL module allowing to make some easy computations in the module itself (the Distance() command for example, lines 101–105).

- And also the possibility of providing input data from the Command Subsystem itself (lines 114, 122, 128 and 136).

esferes.alt

```
1   USE se, demandes;
2
3   // Type definitions
4   #deftype ::esferes_Button_pressed_event se::Button_pressed_event;
5
6   EXPORT #deftype Point STRUCT
7                             px -> real;
8                             py -> real;
9                             pz -> real;
10                          ENDSTRUCT;
11  EXPORT #deftype ColorRGB STRUCT
12                               r -> real;
13                               g -> real;
14                               b -> real;
15                             ENDSTRUCT;
16  EXPORT #deftype Sphere STRUCT
17                             center -> Point;
18                             rad -> real;
19                             color -> ColorRGB;
20                           ENDSTRUCT;
21
22  PROT
23    EXTERN FUNCTION GetWindowId () RETURNS integer;
24    EXTERN PROCEDURE AddSphere (Sphere sph);
25    EXTERN FUNCTION GetColor (::esferes_Button_pressed_event ev) RETURNS ColorRGB;
26    EXTERN FUNCTION GetPoint (::esferes_Button_pressed_event ev) RETURNS Point;
27    EXTERN PROCEDURE RemoveSphere (::esferes_Button_pressed_event ev);
28    EXTERN FUNCTION GetSphere (::esferes_Button_pressed_event ev) RETURNS Sphere;
29    EXTERN PROCEDURE ChangeColorSphere (::esferes_Button_pressed_event ev, ColorRGB c);
30  ENDPROT
31
32  // Local function
33  FUNCTION GetDataSphere () RETURNS Sphere IS
34    Sphere sph;
35    sph.center = GETDATA ("Center point values");
36    sph.rad = GETDATA ("Radius value");
37    sph.color = GETDATA ("Color values");
38    RETURN sph;
39  ENDFUNCTION
40
41  // Exported commands
42  EXPORT PROCEDURE Add () IS
43    Sphere sph;
44    sph=GetDataSphere ();
45    AddSphere (sph);
46  ENDPROCEDURE
47
48  EXPORT PROCEDURE AddDefault () IS
49    Sphere sph;
50    sph.center.px = 0; sph.center.py = 0; sph.center.pz = 0;
51    sph.rad = 1;
52    sph.color.r = .5; sph.color.g = .5; sph.color.b = 1;
53    AddSphere (sph);
54  ENDPROCEDURE
55
```

```
56  EXPORT PROCEDURE AddCenterRad (real cx, real cy, real cz, real r) IS
57    Sphere sph;
58    sph.center.px = cx;
59    sph.center.py = cy;
60    sph.center.pz = cz;
61    sph.rad=r;
62    sph.color.r = .5; sph.color.g = .5; sph.color.b = 1;
63    AddSphere (sph);
64  ENDPROCEDURE
65
66  EXPORT PROCEDURE Remove () IS
67    esferes_Button_pressed_event bp;
68    bp = GETDATA("Select the sphere");
69    RemoveSphere (bp);
70  ENDPROCEDURE
71
72  EXPORT PROCEDURE ChangeColor () IS
73    ChangeColorSphere (GETDATA ("Select the sphere to change color"),
74                       GETDATA ("New color values"));
75  ENDPROCEDURE
76
77  EXPORT PROCEDURE ChangeToColor (real r, real g, real b) IS
78    ColorRGB col;
79    col.r = r; col.g = g; col.b = b;
80    ChangeColorSphere (GETDATA ("Select the sphere to change color"), col);
81  ENDPROCEDURE
82
83  EXPORT PROCEDURE Move () IS
84    Sphere sph;
85    esferes_Button_pressed_event bp;
86    Point newcenter;
87    bp = GETDATA ("Select the sphere to move");
88    sph = GetSphere (bp);
89    newcenter = GETDATA ("New center for the sphere");
90    sph.center.px = newcenter.px;
91    sph.center.py = newcenter.py;
92    RemoveSphere (bp);
93    AddSphere (sph);
94  ENDPROCEDURE
95
96  EXPORT PROCEDURE Distance () IS
97    Sphere sph1, sph2;
98    esferes_Button_pressed_event bp1, bp2;
99    sph1 = GetSphere (GETDATA ("Select the first sphere"));
100   sph2 = GetSphere (GETDATA ("Select the second sphere"));
101   real dcent, dsurf;
102   dcent = sqrt(fabs((sph1.center.px-sph2.center.px)*(sph1.center.px-sph2.center.px)
103                    +(sph1.center.py-sph2.center.py)*(sph1.center.py-sph2.center.py)
104                    +(sph1.center.pz-sph2.center.pz)*(sph1.center.pz-sph2.center.pz)));
105   dsurf = dcent - sph1.rad - sph2.rad;
106   PRINT ("Distance between centers = %f", dcent);
107   PRINT ("Distance between surfaces = %f", dsurf);
108 ENDPROCEDURE
109
110 // Commands to input structured data (Point and Color)
111 EXPORT PROCEDURE ScreenSelectPoint () IS
112   Point p;
113   p = GetPoint (GETDATA("Select a point"));
114   atl_send_input (p);
115 ENDPROCEDURE
116
117 EXPORT PROCEDURE PointCoords () IS
118   Point p;
119   p.px = GETDATA ("X coordinate");
120   p.py = GETDATA ("Y coordinate");
121   p.pz = GETDATA ("Z coordinate");
122   atl_send_input (p);
123 ENDPROCEDURE
124
125 EXPORT PROCEDURE ScreenSelectColor () IS
126   ColorRGB col;
127   col = GetColor (GETDATA("Select the color"));
128   atl_send_input (col);
129 ENDPROCEDURE
```

```
130
131 EXPORT PROCEDURE ColorCoords () IS
132   ColorRGB c;
133   c.r = GETDATA ("Red value");
134   c.g = GETDATA ("Green value");
135   c.b = GETDATA ("Blue value");
136   atl_send_input (c);
137 ENDPROCEDURE
138
139 PROCEDURE main () IS
140   integer win, i;
141   win = GetWindowId ();
142   i = se::X_Control ("a",win,"ButtonPressMask","esferes");
143   IF (i<0) THEN
144     se::Sortida ("Error: Cannot manage the XEvents over that window","e");
145   ENDIF;
146   demandes::Add_Input_Method ("esferes::Point","esferes::ScreenSelectPoint ();",
147                               "Point selection",false);
148   demandes::Add_Input_Method ("esferes::Point","esferes::PointCoords ();",
149                               "Textual coordinates input",false);
150   demandes::Add_Input_Method ("esferes::ColorRGB","esferes::ScreenSelectColor ();",
151                               "Color selection",false);
152   demandes::Add_Input_Method ("esferes::ColorRGB","esferes::ColorCoords ();",
153                               "Textual color coordinates input",false);
154   AddDefault ();
155 ENDPROCEDURE
```

The automatically generated code

The automatically generated code for this example (produced from the previous .atl file) is in the permanent files atl_esferes.H, stub_esferes.H and atl_esferes.C; and in the temporary files _stubXXX.C, all of them listed below. For more information on this subject see also chapter 7.

atl_esferes.H

```
1  #ifndef __ATL_esferesH__
2  #define __ATL_esferesH__
3  #include "Variable.H"
4  #include "atl_se.H"
5  #include "atl_demandes.H"
6
7  typedef atl_Button_pressed_event atl_esferes_Button_pressed_event;
8
9  namespace esferes {
10 struct atl_Point {float px; float py; float pz;
11 atl_Point() {}
12 atl_Point(Variable &v) {
13    if (v.Arbre()==NULL) atl_exit(-1); // Invalid variable
14    px = ((nodereal *)(*(v.Arbre())).accedir(0))->Getvalor();
15    py = ((nodereal *)(*(v.Arbre())).accedir(1))->Getvalor();
16    pz = ((nodereal *)(*(v.Arbre())).accedir(2))->Getvalor();
17    }
18 operator Variable() {
19    Type t("esferes::Point","S(px real,py real,pz real)");
20    Variable v(t,"");
21    v.crea_arbre();
22    *((*(v.Arbre())).accedir(0)) = px;
23    *((*(v.Arbre())).accedir(1)) = py;
24    *((*(v.Arbre())).accedir(2)) = pz;
25    return (v);
26    }
27    };
28 }
29
30 namespace esferes {
31 struct atl_ColorRGB {float r; float g; float b;
32 atl_ColorRGB() {}
```

```
33  atl_ColorRGB(Variable &v) {
34     if (v.Arbre()==NULL) atl_exit(-1); // Invalid variable
35     r = ((nodereal *)(*(v.Arbre())).accedir(0))->Getvalor();
36     g = ((nodereal *)(*(v.Arbre())).accedir(1))->Getvalor();
37     b = ((nodereal *)(*(v.Arbre())).accedir(2))->Getvalor();
38     }
39  operator Variable() {
40     Type t("esferes::ColorRGB","S(r real,g real,b real)");
41     Variable v(t,"");
42     v.crea_arbre();
43     *((*(v.Arbre())).accedir(0)) = r;
44     *((*(v.Arbre())).accedir(1)) = g;
45     *((*(v.Arbre())).accedir(2)) = b;
46     return (v);
47     }
48     };
49  }
50
51  namespace esferes {
52  struct atl_Sphere {atl_Point center; float rad; atl_ColorRGB color;
53  atl_Sphere() {}
54  atl_Sphere(Variable &v) {
55     if (v.Arbre()==NULL) atl_exit(-1); // Invalid variable
56     {Variable v2("S(px real,py real,pz real)",""); v2.crea_arbre();
57      *(v2.Arbre())=*((*(v.Arbre())).accedir(0));atl_Point tpaux(v2);center=tpaux;}
58     rad = ((nodereal *)(*(v.Arbre())).accedir(1))->Getvalor();
59     {Variable v2("S(r real,g real,b real)",""); v2.crea_arbre();
60      *(v2.Arbre())=*((*(v.Arbre())).accedir(2));atl_ColorRGB tpaux(v2);color=tpaux;}
61     }
62  operator Variable() {
63     Type t("esferes::Sphere","S(center S(px real,py real,pz real),rad real,
64                                  color S(r real,g real,b real))");
65     Variable v(t,"");
66     v.crea_arbre();
67     *((*(v.Arbre())).accedir(0)) = *(((Variable) center).Arbre());
68     *((*(v.Arbre())).accedir(1)) = rad;
69     *((*(v.Arbre())).accedir(2)) = *(((Variable) color).Arbre());
70     return (v);
71     }
72     };
73  }
74
75  using esferes::atl_Point;
76  using esferes::atl_ColorRGB;
77  using esferes::atl_Sphere;
78
79  #endif
```

stub_esferes.H

```
1   #include "iodades.H"
2
3   #ifndef NOHEADER
4   #include "esferes.h"
5   #endif
6
7   void AddSphere(const nopar &);
8   io<ColorRGB> GetColor(const nopar &);
9   io<Point> GetPoint(const nopar &);
10  void RemoveSphere(const nopar &);
11  io<Sphere> GetSphere(const nopar &);
12  void ChangeColorSphere(const nopar &,const nopar &);
13  void ChangeColorSphere(const nopar &,const io<ColorRGB> &);
14  void ChangeColorSphere(const io<esferes_Button_pressed_event> &,const nopar &);
```

atl_esferes.C

```
1   #pragma implementation "taula.h"
2   #pragma implementation "Map.h"
3   #pragma implementation "VHMap.h"
4
5   #include "globals.H"
```

```
6   #include "ComunicDistr.H"
7   #include "Driver.H"
8   #include "String.h"
9   #include "gestio_crides.H"
10  #include "Variable.H"
11  #include "DLList.h"
12  #include "iodades.H"
13
14  #ifndef COMUNIC_DISTR
15  #define COMUNIC_DISTR Comunic_Distr
16  #endif
17  #include "inc_atlas.H"
18
19  String nomprogram;
20  gestio_crida_a_rutina gestor_crides_ext;
21
22  #include "stub_esferes.H"
23
24  #ifndef NOHEADER
25  #include "esferes.h"
26  #endif
27  #include "atl_esferes.H"
28
29  COMUNIC_DISTR distrib(CANAL_COMUNIC_DISTR);
30  Driver driv(distrib);
31
32  void aux_GetWindowId(const String & codi,const DLList<Variable *> &parametres) {
33     io_abstract *res;
34     res=new io<long int>(GetWindowId());
35     atl_tkt ticketaux=res->Ticket();
36     if (res->Containsdata()) {
37        long int restp;
38        restp=((io<long int> *)res)->Dades();
39        Variable *vr=new Variable(restp);
40        vr->AddTicket(ticketaux);
41        ReturnValue *rv=new ReturnValue(codi,vr);
42        distrib.envia(rv);
43        }
44     else {
45        Variable *vr=new Variable(ticketaux);
46        ReturnValue *rv=new ReturnValue(codi,vr);
47        distrib.envia(rv);
48        }
49     delete res;
50     }
51
52  void aux_AddSphere(const String & codi,const DLList<Variable *> &parametres) {
53     Pix p=parametres.first();
54     char indexpar=0;
55     atl_tkt ticketaux;
56     ticketaux=parametres(p)->Ticket();
57     io_abstract *iopar0;
58     if (parametres(p)->Gettype().Gettip()!="atl_ticket") {
59        atl_Sphere ptp0(*(parametres(p)));
60        Sphere par0(ptp0);
61        iopar0=new io<Sphere>(par0,ticketaux);
62        indexpar=(indexpar<<1); indexpar+=1;
63        }
64     else {
65        iopar0=new io_base(ticketaux);
66        indexpar=(indexpar<<1);
67        }
68     parametres.next(p);
69     switch (indexpar) {
70        case 0: AddSphere(*((io_base *)iopar0));
71                break;
72        case 1: AddSphere(*((io<Sphere> *)iopar0));
73                break;
74        }
75     ReturnVoid *rv=new ReturnVoid(codi);
76     distrib.envia(rv);
77     delete iopar0;
78     }
79
```

```
80  void aux_GetColor(const String & codi,const DLList<Variable *> &parametres) {
81     Pix p=parametres.first();
82     char indexpar=0;
83     atl_tkt ticketaux;
84     ticketaux=parametres(p)->Ticket();
85     io_abstract *iopar0;
86     if (parametres(p)->Gettype().Gettip()!="atl_ticket") {
87        atl_esferes_Button_pressed_event ptp0(*(parametres(p)));
88        esferes_Button_pressed_event par0(ptp0);
89        iopar0=new io<esferes_Button_pressed_event>(par0,ticketaux);
90        indexpar=(indexpar<<1); indexpar+=1;
91        }
92     else {
93        iopar0=new io_base(ticketaux);
94        indexpar=(indexpar<<1);
95        }
96     parametres.next(p);
97     io_abstract *res;
98     switch (indexpar) {
99        case 0: res=new io<ColorRGB>(GetColor(*((io_base *)iopar0)));
100               break;
101       case 1: res=new io<ColorRGB>(GetColor(*((io<esferes_Button_pressed_event> *)iopar0)));
102               break;
103       }
104    ticketaux=res->Ticket();
105    if (res->Containsdata()) {
106       atl_ColorRGB restp;
107       restp=((io<ColorRGB> *)res)->Dades().conversio_a_tipus_pont();
108       Variable *vr=new Variable(restp);
109       vr->AddTicket(ticketaux);
110       ReturnValue *rv=new ReturnValue(codi,vr);
111       distrib.envia(rv);
112       }
113    else {
114       Variable *vr=new Variable(ticketaux);
115       ReturnValue *rv=new ReturnValue(codi,vr);
116       distrib.envia(rv);
117       }
118    delete iopar0;
119    delete res;
120    }
121
122 void aux_GetPoint(const String & codi,const DLList<Variable *> &parametres) {
123    Pix p=parametres.first();
124    char indexpar=0;
125    atl_tkt ticketaux;
126    ticketaux=parametres(p)->Ticket();
127    io_abstract *iopar0;
128    if (parametres(p)->Gettype().Gettip()!="atl_ticket") {
129       atl_esferes_Button_pressed_event ptp0(*(parametres(p)));
130       esferes_Button_pressed_event par0(ptp0);
131       iopar0=new io<esferes_Button_pressed_event>(par0,ticketaux);
132       indexpar=(indexpar<<1); indexpar+=1;
133       }
134    else {
135       iopar0=new io_base(ticketaux);
136       indexpar=(indexpar<<1);
137       }
138    parametres.next(p);
139    io_abstract *res;
140    switch (indexpar) {
141       case 0: res=new io<Point>(GetPoint(*((io_base *)iopar0)));
142               break;
143       case 1: res=new io<Point>(GetPoint(*((io<esferes_Button_pressed_event> *)iopar0)));
144               break;
145       }
146    ticketaux=res->Ticket();
147    if (res->Containsdata()) {
148       atl_Point restp;
149       restp=((io<Point> *)res)->Dades().conversio_a_tipus_pont();
150       Variable *vr=new Variable(restp);
151       vr->AddTicket(ticketaux);
152       ReturnValue *rv=new ReturnValue(codi,vr);
153       distrib.envia(rv);
```

```
154         }
155      else {
156         Variable *vr=new Variable(ticketaux);
157         ReturnValue *rv=new ReturnValue(codi,vr);
158         distrib.envia(rv);
159         }
160      delete iopar0;
161      delete res;
162      }
163
164  void aux_RemoveSphere(const String & codi,const DLList<Variable *> &parametres) {
165      Pix p=parametres.first();
166      char indexpar=0;
167      atl_tkt ticketaux;
168      ticketaux=parametres(p)->Ticket();
169      io_abstract *iopar0;
170      if (parametres(p)->Gettype().Gettip()!="atl_ticket") {
171         atl_esferes_Button_pressed_event ptp0(*(parametres(p)));
172         esferes_Button_pressed_event par0(ptp0);
173         iopar0=new io<esferes_Button_pressed_event>(par0,ticketaux);
174         indexpar=(indexpar<<1); indexpar+=1;
175         }
176      else {
177         iopar0=new io_base(ticketaux);
178         indexpar=(indexpar<<1);
179         }
180      parametres.next(p);
181      switch (indexpar) {
182         case 0: RemoveSphere(*((io_base *)iopar0));
183                 break;
184         case 1: RemoveSphere(*((io<esferes_Button_pressed_event> *)iopar0));
185                 break;
186         }
187      ReturnVoid *rv=new ReturnVoid(codi);
188      distrib.envia(rv);
189      delete iopar0;
190      }
191
192  void aux_GetSphere(const String & codi,const DLList<Variable *> &parametres) {
193      Pix p=parametres.first();
194      char indexpar=0;
195      atl_tkt ticketaux;
196      ticketaux=parametres(p)->Ticket();
197      io_abstract *iopar0;
198      if (parametres(p)->Gettype().Gettip()!="atl_ticket") {
199         atl_esferes_Button_pressed_event ptp0(*(parametres(p)));
200         esferes_Button_pressed_event par0(ptp0);
201         iopar0=new io<esferes_Button_pressed_event>(par0,ticketaux);
202         indexpar=(indexpar<<1); indexpar+=1;
203         }
204      else {
205         iopar0=new io_base(ticketaux);
206         indexpar=(indexpar<<1);
207         }
208      parametres.next(p);
209      io_abstract *res;
210      switch (indexpar) {
211         case 0: res=new io<Sphere>(GetSphere(*((io_base *)iopar0)));
212                 break;
213         case 1: res=new io<Sphere>(GetSphere(*((io<esferes_Button_pressed_event> *)iopar0)));
214                 break;
215         }
216      ticketaux=res->Ticket();
217      if (res->Containsdata()) {
218         atl_Sphere restp;
219         restp=((io<Sphere> *)res)->Dades().conversio_a_tipus_pont();
220         Variable *vr=new Variable(restp);
221         vr->AddTicket(ticketaux);
222         ReturnValue *rv=new ReturnValue(codi,vr);
223         distrib.envia(rv);
224         }
225      else {
226         Variable *vr=new Variable(ticketaux);
227         ReturnValue *rv=new ReturnValue(codi,vr);
```

```
228         distrib.envia(rv);
229         }
230      delete iopar0;
231      delete res;
232      }
233
234  void aux_ChangeColorSphere(const String & codi,const DLList<Variable *> &parametres) {
235      Pix p=parametres.first();
236      char indexpar=0;
237      atl_tkt ticketaux;
238      ticketaux=parametres(p)->Ticket();
239      io_abstract *iopar0;
240      if (parametres(p)->Gettype().Gettip()!="atl_ticket") {
241         atl_esferes_Button_pressed_event ptp0(*(parametres(p)));
242         esferes_Button_pressed_event par0(ptp0);
243         iopar0=new io<esferes_Button_pressed_event>(par0,ticketaux);
244         indexpar=(indexpar<<1); indexpar+=1;
245         }
246      else {
247         iopar0=new io_base(ticketaux);
248         indexpar=(indexpar<<1);
249         }
250      parametres.next(p);
251      ticketaux=parametres(p)->Ticket();
252      io_abstract *iopar1;
253      if (parametres(p)->Gettype().Gettip()!="atl_ticket") {
254         atl_ColorRGB ptp1(*(parametres(p)));
255         ColorRGB par1(ptp1);
256         iopar1=new io<ColorRGB>(par1,ticketaux);
257         indexpar=(indexpar<<1); indexpar+=1;
258         }
259      else {
260         iopar1=new io_base(ticketaux);
261         indexpar=(indexpar<<1);
262         }
263      parametres.next(p);
264      switch (indexpar) {
265         case 0: ChangeColorSphere(*((io_base *)iopar0),*((io_base *)iopar1));
266                 break;
267         case 1: ChangeColorSphere(*((io_base *)iopar0),*((io<ColorRGB> *)iopar1));
268                 break;
269         case 2: ChangeColorSphere(*((io<esferes_Button_pressed_event> *)iopar0),*((io_base *)iopar1));
270                 break;
271         case 3: ChangeColorSphere(*((io<esferes_Button_pressed_event> *)iopar0),*((io<ColorRGB> *)iopar1));
272                 break;
273         }
274      ReturnVoid *rv=new ReturnVoid(codi);
275      distrib.envia(rv);
276      delete iopar0;
277      delete iopar1;
278      }
279
280  void ini_per_crides() {
281      gestor_crides_ext.lligar_nom_crida("GetWindowId",&aux_GetWindowId);
282      gestor_crides_ext.lligar_nom_crida("AddSphere",&aux_AddSphere);
283      gestor_crides_ext.lligar_nom_crida("GetColor",&aux_GetColor);
284      gestor_crides_ext.lligar_nom_crida("GetPoint",&aux_GetPoint);
285      gestor_crides_ext.lligar_nom_crida("RemoveSphere",&aux_RemoveSphere);
286      gestor_crides_ext.lligar_nom_crida("GetSphere",&aux_GetSphere);
287      gestor_crides_ext.lligar_nom_crida("ChangeColorSphere",&aux_ChangeColorSphere);
288      }
289
290  void main(int argc,char **argv) {
291      nomprogram=argv[0];
292      distrib.initialize(argv[1]);
293      ini_per_crides();
294      driv.set_name_program(nomprogram);
295      ini_process();
296      driv.Dispatch();
297      close(CANAL_COMUNIC_DISTR);
298      exit(0);
299      }
```

_stub04110baa.C

```
1    #include "iodades.H"
2    #include "inc_atlas.H"
3    #include "esferes.h"
4
5    void AddSphere(const nopar &) {
6       atl_send_error("A stub routine has been called",'e');
7       }
```

_stub04110caa.C

```
1    #include "iodades.H"
2    #include "inc_atlas.H"
3    #include "esferes.h"
4
5    io<ColorRGB> GetColor(const nopar &) {
6       atl_send_error("A stub routine has been called",'e');
7       io<ColorRGB> ret;
8       return ret;
9       }
```

_stub04110daa.C

```
1    #include "iodades.H"
2    #include "inc_atlas.H"
3    #include "esferes.h"
4
5    io<Point> GetPoint(const nopar &) {
6       atl_send_error("A stub routine has been called",'e');
7       io<Point> ret;
8       return ret;
9       }
```

_stub04110eaa.C

```
1    #include "iodades.H"
2    #include "inc_atlas.H"
3    #include "esferes.h"
4
5    void RemoveSphere(const nopar &) {
6       atl_send_error("A stub routine has been called",'e');
7       }
```

_stub04110faa.C

```
1    #include "iodades.H"
2    #include "inc_atlas.H"
3    #include "esferes.h"
4
5    io<Sphere> GetSphere(const nopar &) {
6       atl_send_error("A stub routine has been called",'e');
7       io<Sphere> ret;
8       return ret;
9       }
```

_stub04110gaa.C

```
1    #include "iodades.H"
2    #include "inc_atlas.H"
3    #include "esferes.h"
4
5    void ChangeColorSphere(const nopar &,const nopar &) {
6       atl_send_error("A stub routine has been called",'e');
7       }
8
9    void ChangeColorSphere(const nopar &,const io<ColorRGB> &) {
10      atl_send_error("A stub routine has been called",'e');
11      }
12
13   void ChangeColorSphere(const io<esferes_Button_pressed_event> &,const nopar &) {
14      atl_send_error("A stub routine has been called",'e');
15      }
```

The script file that compiles the temporary files and produces the library to be linked with the rest of the process' components is also automatically generated (see also section 7.5.1). In this example, the script generated is the following:

```
#!/bin/sh

ATLAS=/usr/usuaris/sig/mfairen/vonsai/Atlas/;
export ATLAS;
ACE=/homes/hsoftsol2/Atlas/ACE-4.4/ACE_wrappers/build-Linux;
export ACE;

echo 'Compiling stub files...'

g++ -I$ATLAS/include -I$ATLAS/include/suport -I$ACE/ -I$ACE/ace -c \
    -I/usr/usuaris/sig/mfairen/cactus/mesa/Mesa-3.0/include \
    /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110baa.C \
    -o /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110baa.o
rm /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110baa.C

g++ -I$ATLAS/include -I$ATLAS/include/suport -I$ACE/ -I$ACE/ace -c \
    -I/usr/usuaris/sig/mfairen/cactus/mesa/Mesa-3.0/include \
    /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110caa.C \
    -o /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110caa.o
rm /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110caa.C

g++ -I$ATLAS/include -I$ATLAS/include/suport -I$ACE/ -I$ACE/ace -c \
    -I/usr/usuaris/sig/mfairen/cactus/mesa/Mesa-3.0/include \
    /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110daa.C \
    -o /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110daa.o
rm /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110daa.C

g++ -I$ATLAS/include -I$ATLAS/include/suport -I$ACE/ -I$ACE/ace -c \
    -I/usr/usuaris/sig/mfairen/cactus/mesa/Mesa-3.0/include \
    /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110eaa.C \
    -o /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110eaa.o
rm /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110eaa.C

g++ -I$ATLAS/include -I$ATLAS/include/suport -I$ACE/ -I$ACE/ace -c \
    -I/usr/usuaris/sig/mfairen/cactus/mesa/Mesa-3.0/include \
    /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110faa.C \
    -o /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110faa.o
rm /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110faa.C

g++ -I$ATLAS/include -I$ATLAS/include/suport -I$ACE/ -I$ACE/ace -c \
    -I/usr/usuaris/sig/mfairen/cactus/mesa/Mesa-3.0/include \
    /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110gaa.C \
    -o /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110gaa.o
rm /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110gaa.C

echo 'Generating library...'

ld -shared -o libstbesferes.so \
   /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110baa.o \
   /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110caa.o \
   /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110daa.o \
   /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110eaa.o \
   /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110faa.o \
   /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110gaa.o

rm /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110baa.o \
   /usr/usuaris/sig/mfairen/vonsai/preves/esferes/_stub04110caa.o \
   /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110daa.o \
   /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110eaa.o \
   /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110faa.o \
   /usr/usuaris/sig/mfairen/vonsai/proves/esferes/_stub04110gaa.o
```

Notice that there are some compilation flags that cannot be known by the ATLAS code generator. These flags are passed to the script through an input option to the ATLAS code generator. The developer must use this option (`-f "$(FLAGS)"`) to include specific compilation flags to the generated script.

The implementation

The implementation of a process includes the code implementing those external routines declared in the interface and also the classes defining the exported types. It can also include any other necessary types and routines for the internals of the process.

The only compulsory file required by ATLAS in the implementation of a process is the header file (`esferes.h`). The source file can be only one (`esferes.C` in this example) or more than one linked together to produce the binary file (`esferes`). In this example there are also other auxiliary files that are listed also at the end of this section.

The `esferes.h` file defines the classes corresponding to the exported types in the ATL module. See specially the conversion methods to and from the *bridge types* (lines 16–18, 33–35, 56-64 and 78-83). There is also a class which does not correspond to an exported type, `SphereWindow`. It is used to create and manage the sphere world scene.

The `esferes.h` file also has to include the prototypes of the external routines defined in the `esferes.C` file (lines 134–142).

```
1     #ifndef _ESFERES_H_
2     #define _ESFERES_H_
3     #include <DLList.h>
4     #include "GLSIGWindow.h"
5     #include <inc_atlas.H>  // This includes the Atlas API prototypes
6     #include "atl_esferes.hh"
7
8     class Point
9     {
10        double px,py,pz;
11
12      public:
13        Point () {}
14        Point (float x, float y, float z)
15          { px = (double)x; py = (double)y; pz = (double)z; }
16        Point (atl_Point p) { px = (double)p.px; py = (double)p.py; pz = (double)p.pz; }
17        atl_Point conversio_a_tipus_pont ()
18          { atl_Point p; p.px = (float)px; p.py = (float)py; p.pz = (float)pz; return p; }
19        Point &operator = (const Point &p)
20          { px = p.px; py = p.py; pz = p.pz; }
21        double Px () const { return px; }
22        double Py () const { return py; }
23        double Pz () const { return pz; }
24    };
25
26    class ColorRGB
27    {
28        float r,g,b;
29
30      public:
31        ColorRGB () { r = 0.5; g = 0.5; b = 1.0; }
32        ColorRGB (float x, float y, float z) { r = x; g = y; b = z; }
33        ColorRGB (atl_ColorRGB c) { r = c.r; g = c.g; b = c.b; }
34        atl_ColorRGB conversio_a_tipus_pont ()
35          { atl_ColorRGB c; c.r = r; c.g = g; c.b = b; return c; }
36        ColorRGB &operator = (const ColorRGB &c)
37          { r = c.r; g = c.g; b = c.b; }
38        double R () { return r; }
39        double G () { return g; }
40        double B () { return b; }
41    };
42
```

```
43  class Sphere
44  {
45      Point center;
46      double rad;
47      ColorRGB color;
48
49    public:
50      Sphere () {}
51      Sphere (Point p, float r): center(p), rad(r), color() {}
52      Sphere (Point p, float r, ColorRGB c): center(p), rad(r), color(c) {}
53      Sphere (float x, float y, float z, float r): center(x,y,z), rad(r) {}
54      Sphere (float x, float y, float z, float rad, float r, float g, float b):
55        center(x,y,z), rad(r), color(r,g,b) {}
56      Sphere (atl_Sphere s): center(s.center), rad(s.rad), color(s.color) {}
57      atl_Sphere conversio_a_tipus_pont ()
58        {
59          atl_Sphere s;
60          s.center = center.conversio_a_tipus_pont ();
61          s.rad = rad;
62          s.color = color.conversio_a_tipus_pont ();
63          return s;
64        }
65      double Rad () { return rad; }
66      Point Center () { return center; }
67      ColorRGB Color () { return color; }
68      void ChangeColor (ColorRGB c) { color = c; }
69  };
70
71  class esferes_Button_pressed_event{
72      int window;
73      int pos_x,pos_y;
74      int button;
75
76    public:
77      esferes_Button_pressed_event() {};
78      esferes_Button_pressed_event(atl_esferes_Button_pressed_event vp)
79        {
80          window=vp.window;
81          pos_x=vp.pos_x; pos_y=vp.pos_y;
82          button=vp.button;
83        }
84      atl_esferes_Button_pressed_event conversio_a_tipus_pont()
85        {
86          atl_esferes_Button_pressed_event vp;
87          vp.window=window;
88          vp.pos_x=pos_x; vp.pos_y=pos_y;
89          vp.button=button;
90          return vp;
91        }
92      int Window() {return window;}
93      int Pos_x() {return pos_x;}
94      int Pos_y() {return pos_y;}
95      int Button() {return button;}
96  };
97
98  class SphereWindow : public GLSIGWindow
99  {
100     DLList<io<Sphere> *> scene;
101
102   public:
103     SphereWindow (): GLSIGWindow (10,10,600,600,"Spheres World")
104       {
105         // Selecting the expose events to be dealed by the process.
106         XSelectInput( dpy, win, StructureNotifyMask | ExposureMask );
107       }
108     SphereWindow (int x,int y, int w, int h, char *name, Display* disp = NULL):
109       GLSIGWindow (x,y,w,h,name,disp)
110       {
111         // Selecting the expose events to be dealed by the process.
112         XSelectInput( dpy, win, StructureNotifyMask | ExposureMask );
113       }
114     void Resize (int w, int h) { SetWidth(w); SetHeight(h); }
115     void AddSphere (Sphere &sph) { atl_tkt idsph (atl_get_ticket());
116                                    io<Sphere> *iosph = new io<Sphere>(sph,idsph);
```

```
117                                  scene.append(iosph); }
118     void Drawing ();
119     ~SphereWindow ()
120       {
121         for (Pix i=scene.first(); i!=0; scene.next(i))
122           { delete scene(i); scene.del(i); }
123       }
124     Sphere &WhichSphere (double pos_x, double pos_y);
125     atl_tkt RemoveSphere (double pos_x, double pos_y);
126     void RemoveSphere (atl_tkt tck);
127   protected:
128     void drawsphere (Sphere &sph);
129     void SphereWindow::subdivide (double *v1, double *v2, double *v3, long depth,
130                                   const Point &c, double r);
131     Pix which (double pos_x, double pos_y);
132 };
133
134 // Prototypes needed
135 long GetWindowId ();
136 void AddSphere (Sphere sph);
137 ColorRGB GetColor (esferes_Button_pressed_event ev);
138 Point GetPoint (esferes_Button_pressed_event ev);
139 void RemoveSphere (io<esferes_Button_pressed_event> ev);
140 void RemoveSphere (io_base &iob);
141 void ChangeColorSphere (esferes_Button_pressed_event ev, ColorRGB col);
142 Sphere GetSphere (esferes_Button_pressed_event ev);
143
144 #endif
```

The `esferes.C` file implements the external routines declared in the ATL module (lines 162–226) and also other necessary code. In this file we can emphasize three aspects:

- The redefinition of the API routine `ini_process` to make the required initializations of the process (lines 133–160).
- The adding of a new communication channel to the process driver in order to listen some X-events over the created X-window like the expose or resize events. This management requires the use of a new class `Comunic_Xevent` (defined in file `ComunicXevent.H`), and the use of the `Add_handler` method of the driver (see also chapter 7). This is shown in lines 8–11 and 156–159. The routine to manage the X-events is `TractarEvent` (lines 228–246). Notice that these events will not be known to ATLAS unlike the Button pressed events, that in this example are handled by `se` (see the "main" procedure in the ATL module). They obviously are not involved in the recovery of the process or in any journal replay.
- The use of the tickets substitution for the selection to remove a sphere. The "`RemoveSphere`" routine receives an `esferes_Button_pressed_event` and substitutes its ticket (line 199) by the one attached to the corresponding sphere (the process asked for this ticket to the system before -line 115 of the `esferes.h` file).

  The process also implements an overloading routine for the "`RemoveSphere`" that receives an *io_base* as a parameter. This routine (lines 203–207) is to be used in a reexecution of the process (see also section 8.1.3 –*Global data identification*).

```
1    #include <math.h>
2    #include "esferes.h"
3    #include "ComunicXevent.H"
4    #include "Driver.H"
5    #include <GL/glu.h>
6    #include "GLSIGWindow.h"
7
8    // the Atlas driver object.
9    extern Driver driv;
10   // comunication to be added to the Atlas driver.
11   Comunic_Xevent eventwin;
12
13   void normalize (double v[3], double r)
14   {
15     double d = sqrt (v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);
16     v[0] = v[0]*r/d; v[1] = v[1]*r/d; v[2] = v[2]*r/d;
17   }
18
19   void SphereWindow::subdivide (double *v1, double *v2, double *v3, long depth,
20                                 const Point &c, double r)
21   {
22     double v12[3], v23[3], v31[3];
23
24     if (depth == 0)
25     {
26       glBegin (GL_POLYGON);
27         glNormal3f ((float)(v1[0]/r),(float)(v1[1]/r),(float)(v1[2]/r));
28         glVertex3f ((float)(v1[0]+c.Px()),
29                     (float)(v1[1]+c.Py()),
30                     (float)(v1[2]+c.Pz()));
31         glNormal3f ((float)(v2[0]/r),(float)(v2[1]/r),(float)(v2[2]/r));
32         glVertex3f ((float)(v2[0]+c.Px()),
33                     (float)(v2[1]+c.Py()),
34                     (float)(v2[2]+c.Pz()));
35         glNormal3f ((float)(v3[0]/r),(float)(v3[1]/r),(float)(v3[2]/r));
36         glVertex3f ((float)(v3[0]+c.Px()),
37                     (float)(v3[1]+c.Py()),
38                     (float)(v3[2]+c.Pz()));
39       glEnd ();
40       return;
41     }
42     for (int i=0; i<3; i++)
43     {
44       v12[i] = (v1[i]+v2[i])/2;
45       v23[i] = (v2[i]+v3[i])/2;
46       v31[i] = (v3[i]+v1[i])/2;
47     }
48     normalize (v12, r); normalize (v23, r); normalize (v31, r);
49     subdivide (v1, v12, v31, depth-1, c, r);
50     subdivide (v2, v23, v12, depth-1, c, r);
51     subdivide (v3, v31, v23, depth-1, c, r);
52     subdivide (v12, v23, v31, depth-1, c, r);
53   }
54
55   // Drawing method for a Sphere
56   void SphereWindow::drawsphere (Sphere &sph)
57   {
58     double r = sph.Rad()*sin(63.43*PI/180.0);
59     double h = sph.Rad()*cos(63.43*PI/180.0);
60     double w = 72.0*PI/180.0;
61     double vertexs[12][3] =
62       { {0.,0.,sph.Rad()}, {0.,-r,h}, {r*sin(w),-r*cos(w),h}, {r*sin(2*w),-r*cos(2*w),h},
63         {r*sin(3*w),-r*cos(3*w),h}, {r*sin(4*w),-r*cos(4*w),h}, {0.,r,-h},
64         {-r*sin(w),r*cos(w),-h}, {-r*sin(2*w),r*cos(2*w),-h}, {-r*sin(3*w),r*cos(3*w),-h},
65         {-r*sin(4*w),r*cos(4*w),-h}, {0.,0.,-sph.Rad()} };
66     int tindices[20][3] =
67     { {0,1,2}, {0,2,3}, {0,3,4}, {0,4,5}, {0,5,1}, {1,9,2}, {2,10,3}, {3,6,4}, {4,7,5},
68       {5,8,1}, {9,10,2}, {10,6,3}, {6,7,4}, {7,8,5}, {8,9,1}, {10,9,11}, {6,10,11},
69       {7,6,11}, {8,7,11}, {9,8,11} };
70
71     // Material
72     float mat[] = { sph.Color().R(), sph.Color().G(), sph.Color().B(), 1. };
73     glMaterialfv (GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, mat);
74     for (int j=0; j<20; j++)
```

```
75      subdivide (&vertexs[tindices[j][0]][0], &vertexs[tindices[j][1]][0],
76                 &vertexs[tindices[j][2]][0], 3, sph.Center(), sph.Rad());
77  }
78
79  // Drawing method for a SphereWindow
80  void SphereWindow::Drawing ()
81  {
82    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
83    for (Pix i=scene.first(); i!=0; scene.next(i))
84      drawsphere (scene(i)->Dades());
85    glFlush ();
86    Swap ();
87  }
88
89  Pix SphereWindow::which (double pos_x, double pos_y)
90  {
91    double zmin = -30;
92    Pix aux = scene.first();
93
94    for (Pix i=scene.first(); i!=0; scene.next(i))
95      if ((((scene(i)->Dades()).Center().Px()-(scene(i)->Dades()).Rad()) < pos_x)
96          && (pos_x < ((scene(i)->Dades()).Center().Px()+(scene(i)->Dades()).Rad()))
97          && (((scene(i)->Dades()).Center().Py()-(scene(i)->Dades()).Rad()) < pos_y)
98          && (pos_y < ((scene(i)->Dades()).Center().Py()+(scene(i)->Dades()).Rad()))
99          && (scene(i)->Dades()).Center().Pz() > zmin)
100     { zmin = (scene(i)->Dades()).Center().Pz(); aux = i; }
101   return aux;
102 }
103
104 Sphere &SphereWindow::WhichSphere (double pos_x, double pos_y)
105 {
106   Pix aux = which (pos_x, pos_y);
107   return (scene(aux)->Dades());
108 }
109
110 atl_tkt SphereWindow::RemoveSphere (double pos_x, double pos_y)
111 {
112   Pix aux = which (pos_x, pos_y);
113   atl_tkt ticketaux = scene(aux)->Ticket();
114   delete scene(aux);
115   scene.del(aux);
116   return (ticketaux);
117 }
118
119 void SphereWindow::RemoveSphere (atl_tkt tck)
120 {
121   for (Pix i=scene.first(); i!=0; scene.next(i))
122     if (scene(i)->Ticket()==tck)
123     {
124       delete scene(i);
125       scene.del(i);
126       break;
127     }
128 }
129
130 // Global window for the process
131 SphereWindow spherewin;
132
133 // Initializations for the process.
134 void ini_process ()
135 {
136   float light[] = {0.5, 0.5, 0.5, 1.};
137   float lightspec[] = {0.2, 0.2, 0.2, 1.};
138
139   glClearColor (0.0, 0.0, 0.0, 1.0);   /* black color for the background */
140   glEnable (GL_DEPTH_TEST);       /* activating z-buffering */
141   // Enabling lighting
142   glEnable (GL_LIGHTING);
143   glLightfv (GL_LIGHT0, GL_AMBIENT, light);
144   glLightfv (GL_LIGHT0, GL_DIFFUSE, light);
145   glLightfv (GL_LIGHT0, GL_SPECULAR, lightspec);
146   glEnable (GL_LIGHT0);
147   // Orthogonal projection
148   glMatrixMode (GL_PROJECTION);
```

```
149   glLoadIdentity ();
150   glOrtho (-20., 20., -20., 20., 1., 60.);
151   // The user is always in the Z axe
152   glMatrixMode (GL_MODELVIEW);
153   glLoadIdentity ();
154   gluLookAt (0., 0., 30., 0., 0., 0., 0., 1., 0.);
155   spherewin.Drawing ();
156   // Adding a communications channel to the Atlas driver.
157   int X_cn = XConnectionNumber (spherewin.GetDisplay());
158   eventwin.set_handle(X_cn);
159   driv.Add_handler (eventwin);
160 }
161
162 long GetWindowId ()
163 {
164   return (spherewin.GetWindow());
165 }
166
167 void AddSphere (Sphere sph)
168 {
169   Sphere *ss = new Sphere(sph);
170   spherewin.AddSphere (*ss);
171   spherewin.Drawing ();
172 }
173
174 ColorRGB GetColor (esferes_Button_pressed_event ev)
175 {
176   float w = spherewin.GetWidth();
177   float h = spherewin.GetHeight();
178   Sphere &sph=spherewin.WhichSphere ((float)(ev.Pos_x()-(w/2.0))/(w/40.0),
179                                      (float)(ev.Pos_y()-(h/2.0))/-(h/40.0));
180   return (sph.Color());
181 }
182
183 Point GetPoint (esferes_Button_pressed_event ev)
184 {
185   float w = spherewin.GetWidth();
186   float h = spherewin.GetHeight();
187   Point p((float)(ev.Pos_x()-(w/2.0))/(w/40.0),
188           (float)(ev.Pos_y()-(h/2.0))/-(h/40.0), 0.0);
189   return (p);
190 }
191
192 void RemoveSphere (io<esferes_Button_pressed_event> ev)
193 {
194   float w = spherewin.GetWidth();
195   float h = spherewin.GetHeight();
196   atl_tkt tckaux;
197   tckaux = spherewin.RemoveSphere ((float)((ev.Dades()).Pos_x()-(w/2.0))/(w/40.0),
198                                    (float)((ev.Dades()).Pos_y()-(h/2.0))/-(h/40.0));
199   atl_substitute_ticket (ev.Ticket(),tckaux);
200   spherewin.Drawing ();
201 }
202
203 void RemoveSphere (io_base &iob)
204 {
205   spherewin.RemoveSphere (iob.Ticket());
206   spherewin.Drawing ();
207 }
208
209 void ChangeColorSphere (esferes_Button_pressed_event ev, ColorRGB col)
210 {
211   float w = spherewin.GetWidth();
212   float h = spherewin.GetHeight();
213   Sphere &sph=spherewin.WhichSphere ((float)(ev.Pos_x()-(w/2.0))/(w/40.0),
214                                      (float)(ev.Pos_y()-(h/2.0))/-(h/40.0));
215   sph.ChangeColor (col);
216   spherewin.Drawing ();
217 }
218
219 Sphere GetSphere (esferes_Button_pressed_event ev)
220 {
221   float w = spherewin.GetWidth();
222   float h = spherewin.GetHeight();
```

```
223   Sphere sph=spherewin.WhichSphere ((float)(ev.Pos_x()-(w/2.0))/(w/40.0),
224                                      (float)(ev.Pos_y()-(h/2.0))/-(h/40.0));
225   return sph;
226 }
227
228 // Treatment for the X-events coming from the X-window
229 void TractarEvent ()
230 {
231   XEvent e;
232   while (XPending(spherewin.GetDisplay()))
233     {
234       XNextEvent (spherewin.GetDisplay(), &e);
235       if ((e.type == ConfigureNotify) || (e.type == Expose))
236         spherewin.Drawing ();
237       if (e.type == ConfigureNotify)
238       {
239         spherewin.Resize (e.xconfigure.width, e.xconfigure.height);
240         glMatrixMode(GL_PROJECTION);
241         glViewport(0,0,spherewin.GetWidth(),spherewin.GetHeight());
242         glMatrixMode(GL_MODELVIEW);
243         spherewin.Drawing ();
244       }
245     }
246 }
```

ComunicXevent.H

```
1    #ifndef _COMUNICXEVENT_H
2    #define _COMUNICXEVENT_H
3    #include <sys/types.h>
4    #include <sys/socket.h>
5    #include "SOCK_Stream.h"
6    #include "INET_Addr.h"
7    #include "config.h"
8    #include "Reactor.h"
9    #include "globals.H"
10   #include "Message.H"
11
12   extern void TractarEvent ();
13
14   class Comunic_Xevent : public ACE_Event_Handler
15   {
16       int filedesc;
17
18     public:
19       Comunic_Xevent () {}
20       Comunic_Xevent (int fd) : filedesc(fd) {}
21       void set_handle (ACE_HANDLE fd) { filedesc = fd; }
22       ACE_HANDLE get_handle () const { return filedesc; }
23       int handle_input (ACE_HANDLE fd) { TractarEvent ();
24                                          return 0; }
25   };
26
27   #endif
```

GLSIGWindow.h

```
1    #ifndef __GLSIGWINDOW_H_
2    #define __GLSIGWINDOW_H_
3    #include <iostream.h>
4    #include <stdio.h>
5    #include <stdlib.h>
6    #include <GL/glx.h>             /* this includes the necessary X headers*/
7    #include <GL/gl.h>
8
9    class GLSIGWindow
10   {
11     public:
12       static Display         *dpy;
13
14     protected:
15       Window          win;
16       Colormap        cmap;
```

```
17      XSetWindowAttributes swa;
18      static XVisualInfo    *vi;
19      GLXContext      cx;
20      int x,y,w,h;
21      char *name;
22
23    public:
24      GLSIGWindow(int x,int y, int w, int h, char *name, Display* disp = NULL);
25      virtual ~GLSIGWindow();
26      void        SetPos( int x, int y ) { XMoveWindow( dpy, win, x, y ); }
27      void        SetPosX(int posX) { x = posX; }
28      void        SetPosY(int posY) { y = posY; }
29      void        SetWidth( int weight )  { w=weight; }
30      void        SetHeight( int hight )  { h=hight; }
31      void        SetName( char* name );
32      // Consultors
33      int         GetPosX() { return x; }
34      int         GetPosY() { return y; }
35      int         GetWidth() { return w; }
36      int         GetHeight() { return h; }
37      Display*    GetDisplay() { return dpy; }
38      Window      GetWindow() { return win; }
39      static int  GetScreenWidth();
40      static int  GetScreenHeight();
41
42      operator   Window() { return win; }      // Conversion Operator
43      void       Activate() { glXMakeCurrent(dpy, win, cx); }
44      void       Swap() { glXSwapBuffers(dpy,win); }
45
46    private:
47      Window     MakeRGBdbWindow( Display *dpy, int x, int y, int width, int height );
48  };
49
50  #endif  // GLSIGWINDOW
```

GLSIGWindow.cc

```
1   #include "GLSIGWindow.h"
2
3   XVisualInfo* GLSIGWindow::vi;
4   Display* GLSIGWindow::dpy;
5
6   static Bool WaitForMapNotify(Display *d, XEvent *e, char *arg)
7   {
8     if ((e->type == MapNotify) && (e->xmap.window == (Window)arg))
9       return GL_TRUE;
10    return GL_FALSE;
11  }
12
13  GLSIGWindow::GLSIGWindow(int x,int y, int w, int h, char* name, Display* dsp)
14              : x(x), y(y), w(w), h(h), name(name)
15  {
16    // Obtenim un display si no ens el passen
17    if(!dsp)
18      dpy = XOpenDisplay(NULL);
19    else
20      dpy = dsp;
21    win = MakeRGBdbWindow( dpy, x, y, w, h );
22    int argc=0; char **argv;
23    XSetStandardProperties(dpy, win, name, name, None, argv, argc, NULL);
24    XSizeHints    size_hints;
25    XWMHints      wm_hints;
26    XClassHint    class_hints;
27    XTextProperty windowName;
28
29    size_hints.flags      = USPosition | PMaxSize | PMinSize;
30    size_hints.min_width  = 50;
31    size_hints.min_height = 50;
32    size_hints.max_width = GetScreenWidth();
33    size_hints.max_height = GetScreenHeight();
34    if (XStringListToTextProperty (&name, 1, &windowName) == 0)
35      cerr << ": structure allocation for windowName failed." << endl;
36    wm_hints.initial_state = NormalState;
37    wm_hints.input         = True;
```

```
38    wm_hints.flags         = StateHint | InputHint;
39    class_hints.res_name   = "";
40    class_hints.res_class  = "Basicwin";
41    XSetWMProperties (dpy, win, &windowName, &windowName, &name, 1,
42                      &size_hints, &wm_hints, &class_hints);
43    XEvent event;
44    XMapWindow(dpy, win);
45    XIfEvent(dpy, &event, WaitForMapNotify, (char *)win);
46  }
47
48  GLSIGWindow::~GLSIGWindow()
49  {
50    glFlush();
51    glFinish();
52    glXMakeCurrent(dpy, 0, NULL);
53    glXDestroyContext(dpy, cx);
54    // Destruim la finestra
55    XDestroyWindow(dpy, win);
56    XFlush(dpy);
57  }
58
59  Window GLSIGWindow::MakeRGBdbWindow (Display *dpy, int x, int y, int width, int height)
60  {
61    int attrib[] = { GLX_RGBA, GLX_RED_SIZE, 1, GLX_GREEN_SIZE, 1,
62                     GLX_BLUE_SIZE, 1, GLX_DEPTH_SIZE, 16, GLX_DOUBLEBUFFER,
63                     GLX_STENCIL_SIZE, 1, None };
64    int scrnum;
65    XSetWindowAttributes attr;
66    unsigned long mask;
67    Window root;
68    Window win;
69
70    scrnum = DefaultScreen( dpy );
71    root = RootWindow( dpy, scrnum );
72    vi = glXChooseVisual( dpy, scrnum, attrib );
73    if (!vi)
74      {
75        cout << "Error: couldn't get an RGB, Double-buffered visual" << endl;
76        exit(1);
77      }
78    /* window attributes */
79    attr.background_pixel = 0;
80    attr.border_pixel = 0;
81    attr.colormap = DefaultColormap (dpy, DefaultScreen(dpy));
82    attr.event_mask = StructureNotifyMask | ExposureMask;
83    mask = CWBackPixel | CWBorderPixel | CWColormap | CWEventMask;
84    win = XCreateWindow( dpy, root, x, y, width, height, 0, vi->depth, InputOutput,
85                         vi->visual, mask, &attr );
86    cx= glXCreateContext( dpy, vi, NULL, GL_TRUE );
87    glXMakeCurrent( dpy, win, cx );
88    return win;
89  }
90
91  void GLSIGWindow::SetName( char* name )
92  {
93    XTextProperty windowName;
94
95    if (XStringListToTextProperty (&name, 1, &windowName) == 0)
96      cout << ": structure allocation for windowName failed." << endl;
97    XSetWMName(dpy, win, &windowName);
98  }
99
100 int GLSIGWindow::GetScreenWidth()
101 {
102   return DisplayWidth(dpy, DefaultScreen(dpy));
103 }
104
105 int GLSIGWindow::GetScreenHeight()
106 {
107   return DisplayHeight(dpy, DefaultScreen(dpy));
108 }
```

Makefile

```
1  ATLAS_ROOT = /usr/usuaris/sig/mfairen/vonsai/Atlas
2
3  #-----------------------------------------------------------------------------
4  #       Include ATLAS macros
5  #-----------------------------------------------------------------------------
6
7  include         $(ATLAS_ROOT)/macros/macros_atlas.$(ARQ)
8
9  #--------
10 # The ARQ variable is a parameter for the make command.
11 # You must call this Makefile doing:
12 #       $> gmake (or make) ARQ=xxxx
13 # where xxxx is the set of characters correspondig to the architecture
14 # you are using to compile your process (Sun5, IRIX6, HP10 or Linux).
15 #--------
16
17 #-----------------------------------------------------------------------------
18 #       Local macros
19 #-----------------------------------------------------------------------------
20
21 LATLASFLAGS = -L$(LIBATLAS) -lobjsatl -latlas $(LDLIBS)
22 OBJ = $(VDIR)atl_esferes.o $(VDIR)esferes.o $(VDIR)GLSIGWindow.o
23
24 MESA = /usr/usuaris/sig/mfairen/cactus/mesa/Mesa-3.0
25 MESAFLAGS = -I$(MESA)/include
26 MESALIB = -L$(MESA)/lib -lMesaGLU -lMesaGL
27
28 XLIBS = -L/usr/X11/lib -L/usr/X11R6/lib -lX11 -lXext -lXmu -lXt -lXi -lSM -lICE
29
30 #-----------------------------------------------------------------------------
31 #       Local targets
32 #-----------------------------------------------------------------------------
33
34 all:    objdirs esferes
35
36 atl_esferes.C atl_esferes.H: esferes.atl
37         $(ATLAS_ROOT)/bin/generador -f "$(MESAFLAGS)" esferes.atl
38
39 $(VDIR)GLSIGWindow.o: GLSIGWindow.cc
40         $(COMP.cc) $(MESAFLAGS) -o $(VDIR)GLSIGWindow.o GLSIGWindow.cc
41
42 $(VDIR)esferes.o: esferes.C
43         $(COMP.cc) $(MESAFLAGS) -o $(VDIR)esferes.o esferes.C
44
45 $(VDIR)atl_esferes.o : atl_esferes.C
46         $(CTEMP.cc) $(MESAFLAGS) -o $(VDIR)atl_esferes.o atl_esferes.C
47
48 esferes: $(OBJ)
49         $(LINK.cc) -o $@ $(OBJ) $(MESALIB) $(LATLASFLAGS) $(XLIBS) -L./ -lstbesferes
50
51
52 clean:
53         rm -f $(VDIR)*.o core atl_esferes.* stub_esferes.H esferes
54
55 objdirs: $(VDIR)
56
57 $(VDIR):
58         test -d $@ || mkdir $@
```

9.5 Applications using ATLAS

ATLAS is being extensively used in our lab. In the following subsections we present four nontrivial applications that have been developed entirely in ATLAS (*NewDMI*, *Motlles*) or ported to ATLAS (*Octrees*, *VolAtlas*).

These experiences confirm that ATLAS offers a friendly environment to develop this kind of applications; programmers quickly became familiar with it, and porting old applications to ATLAS required small efforts.

9.5.1 *VolAtlas:* a volume modeling application

VolAtlas is a platform for the modeling and visualization of volume models which uses the voxels model representation. This representation consists of a regular subdivision of the volume region in identical cubic cells (voxels) parallel to the coordinate axes. Each voxel has an associated value for each represented property.

The *VolAtlas* platform is subdivided in different processes and the operations applicable to the voxels model are: visualization, filtering, creation of voxels models from other voxels models, generation of isosurfaces using different algorithms, etc... A snapshot of this application can be seen in figure 9.8.

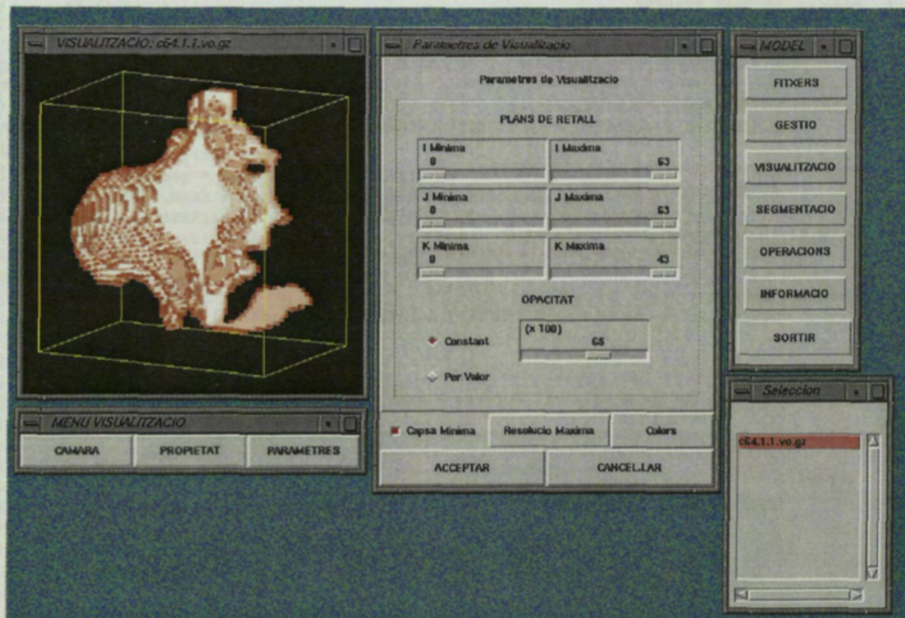


Figure 9.8: Snapshot of the *VolAtlas* application.

9.5.2 *Octrees:* a solid modeling application using extended octrees

The extended octree model is an extension of the classic octree model which includes new terminal nodes (face, edge, vertex and quasivertex). These new nodes contain a part of the solid boundary and the octree exactly represents the polyhedrical objects.

The *Octrees* application offers the main functionalities of the extended octree: conversion between solid models in BRep and extended octree, boolean operations among extended octrees, compression of the model and visualization.

The application has been ported to ATLAS and offers commands to communicate with the BRep modeling application *NewDMI.* Figure 9.9 shows a snapshot of this application.

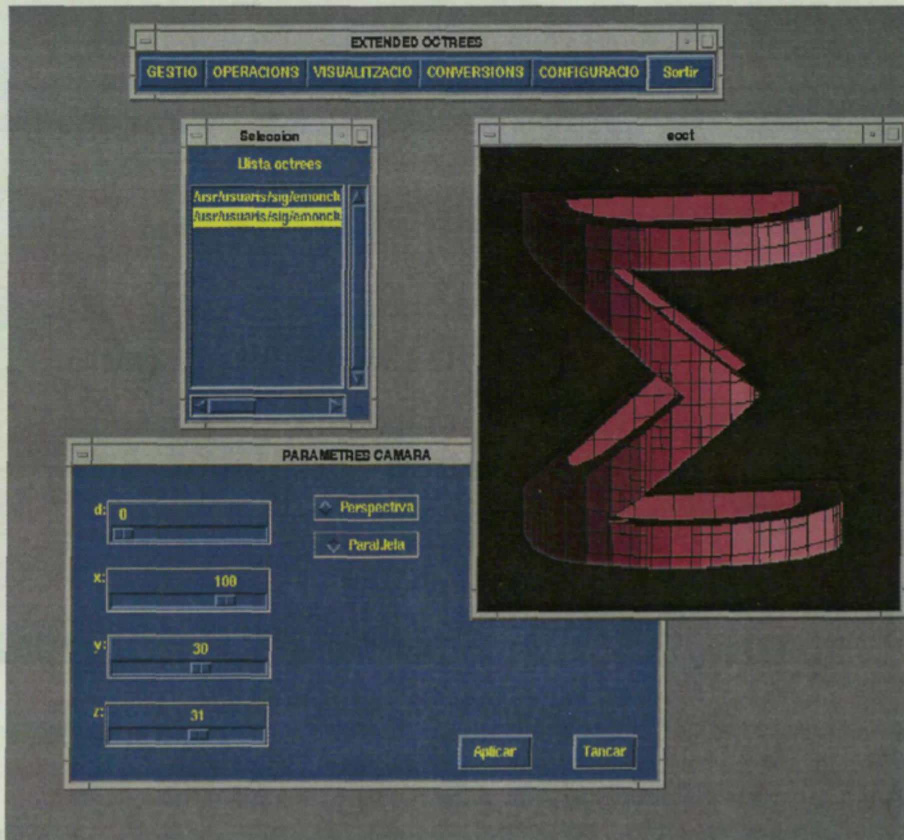


Figure 9.9: Snapshot of the *Octrees* application.

9.5.3 *NewDMI:* a BRep modeling application

NewDMI is a geometric kernel of a CAD system. It uses a geometric boundary representation (BRep) to model objects. It supports basic parametric elements (cylinder, sphere, cone, ...) to create more complex objects and can also manage translation and rotation sweeps. It is specialized on the ship design (valves, pipes, motors, ...).

NewDMI offers commands to create new objects, to edit the scene (add, remove, or select objects in the scene), to visualize the scene and to operate with the scene (zooms, sections 3D and 2D, ...). Figure 9.10 shows a snapshot of this application.

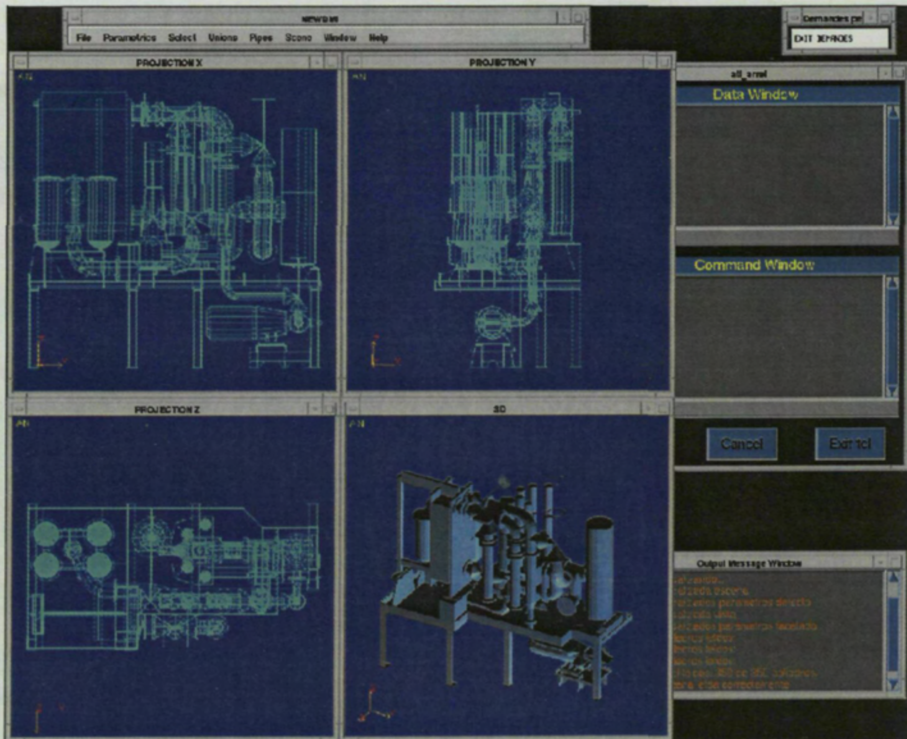


Figure 9.10: Snapshot of the *NewDMI* application.

9.5.4 *Motlles:* a CAD system for plastic injection moulds

The *Motlles* application is a CAD system of solids and surfaces adapted to mould applications and with an output connected to a finite elements analysis module (CAE).

This application is oriented to the design of injection moulds. These moulds can be seen as the negative of a piece. The piece may have a complicated shape,

so it must be represented as a combination of plane faces and surface faces. The application uses a boundary representation (BRep) which can keep faces of both plane and surface types. A snapshot of the application can be seen in figure 9.11.

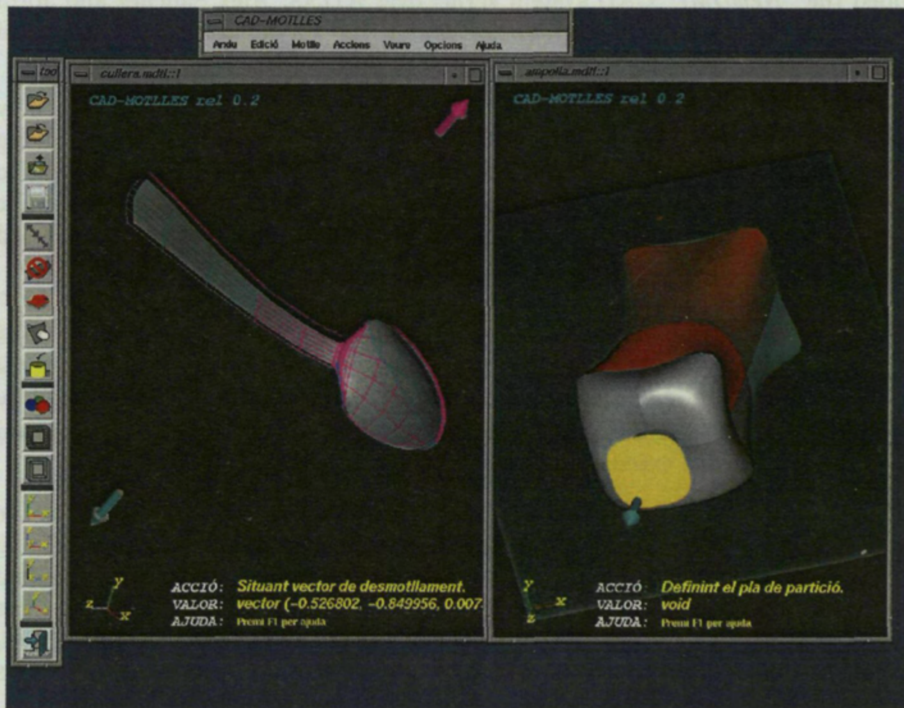


Figure 9.11: Snapshot of the *Motlles* application.

9.6 Evaluation of the system. The developers opinions

To evaluate a system like ATLAS, oriented to the development of large applications, the development of a toy application like the *"esferes"* presented in section 9.4 is not enough. Moreover, some ATLAS objectives like transparency, easiness of use or usefulness in large applications development, are aspects to be evaluated by other developers not directly related to ATLAS. Our own opinion in these aspects is not enough either.

A possible evaluation process thus is getting the opinions and experiences of real ATLAS users, i.e. people who use ATLAS as a platform for developing their own applications. We built a questionnaire asking for advantages and drawbacks found in the use of ATLAS as a development platform. We want to thank Eva Monclús, David Corbalán, Àlex Sánchez and Jordi Martín for helping

us on doing this evaluation. They are not all the developers that have used, and therefore tested ATLAS, but they are those who have used it the most.

There are four large applications developed over ATLAS and briefly presented in the last section (section 9.5). Eva Monclús ported both *"Octrees"* and *"VolAtlas"* to ATLAS. David Corbalán was the first ATLAS user and developed the *"NewDMI"* CAD system kernel over ATLAS. Àlex Sánchez and Jordi Martín both worked on the *"Motlles"* application (Àlex started it and Jordi joined later).

In this section we summarize the results of the questionnaire answers given by these ATLAS users. They did not have the opportunity of testing the journaling part of ATLAS (its structure and functionalities), because it has been added to the prototype during the last year and it is not available yet in the ATLAS public distribution. Taking this into account, the comments received can be grouped in the following aspects:

- *Rigidity of the* ATLAS *Input Subsystem.* All interviewed ATLAS users agreed in criticizing the rigidity of this subsystem which does not allow to input complex data directly and whose interface is not very friendly nor flexible. We are aware of these limitations and have already proposed some possible extensions to this Input Subsystem in chapter 5.
- *Initial fragility and instability.* Most of these users also complain for the initial fragility and instability of the system, and the difficulty of debugging application processes without a specific tool to allow it (a process could not be debugged directly because `distr` wants to hear from processes periodically). Since ATLAS and those applications grew up in parallel, they had to use the first beta (and sometimes alpha) versions of ATLAS which produced these inconveniences.
- *Something missing in the ATL.* David also considered some extensions to the ATL functionalities were missing, specially giving support to other kind of types like lists.
- *Facilitates the development.* Another general opinion is that using ATLAS facilitates the development in terms of testing new application functionalities, because its development does not require any added interface and can be very easily included in the application in order to be tested.
- *Allows an easy communication between different applications.* They also approve the easiness of communicating an application to others, even when the application is not split in different processes. In fact, the *"NewDMI"* and *"Octrees"* applications (see section 9.5) are both implemented in only one application process and adding a command to communicate both of them to allow *"NewDMI"* to ask for a boolean operation to *"Octrees"*, took just about 5 minutes to implement[2].

  When the application consists of several processes, this easiness is more effective. Referring to *"VolAtlas"*, Eva Monclús said: "this division in

[2] The routines offering the functionalities needed by the *"NewDMI"* were already implemented in *"Octrees"*, so the only work to do was implementing the interaction between the two applications in the ATL module.

independent and small tasks allows the final user to decide which ones he is going to use, thus he only starts those processes implementing these tasks and avoids to start the whole VolAtlas platform. In our case, using ATLAS is also positive because some projects developed by students about volume modelling and visualization can be included to the *VolAtlas* platform as new functionalities in a surprisingly easy way."

- *Distribution and communications transparency.* Other positive aspect found in ATLAS by the users is the transparency of communications and distribution. Sometimes they just use one architecture while developing, but they do not care about how many processes ATLAS is using (except for those belonging to their applications) or how these processes communicate to each other.

From these developers comments we can conclude some results which are clearly positive even when ATLAS did not offer yet the journaling functionalities, which are expected to be the ATLAS most valuable facilities for the computer graphics applications' world:

- ATLAS is evaluated positively in terms of transparency and easiness of use for the developer;
- except for the first versions fragility, developers agreed on its usefulness (so they prefer using ATLAS better than not using it);
- although the applications presently using it are not related to each other, they show a certain level of reusability (in VolAtlas, for example, the students use its processes and environment to test easily their implementations are working).

As already said, there are some important functionalities in ATLAS that developers could not test yet because they were still not available in the public version they use. Although this new part of ATLAS has been only tested by our small applications, a real test in large applications would be desirable. We plan it will be tested from now on in the new applications which are going to use ATLAS as well as in the new developments for the old ones.

Chapter 10

Extensions

During the development of ATLAS, we realized that some functionalities can be extended, either to make them more flexible or to offer added facilities to the developer.

Even though these extensions cannot be included as part of ATLAS at the moment because they are not fully designed and incorporated to the system, tentative designs for them have been thought. We think they are worth mentioning here.

10.1 Synchronous requests of input data from application processes

As said in section 9.2, the current version of ATLAS does not allow an application process to stop and wait for an input data previously requested by this process using the "`atl_send_request`" API call[1]. The main execution of an ATLAS application process is guided by its communications driver, who listens to one or more communications channels and acts depending on the received message.

Since the ATLAS asynchronous "input datum↔request" association does not assure an immediate answer for a given request, the process driver is not designed to stop waiting for a response to a certain request.

A specific design is thus necessary to allow this synchronous management of data requests from an application process.

The design thought to add this possibility to ATLAS is similar to the "select/recv" system calls to manage communication channels in BSD sockets over TCP/IP. The process will use the "`atl_send_request`" API call to ask for data asynchronously as it is used presently. But to get the requested data a new API routine "`atl_get_response`" will be called. This new routine will be able to act

[1]This does not means that currently an application over ATLAS cannot implement a synchronous management of data, but it cannot do it directly from an application process. A synchronous management can be done by asking these data from an ATL command (explained in chapter 4).

waiting or *not waiting* for the datum. *Not waiting* means it checks if the datum has arrived and if not, it keeps on with the process execution. *Waiting* means the process checks if the datum has arrived and if not, it *"stops"* its execution waiting for the requested data to arrive. This *stopping* of the process will not be complete. A treatment allowing messages from `distr` to arrive will remain, but not handling them, just storing them to be treated later, except for the expected datum.

10.2 Overloading in ATL

Extending the ATL language to permit functions or procedures to be overloaded seems a not very difficult aim.

The only needed change in the compiler is on the function and procedure name management. Just by adding the parameter type names as part of the routine name on the symbol table, the ATL language will be able to deal with overloading.

But other important requirement for ATLAS supporting this overloading is that the automatic code generator should be able to manage it.

The generated code should have an auxiliary routine (see section 7.5.1) for each function or procedure instance of the overloading, but all of them (related to the same exported C++ routine) must call the same process routine (which will use the C++ overloading functionality), passing to it the corresponding parameters for this instance.

10.3 Extending the GETDATA command to accept timeouts for its requests

As we already explained in chapter 6, when a data request expires its timeout without getting input data to serve it, a null response is generated to answer this request. This null response has information about the request it is serving (contains the request identifier) but does not contain data.

When the request is sent by an application process, and since the management of the answer is implemented by the application developer, his code is responsible of being able to treat a null response if it arrives.

But when the request is done by the ATLAS `Command Subsystem` (through a GETDATA in the ATL code -see section 4.1.6 - *The GETDATA function*) the Virtual Machine of the `Command Subsystem` is the one receiving the response and acting to use it for keeping on with the command execution. The Virtual Machine thus needs a value of that type to continue its execution without problems.

The extension proposal in this case is to add two more parameters to the GETDATA command. The first one will be the timeout value for the request being sent and the second one will be a default value to be used in case the answer is a null response. This two added parameters will be optional to facilitate using the GETDATA without timeout.

10.4 The Command Substitution possibility

In an interactive interpreter like the ATLAS Command Subsystem, offering the possibility to substitute a user command already being executed could be interesting. The possibility of substituting commands can be useful mainly in two cases:

- in case the user made a mistake choosing an option to input data. He wanted to input data through a certain command and chose the wrong one;
- in case the application does this choosing by default and in some cases the user does not want to follow the default.

Imagine a user is working with an application that displays a 3D scene and allows to select a point by entering either a *MouseButtonPressed* event over a certain display pixel, which will select the point nearer this pixel in the displayed scene; or the exact 3D coordinates of the point in the scene, which will select exactly that point. The user wants to select a point which is not visible in the displayed scene because it is hidden behind other objects. But the user fails when he has to choose the command to be used to get these data and he chooses the one asking for a *MouseButtonPressed* event. Without the possibility of substituting the command started, the user only can keep on with this command and after done use the UNDO functionality of the journaling (see section 8.1.3) to go back un-doing the command.

The design we present in this section for a command substitution will solve problems like this and will also add flexibility because having this substitution possibility the developer can decide to put a default command (which would start automatically) in places where the user would almost always use this default.

Not all started commands can be substituted, only those that have not changed yet the global state of the Virtual Machine. A command can be substituted thus if at the time of substitution:

- the command has not modified any global variable;
- the command has not modified any parameter passed by reference to itself;
- the command has not called any external function or procedure.

Any command fulfilling these conditions when a substitution is attempted can be aborted. After this abortion the user can choose another command to substitute the first one.

In fact the substitution itself is not compulsory in this design. Since the conditions to be able to abort a command only depend on the command to abort, the user can ask for the interruption of a command even if he does not intend to start another command to substitute the one aborted. This will depend, of course, on the application.

The only way to select a command to be aborted is by choosing the input request generated by the command, so this can be added to the "demandes" utility process as a sort of mouse selection over the request, or by adding an API routine passing to it the request identifier, that is thus usable from any process knowing about the request identifier (subscribed to the ADD_DEMAND ATLAS event, for example).

If the request selected was not produced by a command in the Virtual Machine it cannot be aborted.

10.5 Miscellaneous

There are other miscellaneous extensions to the current prototype which are worth mentioning.

- *The management of errors in the starting of a process.* In order to make ATLAS more robust in starting an application, and also be consistent with the broadcast mechanism and the process execution (see chapter 6), an error management is required when a process cannot be started in the chosen host.

  There are two different problems to deal with:

  - the chosen host is not available;
  - the exec for the process called by the server on the chosen host fails.

  In the first case, distr should try to start the process again by choosing another host which offers this process. If there is not another host offering this process the error management ends and an error message is given to the user.

  In the second case, the reason of fault can be the lack of availability of the process in the specified path (the one with highest priority). distr then should try again in the same host (which is the best in terms of load coefficient –see chapter 6) but specifying the path which follows the previous one in the priority list (see also *"Process-Host table"* in chapter 6).

  Since the decision of giving more importance to the host–choosing than to the path priority is heuristic and it can be restrictive for the final user, we can also make this more flexible by allowing the user to configure the change of priorities for ATLAS executions. By adding another section in the ".AtlasSettings" file the user may indicate the reverse on this priorities, so ATLAS would change the host before the process path.

- *More flexibility on the syntax of the hosts list.* The list of hosts the user may include in the ".AtlasSettings" file is used to filter the hosts that can be involved in his applications (see chapter 6). The user may want to associate processes to hosts, allowing one host to be only used for executing those processes or disallowing it to execute some processes. The current syntax for the list of hosts only permits to put host names in, so when a

host is in the list any process being offered by this host can be executed there.

The extension proposal for this syntax is very simple: for each host the user can add a list of process names that will act as a filter for the processes allowed on that host. Each process name may be preceded by the character '!' which means this process will not be executed on the host. As an example, if we have in the ".AtlasSettings" file the following lines:

```
$HOSTS
hostname1:hostname2{proc1,proc2,proc3}:hostname3{!proc2}
```

it means the only hosts allowed are `hostname1`, `hostname2` and `hostname3`; and for `hostname2` and `hostname3` the list of processes is also restricted. In `hostname2` ATLAS will only execute `proc1`, `proc2` and `proc3` and `hostname3` is allowed to execute any process except `proc2`.

Having this possibility the user has much more flexibility to decide the distribution of his application processes.

Chapter 11

Conclusions and future work

In this thesis a software platform, ATLAS, has been presented. It is designed to allow developers to incorporate advanced features in their development with the least hassle, and it achieves all the objectives proposed for the thesis (described in chapter 1).

By now, ATLAS is being extensively used within our lab in the development of large applications for some research projects. Its design favours the construction of reusable modules that can relatively easily be combined with each other. It has also proved successful for the developers who use it, because they quickly get familiar with its use and take profit on its functionalities without much effort.

Although at a first glance ATLAS seems to be very similar to other DOC systems, it offers other functionalities like the journaling mechanism, execution time configurability, asynchronous calls, generic user interface, etc. All these are designed in together with the communications mechanism and the processes' distribution in order to achieve the maximum efficiency by combining them to solve the whole problem. The scheme of figure 11.1 can help see the relations among the adopted solutions to solve the problem as a tight combination of the desired objectives.

The *input data management* shown as an ATLAS concept includes three managements of input data explained separately in the thesis:

- The generic Input Subsystem (explained in chapter 5).
- The *asynchronous matching* functionality assigning input data to requests done by `distr` (explained in section 6.3).
- The *global data identification* done also by `distr` to be a functionality of the journaling (explained in section 8.1.3).

In *other tools* we include the tools to allow the advanced user to overrule some defaults or use ATLAS at a lower level to gain control of how things are handled.

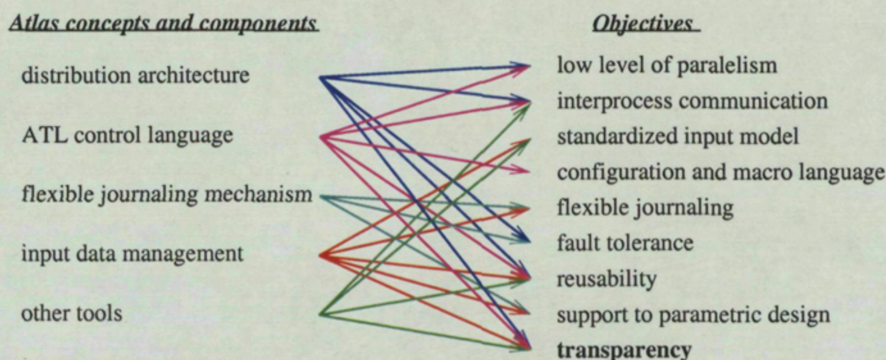


Figure 11.1: Relations among ATLAS concepts and objectives

These tools are the ATLAS events mechanism (explained in section 6.4), the ATLAS API library (explained in section 9.2) and the utility processes (explained in section 9.3).

The independent solutions adopted to address the different objectives are not new on their own. The contributions of this thesis are, instead, in seeking and displaying a solution to all of the aforementioned objectives jointly and harmoniously; also in showing means by which this can be attained with little or no performance cost and with minimal special training of the programmers. Finally, the design of our journaling system is to our knowledge original in the simultaneous solution of all of its requirements.

Future work

In the ATLAS architecture an extension to make the distribution of processes work over a WAN is easy[1]. This makes us think in the possibility of extending the architecture to allow ATLAS applications to support CSCW.

Once an application has been built upon ATLAS, it can be turned into a CSCW-supporting application 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 a master. This, although limited, would turn essentially every ATLAS application into a CSCW-capable application, with no or extremely little effort by the developers, as per ATLAS requirements (see figure 11.2).

If we consider the ATLAS processes as objects, another interesting feature would be adding the possibility of creating more than one instance of these objects at a time. The work needed includes the extension of the capabilities of the ATL language to permit the management of multiple instances of a process.

Thinking on the sort of applications ATLAS is addressed to (split in several processes being executed in different machines), it is possible, and even likely,

[1] This is currently being undertaken by some users for a state-wide project

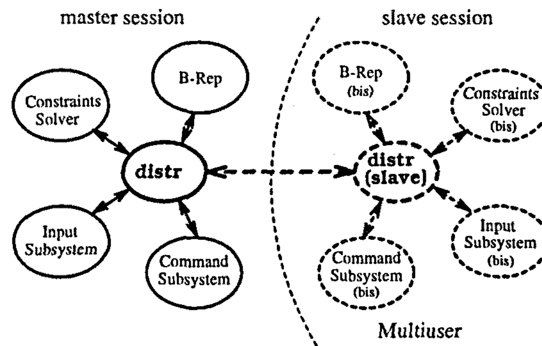


Figure 11.2: Possible extension to a CSCW-supporting architecture

to find situations where different versions of a process are available to be executed. Another future task thus may go in this direction, adding to ATLAS the possibility of controlling and supporting different versions of processes and other files as well.

As stated in section 2.2, we also plan to eventually develop a generic parametric solver service for ATLAS based on other work developed in our group.

Finally, a more ambitious line of extension would entail the adaption of ATLAS' kernel to use threads when available.

Bibliography

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

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

[3] Pere Pau Vázquez and Pere Brunet. The Multientity Definition Textual Language (MDTL) geometric format. Specification and API. *Report LSI-99-2-T*, 1999.

[4] Jon Siegel. *CORBA Fundamentals and Programming.* OMG, 1996.

[5] Zhonghua Yang and Keith Duddy. Distributed Object Computing with CORBA. (A State-of-the-Art Report on OMG/CORBA). *ACM Operating System Reviews*, 30(2), April 1996.

[6] Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications*, 14(2), 1997.

[7] Mark Roy and Alan Ewald. Inside DCOM., April 1997. http://www.dbmsmag.com/9704d13.html.

[8] Troy Downing. *Java RMI. Remote Method Invocation.* IDG publishers, 1997.

[9] Kenneth P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12), 1993.

[10] R. Van Renesse, K.P. Birman, and S. Maffeis. HORUS: A Flexible Group Communication System. *Communications of the ACM*, 39(4), 1996.

[11] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-system for High Availability. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[12] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos. Totem: A Fault-tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4), 1996.

[13] K.P. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th Annual Symposium on Operating Systems Principles*, pages 123–138, November 1987.

[14] Jean-Charles Fabre and Tanguy Prennou. A Metaobject Architecture for Fault-Tolerant Distributed Systems: The FRIENDS Approach. *IEEE Transactions on Computers*, 47(1), 1998.

[15] S.K. Shrivastava, G.N. Dixon, and G.D. Parrington. An Overview of Arjuna: A Programming System for Reliable Distributed Computing. *IEEE Software*, 8(1):63–73, 1991.

[16] A Geist, A. Bequelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. PVM: Parallel Virtual Machine. A User's Guide and Tutorial Networked Parallel Computing. *http://www.netlib.org/pvm3/book/pvm-book.html.*

[17] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Parallel Languages and Systems*, 7(1), 1985.

[18] W. T. O'Connell, G. K. Thiruvathukal, and T. W. Christopher. Distributed Memo: A Heterogeneously Distributed and Parallel Software Development Environment. In *Proceedings of the 23rd International Conference on Parallel Processing*, St. Charles, IL., August 1994.

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

[20] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes, and Richard L. Phillips. *Introduction to Computer Graphics.* Addison-Wesley, 1994.

[21] Michael Barborak, Miroslaw Malek, and Anton Dahbura. The Consensus Problem in Fault-Tolerant Computing. *ACM Computing Surveys*, 25(2), June 1993.

[22] Felix C. Gartner. Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. *ACM Computing Surveys*, 31(1), March 1999.

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

[24] Thomas Lord. The Guile Architecture for Ubiquitous Computing. In *Proceedings of the Usenix Tcl/Tk Workshop*, 1995.

[25] Larry Wall, Tom Christiansen, Randal L. Schwartz, and Stephen Potter. *Programming Perl (2nd Edition).* O'Reilly & Associates, 1996.

[26] Mark Lutz. *Programming Python. Object-Oriented scripting.* O'Reilly & Associates, 1996.

[27] Terrence J. Parr. Language Translation Using PCCTS and C++ (A Reference Guide), June 1995. Address: http://www.parr-research.com/ parrt.

[28] Ken Arnold and James Gosling. *The Java Programming Language Second Edition.* Addison-Wesley, 1998.

[29] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1998.

[30] Douglas 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.

[31] Douglas C. Schmidt. The ADAPTIVE communication environment: Object-oriented network programming components for developing client/server applications. In *12th Sun Users Group Conference*, 1994.

[32] H. Rex Hartson and Debora Hix. Human–Computer Interface Development: Concepts and Systems for Its Management. *ACM Computing Surveys*, 21(1), March 1989.

[33] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[34] *SUN: Network programming guide.* Chapters 10,11 and 12: A Socked-Based Interprocess Communications.

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

[36] 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.

Appendix A

ATL Grammar Description

```
modul            ←   ( bloc )+ Eof
bloc             ←   ( "EXPORT" ( deftip
                                | {"ASYNC"} ( procedure | function )
                                | var_decl_exp ";" )
                     | deftip
                     | proto
                     | use
                     | unuse
                     | reuse
                     | procedure
                     | function
                     | "ASYNC" ( procedure
                               | function
                               | func_call ) ";"
                     | name ( var_d ( "," var_d )* ";"
                            | func_call_trunc ";" )
                     | var_decl1 ";'
                     | ";" )
deftip           ←   "#deftype" { "::" } IDENTIFIER ( tipus | name | "_atl_unknown")
tipus            ←   ( "STRUCT"
                          IDENTIFIER "->" ( tipus | name ) ";"
                          ( IDENTIFIER "->" ( tipus | name ) ";" )*
                       "ENDSTRUCT"
                     | "VECTOR" ( "[" NUMBER "]" )+ "OF" ( tipus | name ) )
proto            ←   "PROT" ( prototipus )+ "ENDPROT"
prototipus       ←   ( "EXPORT" {"ASYNC"} ( prot_proc | prot_func ) ";"
                     | "EXTERN" {"ASYNC"} ( prot_proc | prot_func ) ";"
                     | {"ASYNC"} ( prot_proc | prot_func ) ";"
                     | ";" ).
prot_proc        ←   "PROCEDURE" ( prot_proc_main
                                 | { "::" } IDENTIFIER "(" param_list ")" )
                          { "INVERSE OF" name }
prot_proc_main   ←   "main" "(" ")"
```

```
prot_func      ←   "FUNCTION" { "::" } IDENTIFIER "(" param_list ")"
                         "RETURNS" name
                         { "INVERSE OF" name }
use            ←   "USE" IDENTIFIER ( "," IDENTIFIER )* ";"
unuse          ←   "UNUSE" IDENTIFIER ( "," IDENTIFIER )* ";"
reuse          ←   "REUSE" IDENTIFIER ( "," IDENTIFIER )* ";"
procedure      ←   "PROCEDURE" ( procmain
                                 | { "::" } IDENTIFIER "(" param_list ")" "IS"
                                       sentences "ENDPROCEDURE" )
procmain       ←   "main" "(" ")" "IS"
                         sentences "ENDPROCEDURE" )
function       ←   "FUNCTION" { "::" } IDENTIFIER "(" param_list ")"
                        "RETURNS" name "IS"
                           sentences
                   "ENDFUNCTION"
param_list     ←   { param ( "," param )* }
param          ←   name { "&" } IDENTIFIER
var_decl_exp   ←   ( var_decl1 | name var_d ( "," var_d )* )
var_decl1      ←   tipus var_d ( "," var_d )*
var_d          ←   { "::" } IDENTIFIER
sentences      ←   ( conditional
                   | condmult
                   | iteration
                   | inprocess
                   | name ( assign_trunc
                          | var_d ( "," var_d )*
                          | func_call_trunc ) ";"
                   | func_call_async ";"
                   | retorn ";"
                   | escriu ";' '
                   | var_decl1 ";"
                   | ";" )+
conditional    ←   "IF" expressio "THEN" sentences
                             { "ELSE" sentences }
                   "ENDIF"
condmult       ←   "CASE" expressio "IS"
                         ( "WHEN" constant "DO" sentences )*
                         { "OTHERWISE" sentences }
                   "ENDCASE"
iteration      ←   ( "WHILE" expressio "DO" sentences "ENDWHILE"
                   | "FOR" "(" assignment ";" expressio ";" assignment ")"
                          "DO" sentences
                     "ENDFOR" )
assignment     ←   name assign_trunc
assign_trunc   ←   variable_trunc "="
                             ( expressio
                             | func_call_async
                             | "GETDATA" "(" expressio { "," expressio } ")" )
inprocess      ←   "WITH" IDENTIFIER "DO" sentences "ENDWITH"
```

```
func_call_async   ←   "ASYNC" func_call
func_call         ←   name func_call_trunc
func_call_trunc   ←   "(" expr_list ")"
expr_list         ←   { ( expressio
                        | "GETDATA" "(" expressio { "," expressio } ")" )
                        ( "," ( expressio
                                | "GETDATA" "(" expressio { "," expressio } ")" ) )* }
retorn            ←   "RETURN" { expressio }
escriu            ←   "PRINT" "(" STRING "," variable ")"
variable          ←   name variable_trunc
variable_trunc    ←   { acces }
acces             ←   ( "." IDENTIFIER
                      | "[" expressio "]" )+
name              ←   ( PREFIX IDENTIFIER
                      | IDENTIFIER
                      | "::" IDENTIFIER )
expressio         ←   expcmp ( "&&" expcmp | "||" expcmp )*
expcmp            ←   e1 ( "==" e1 | "! =" e1 | "<" e1 | "<=" e1 | ">" e1 | ">=" e1 )
e1                ←   e2 ( "+" e2 | "−" e2 )*
e2                ←   e3 ( "*" e3 | "/" e3 | "%" e3 )*
e3                ←   ( "{" name "}" e4
                      | e4 )
e4                ←   { ( "−" | "!" ) } ( constant
                                        | "(" expressio ")"
                                        | name ( variable_trunc
                                                 | func_call_trunc ) )
constant          ←   ( number | string | boolea )
number            ←   ( NUMBER { "." { NUMBER } }
                      | "." NUMBER
string            ←   STRING
boolea            ←   ( ( "TRUE" | "true" )
                      | ( "FALSE" | "false" ) )


NUMBER                [0-9]+
STRING                \" (~[\"] | \\\")* \"
PREFIX                [a-zA-Z][a-zA-Z0-9_]*::
IDENTIFIER            [a-zA-Z][a-zA-Z0-9_]*
```

Appendix B

Intermediate code instructions

This appendix describes all the intermediate code instructions that the ATLAS Virtual Machine accepts. For each instruction it explains also its operands and the effect of its execution[1].

POP	$\Longrightarrow$	Instruction code = 0 This instruction is used to discard elements from the temporary stack. It removes the element at the top of the temporary stack. It does the 'pop' and deletes the contents of the operand.
PUSH PUSH op1	$\Longrightarrow$	Instruction code = 1 This instruction can be used with or without operand. If it goes with an operand it pushes this operand in the temporary stack.

The following instructions take the operands (one or two depending on the operation) from the top of the temporary stack removing them from the stack (executing the "Buida" method for each one when done) and push the operation result onto the stack. The operands have been pushed into the stack in the order they have been read (parsed), so accessing to the top of the stack we find first the second operand and after that the first one.

[1]The implementation behaviour is described changing to the sans serif font in order to distinguish it from the more abstract explanation level.

ADDI ADDF ADDS	⟹	Instruction codes = 2, 17 i 28 respectively. These instructions do the algebraic addition of two operands in cases of integers (I) or reals (F). In case of string operands (S) it does the concatenation of the two operands. The result has always the same type as the operands. Constant operands are cast into the corresponding subclass (Operand_int –I–, Operand_float –F– or Operand_string –S–) and variables are cast into Operand_var. The result is always a constant operand.
SUBI SUBF	⟹	Instruction codes = 3 i 18 respectively. These instructions compute the difference between two operands of type integer (I) or real (F) –first operand minus second operand–. The result is always of the same type as the operands. Constant operands are cast into the corresponding subclass (Operand_int –I– or Operand_float –F–) and variables are cast into Operand_var. The result is always a constant operand.
MULI MULF	⟹	Instruction codes = 4 i 19 respectively. These instructions compute the product of two operands of types integer (I) or real (F). The result is always of the same type as the operands. Constant operands are cast into the corresponding subclass (Operand_int –I– or Operand_float –F–) and variables are cast into Operand_var. The result is always a constant operand.
DIVI DIVF	⟹	Instruction codes = 5 i 20 respectively. These instructions compute the division of two operands of types integer (I) or real (F) –first operand divided by second operand–. The result is always of the same type as the operands. Constant operands are cast into the corresponding subclass (Operand_int –I– or Operand_float –F–) and variables are cast into Operand_var. The result is always a constant operand.
MOD	⟹	Instruction code = 6 This instruction computes the reminder of the integer division between two integer operands. The result is also an integer. Constant operands are cast into Operand_int and variables are cast into Operand_var. The result is always an Operand_int.

NEGI
NEGF

$\Longrightarrow$ Instruction codes = 7 i 21 respectively.
These instructions change the sign of an operand of type integer (I) or real (F). The result has the same type as the operand.
Constant operands are cast into the corresponding subclass (Operand_int –I– or Operand_float –F–) and variables are cast into Operand_var. The result is always a constant operand.

EQI
EQF
EQS
EQB

$\Longrightarrow$ Instruction codes = 8, 22, 29 i 31 respectively.
These instructions check if two operands of type integer (I), real (F), string (S) or boolean (B) are equal (have the same value). The result is always a boolean.
Constant operands are cast into the corresponding subclass (Operand_int –I–, Operand_float –F–, Operand_string –S– or Operand_bool –B–) and variables are cast into Operand_var. The result is always an Operand_bool.

NEQI
NEQF
NEQS
NEQB

$\Longrightarrow$ Instruction codes = 9, 23, 30 i 32 respectively.
These instructions check if two operands of type integer (I), real (F), string (S) or boolean (B) are not equal (have different values). The result is always a boolean.
Constant operands are cast into the corresponding subclass (Operand_int –I–, Operand_float –F–, Operand_string –S– or Operand_bool –B–) and variables are cast into Operand_var. The result is always an Operand_bool.

LEI
LEF

$\Longrightarrow$ Instruction codes = 10 i 24 respectively.
These instructions check if the first operand, integer (I) or real (F), is smaller than the second. The result is always a boolean.
Constant operands are cast into the corresponding subclass (Operand_int –I– or Operand_float –F–) and variables are cast into Operand_var. The result is always an Operand_bool.

LEQI
LEQF

$\Longrightarrow$ Instruction codes = 11 i 25 respectively.
These instructions check if the first operand, integer (I) or real (F), is smaller than or equal to the second. The result is always a boolean.
Constant operands are cast into the corresponding subclass (Operand_int –I– or Operand_float –F–) and variables are cast into Operand_var. The result is always an Operand_bool.

GTI
GTF

⟹ Instruction codes = 12 i 26 respectively.
These instructions check if the first operand, integer (I) or real (F), is greater than the second. The result is always a boolean.
Constant operands are cast into the corresponding subclass (Operand_int –I– or Operand_float –F–) and variables are cast into Operand_var. The result is always an Operand_bool.

GEQI
GEQF

⟹ Instruction codes = 13 i 27 respectively.
These instructions check if the first operand, integer (I) or real (F), is greater than or equal to the second. The result is always a boolean.
Constant operands are cast into the corresponding subclass (Operand_int –I– or Operand_float –F–) and variables are cast into Operand_var. The result is always an Operand_bool.

AND

⟹ Instruction code = 14
This instruction does the boolean operation AND between two boolean operands. The result is also a boolean.
Constant operands are cast into an Operand_bool and variables are cast into Operand_var. The result is also an Operand_bool.

OR

⟹ Instruction code = 15
This instruction does the boolean operation OR between two boolean operands. The result is also a boolean.
Constant operands are cast into an Operand_bool and variables are cast into Operand_var. The result is also an Operand_bool.

NOT

⟹ Instruction code = 16
This instruction inverts the value of a boolean operand. The result is also a boolean.
Constant operands are cast into an Operand_bool and variables are cast into Operand_var. The result is also an Operand_bool.

The MOV instructions assign a value to a variable. Both the value and the variable are at the top of the temporary stack (the value is the one at the top). Once the assignment is done both operands are removed from the stack.

MOVI
MOVF
MOVS
MOVB

⟹ Instruction codes = 33, 34, 35 i 36 respectively. These instructions assign values of basic types: integer (I), real (F), string (S) or boolean (B). Constant operands (for the operand to be moved) are cast into the corresponding subclass (Operand_int –I–, Operand_float –F–, Operand_string –S– or Operand_bool –B–) and variables are cast into Operand_var. The operand where the value must be moved is always an Operand_var and the method to use can be a *Setvalor* or *SetIO* depending on the operand to be assigned (constant or io_abstract pointer).

MOVN

⟹ Instruction code = 37
This instruction assigns to a variable the value of another variable or part of it.
The two operands are io_abstract pointers (Operand_var) and the assignment is done by using the method *SetIO*.

BRF op1

⟹ Instruction code = 38
This is a branch instruction if the condition is false. It checks the condition value at the top of the temporary stack and if it is false it jumps to the instruction of the code table with the index indicated at the operand 'op1'.
Constant operands are cast into Operand_bool and variables are cast into Operand_var but they are not popped from the stack.
The operand 'op1' is an Operand_int and it is used only if the value of the condition is false, causing in this case the change of the execution index to this value.

BR op1

⟹ Instruction code = 39
This is the unconditional branch instruction. It always jumps to the instruction of the code table with the value received at the operand 'op1' as index.
The operand 'op1' is an Operand_int and the execution index is directly changed to this integer value.

ACCS op1 $\Longrightarrow$ Instruction code = 40
This instruction reads a field of a structure. It takes the node pointer of the structure from the top of the temporary stack and access to the field having as index the value received into the operand 'op1'. The resulting node pointer is pushed also to the temporary stack.
The operand 'op1' is an Operand_int and the operand at the top of the temporary stack is an Operand_var which will be removed from the stack. The value of the node pointer in the Operand_var (inside the `io<node *>`) is used to call the 'node' method *accedir* passing it the value of the index given by 'op1'. The result of this method is the node pointer to be pushed into the temporary stack as a result of this accessing instruction.

ACCV $\Longrightarrow$ Instruction code = 41
This instruction reads an element of a vector. It takes from the temporary stack both the node pointer of the vector and the index of the element to access. The resulting node pointer is pushed to the temporary stack.
The index for the element to access is at the top of the stack. Constant operands are cast into Operand_int and variables are cast into Operand_var. After this index the temporary stack has the vector node pointer. Both operands will be removed from the stack. In case the method *accedir* of the vector node pointer returns NULL it means the index is not inside the range for this vector and an execution error finishing this execution will be triggered; otherwise the node pointer returned is pushed onto the stack (inside the `io<node *>` of an Operand_var).

ERR $\Longrightarrow$ Instruction code = 46
This instruction detects whether a function execution has not gone through any return instruction. At this point the execution must produce an error because it is a function without a return sentence.

CINT op1 op2 ⟹ Instruction code = 42

This instruction calls to a function or procedure internal to the Command Subsystem. The operand 'op1' has the address in the symbol table corresponding to the function or procedure to be called (because of the preprocessing of this instruction). The operand 'op2' has a boolean indicating if the call is asynchronous. The instruction first checks if the code table is defined, producing an execution error if not. If the code table is defined and it is a synchronous call it pushes to the execution stack the address of the code table being executed and the current index to this table and the base pointer for the current activation bloc as well. When the call is asynchronous it creates a new *ExecStep* using a duplicate of the temporary stack for this new *ExecStep* (see section 4.3). Once this information has been stored the called function or procedure code table is executed.

The code table address is pushed as an Operand_punter and the execution index and the base pointer as Operand_int. In order to start the execution of the called code table its address is going to be the current code table address, the current index is set to -1 (at the end of the current loop it will become 0) and the present activation bloc base pointer is set to the current length of the execution stack.

RET ⟹ Instruction code = 43

This instruction indicates a return of a function or a procedure.

First of all it must remove from the execution stack those operands used by the local variables and parameters of this function or procedure (the contents of those operands have been deleted before by the execution of instructions RMV). After this it takes from the execution stack the base pointer of the previous activation bloc and the return address (code table and index) where the execution control has to return. This values are going to be considered as the present execution values.

Finally the execution status (ExecStep) must be updated with these new data.

VRBL op1 op2 op3 $\Longrightarrow$ Instruction code = 44

This instruction is used whenever a local variable is created. It pushes a new variable onto the execution stack. The operands received are respectively the name, the type and the final type of the variable to be created.

All operands are string constants (Operand_string). A *Type* object is created from the type and the final type of the variable, and a *Variable* object is created from the *Type* and the name of the variable. The operand pushed to the execution stack is an Operand_ptrvar containing the pointer to this *Variable*. The node tree is not created at this point.

GETV op1 op2 $\Longrightarrow$ Instruction code = 45

This instruction access to a local variable. The operand 'op1' is the offset of this variable in the execution stack and the operand 'op2' is a boolean indicating whether the access to the variable is as an l-value or not (this is usable for the dirty variables mechanism described in section 4.3). The instruction access to the variable and pushes it into the temporary stack.

The operand 'op1' is an integer constant (Operand_int) used to compute the offset in the execution stack to access to the variable. The operand 'op2' is a boolean constant (Operand_bool). Once the variable has been accessed in the execution stack (Operand_ptrvar), the instruction checks whether the node tree of the variable is already created or not and it creates it if it is needed. The node pointer is then pushed to the temporary stack (Operand_var).

PARV op1 op2 op3 $\Longrightarrow$ Instruction code = 47

This instruction is used to create local variables to store parameters passed by value. It pushes a new variable onto the execution stack. The operands given to the instruction are respectively the name, the type and the final type of the parameter to be created.

All three operands are string constants (Operand_string). It creates a *Type* object from the type and the type definition, and a *Variable* object from the *Type* and the name. The tree of the variable is created and the address of this *Variable* is pushed to the execution stack (Operand_ptrvar –containing a *Variable* pointer).

SETV op1 op2 ⟹ Instruction code = 48
This instruction sets the initial value of a parameter. The operand 'op1' is the offset in both stacks (temporary and execution) and the operand 'op2' indicates the type of the node of the parameter. Both operands 'op1' and 'op2' are integer constants (Operand_int). It access to the execution stack taking the node pointer where the parameter value must be set. If this value (accessing to the temporary stack) is constant it must be of a basic type and depending on the type indicated by 'op2' it takes the correct operand and assigns it to the variable node pointer. If the value is a node pointer the type indicated by 'op2' can be also a structure or a vector and the assignment must be done by using the node copy operator (with the corresponding cast).

PARR op1 op2 op3 ⟹ Instruction code = 49
This instruction is used to create local variables to store parameters passed by reference. It pushes a new variable onto the execution stack. The operands given to the instruction are respectively the name, the type and the final type of the parameter to be created.
All three operands are string constants (Operand_string). It creates a *Type* object from the type and the type definition, and a *Variable* object from the *Type* and the name. Then the address of this *Variable* is pushed to the execution stack (Operand_ptrvar –containing a *Variable* pointer).
It does not create the tree of the variable because the parameter is passed by reference and will be set with the node pointer passed as a parameter.

SETR op1 ⟹ Instruction code = 50
This instruction sets the parameter passed by reference to the corresponding pointer. The operand 'op1' is the offset in both stacks (temporary and execution) to the correct parameter.
The operand 'op1' is an integer constant (Operand_int). From the temporary stack (and the 'op1' offset) it gets the node pointer which is the value for the parameter. This is assigned to the *Variable* for the parameter at the execution stack (with the 'op1' offset). It must be noticed that the variable cannot have the tree created because the node pointer being assigned becomes this tree, shared by both.

RMV op1 ⟹ Instruction code = 51
It removes the variable at the offset indicated by 'op1' from the execution stack.
The operand 'op1' is an integer constant (Operand_int). The instruction deletes (executing its destructor) the variable in this position of the execution stack. It does not do the pop of this variable. It will be done by the RET instruction.

CEXT op1 op2 op3 ⟹ Instruction code = 52
This instruction is to execute an external function call. At 'op1' it receives the name of the external function to be called, at 'op2' the number of parameters for this function and at 'op3' a boolean indicating if the call is asynchronous.
The three operands 'op1', 'op2' and 'op3' are constants, the first one string, the second one integer and the third one boolean (Operand_string, Operand_int and Operand_bool). From the first one the instruction can extract the name of the external process and the name of the routine. With this information, the number of parameters and the number assigned to this call (given from a global variable being incremented any time it is used), it generates a routine call message (*CallRoutine*) which will be sent to the Command Subsystem driver. After this an object keeping the information for this call (*OpenedCall*, with the name and the number of the routine called –see subsection *"External routine calls"* in chapter 4) is created and is appended to the corresponding list depending on the value of 'op3' (synchronous calls list or asynchronous calls list –see also section 4.3).

GETN opl op2 ⟹ Instruction code = 53

This instruction pushes onto the temporary stack an io_abstract pointer containing the node pointer corresponding to a global variable that is in the symbol table.

The operand 'opl' contains the address in the symbol table containing the global variable to be accessed ('opl' has this address because of the preprocessing to this instruction). The operand 'op2' is a boolean indicating whether the access to the variable is as an l-value or not (usable for the dirty variables mechanism described in section 4.3). First of all it checks that the variable is defined (it may have been removed after its declaration), giving an error if it does not. If there is no error it gets the value of the node pointer of the *Variable* tree (creating it if needed) and pushes it onto the temporary stack (Operand_var).

REQD opl op2 ⟹ Instruction code = 54

This instruction generates a request of input data of the type indicated by operand 'opl'. The operand 'op2' is a boolean indicating if there is a timeout value for this request.

Both operands 'opl' and 'op2' are constants of types string (Operand_string) and boolean (Operand_bool) respectively. If the value of 'op2' is true the timeout value for the request can be found at the top of the temporary stack (if it is constant Operand_int and Operand_var if not –with the adequate cast–). The value passed to the GETDATA (in the ATL language) as a message for the request is also at the temporary stack and its type is string (constant or not). Both operands are removed from the temporary.

An input data request message is created (*Request-Data*) with a number for the request (a global variable being incremented each time it is used), the name of the type, the message for the request and the timeout value, and sent to the Command Subsystem driver.

In the current version the timeout is allways sent with a -1 value.

After the request message is sent, the execution status is updated and an object to wait for the answer to this request (*WaitingInput*, with the number of the request and the execution status –see subsection *"Communication with the Command Subsystem driver"* in chapter 4), is created and is appended to the corresponding list.

WAIT ⟹ Instruction code = 57

This instruction causes the execution to stop in order to wait for the result of the external routine just called (generally this instruction is the last one after all parameter instructions for this call when the call is synchronous).
It updates the status of the execution and stores it into the object created to wait for this call (*Opened-Call*). Then it stops the current execution of the interpreter (execution of the present code table) and returns the control to the Virtual Machine driver.

PEXV op1 op2 op3 ⟹ Instruction code = 55

This instruction generates a parameter to be passed by value to an external routine call. The operands 'op1', 'op2' and 'op3' are respectively the type and the final type of the parameter and its offset in the temporary stack.
The first two operands, 'op1' and 'op2', are string constants (Operand_string) and the third one, 'op3', is an integer constant (Operand_int).
From the type and its definition a *Type* object is created, and with this *Type* a *Variable* object, generating also the tree for it in order to have the node pointer where the parameter value to be sent must be copied. If this value (obtained from the temporary stack) is constant it will be of a basic type, and depending on the type, it is cast into the correct operand type and assigned to the parameter *Variable*. If it is a node pointer, it is assigned using the 'node' copy operator (with the corresponding cast).
Once the *Variable* has been set with the parameter value, the instruction gets the name of the external routine call from the last external routine call sent and with this name, the number of the call and the *Variable* it generates a parameter message (*Parameter*) which is sent to the Command Subsystem driver.

CDEF op1 op2 ⟹ Instruction code = 58
This instruction executes a default function call of the ATL language. The addresses of these functions are into a structure with two tables which allows access to one of this functions just by knowing the corresponding indices in these two tables. The first operand 'op1' contains the index to the generic functions table, and the second one, 'op2', contains the index to the specific functions table (see subsection *"The execution"* in chapter 4).
The instruction executes the right function by calling to the generic function corresponding to the first index and passing to it the position in the specific functions table (given by the second index) and a reference to the temporary stack.

PEXR op1 op2 op3 ⟹ Instruction code = 56
This instruction generates a parameter to be passed "by reference" to an external routine call. The operands 'op1', 'op2' and 'op3' are the type, the final type and the offset to the value in the temporary stack.
The two first operands, 'op1' and 'op2', are string constants (Operand_string) and the third one, 'op3', is an integer constant (Operand_int).
From the type and its definition a *Type* object is created, and with this *Type* a *Variable* object which is initialized with the value of the node pointed to by the value of the Operand_var taken from the temporary stack at the offset position (the management of passing parameters "by reference" to external routines uses a *copy-in copy-out* paradigm). Then the instruction gets the name of the external routine call from the last external routine call sent and with this name, the number of the call and the just built *Variable* it generates a parameter message (*Parameter*) which is sent to the Command Subsystem driver. As this parameter must be returned by the external routine, a new *Variable* must be generated setting in it the node pointer to point to where the result parameter must be copied. This *Variable* is kept in the object created to keep the return of the external routine call (*OpenedCall*).

MASS ⟹ Instruction code = 59
This instruction is a "move" for an asynchronous call in an assignment sentence (see also section 4.3).
It takes the node pointer from the top of the temporary stack and puts it in the *Variable* reserved for it in the structure of the asynchronous call.

POPA ⟹ Instruction code = 60
Instruction "pop" for an asynchronous call. It is used when the call is internal and it decides that the return value of the call (if it exists) is not needed (see also section 4.3).

PRINT op1 ⟹ Instruction code = 200
This instruction writes into the output window of the Input Subsystem. It receives as operand 'op1' a string constant containing the format to be used to call the 'sprintf' C function. This format is limited to only one variable.
The value of the variable is found at the top of the temporary stack and it always is an *Operand_var*. The work to do is then to access to the value to be written, call to the 'sprintf' to build the string to be written and generate an external call to the routine "se::Sortida" (sending a *CallRoutine* message) generating also the parameters (*Parameter*) to be passed to this external routine (the string generated by 'sprintf' and another with a constant value "m"). This external call is exported by the Input Subsystem module and in this case it is called without waiting for its return, so it will be totally asynchronous.

END ⟹ Instruction code = 100
Instruction for ending an execution. It removes from the execution stack the parameters in case it was a function call execution and finishes the execution.
The content of the parameters has been deleted by the RMV instructions but END must pop them from the stack. It also destroys the current execution status (*ExecStep*) before ending. In case this *ExecStep* was for an internal asynchronous call this destruction requires a handshake with the calling routine explained in section 4.3.