UNIVERSITAT POLITÈCNICA
DE CATALUNYA

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

Marta Fairén

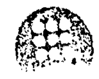
June, 2000

PhD Thesis
Supervised by Dr. Àlvar Vinacua

Software PhD program
LiSI Department,
UPC

T 00/158
1400325068

Biblioteca Rector Gabriel Ferraté
UNIVERSITAT POLITÈCNICA DE CATALUNYA


UNIVERSITAT POLITÈCNICA
DE CATALUNYA

Als meus pares,

DE CATALUNYA

Acknowledgements

First of all I what to thank Àlvar Vinacua for his undoubtedly excellent supervision. This thesis would not have been a reality without his ideas, inspiration and guidance. I give my gratitude to him for his constant dedication, his generous support and his patience as well during these years.

I would also like to thank those people who have been involved in the work of the thesis in some way. Thanks then to Ignacio Ruiz, for implementing the first version of the Input Subsystem and the utility processes "demandes" and "entrades"; to Daniel Sánchez-Crespo, for implementing the mechanisms involved in the multiplatform features; and more specially to Manuel Vivó, who not only implemented the automatic code generator but also has been an unconditional friend and encouraged me to feel more confident about myself.

I am also grateful to Eva Monclús, David Corbalán, Àlex Sánchez, Jordi Martín, Óscar Soriano and Óscar Sanjuan, who where the *beta-testers* for the initial versions of ATLAS. They helped me a lot in finding and fixing errors by developing big applications over it.

My gratitude to all members of the Computer Graphics Section (SIG) of the LSI department at the UPC, for providing me with more than adequate resources and a nice working environment. Thanks also to the computing laboratory (specially to José Luis Montero) for their help on the technical aspects of the equipment. I would also like to acknowledge support of a Spanish MEC grant for this work.

I give acknowledgements as well to the group of Computer Graphics of the University of Girona, to the Institute of Robotics and Industrial Informatics of the UPC, and to the Computer Graphics groups of the Universities of Granada and Zaragoza for trusting in ATLAS to be used as a platform for some of their projects.

Special mention in these acknowledgements is deserved by Isabel Navazo and Pere Brunet for their unconditional support, encouragement and sympathy.

Finally the biggest gratitude is to my family and more specially to my parents without whom none of this would have been even possible.

Contents

1 Introduction 1

1.1 Objectives 2

1.2 Other related work 3

2 Interactive Computer Graphics applications 7

2.1 On the specific problems of ICGA 7

2.1.1 Volatile references 7

2.1.2 Sharing Information 8

2.2 ATLAS' data labels 9

3 The ATLAS components 11

3.1 Centralized vs. symmetric architecture 11

3.2 The failure resistance (crash fault-tolerance) in ATLAS 13

3.3 A bird's eye view of ATLAS 13

4 The ATL language, its compilation and interpretation 15

4.1 Language description: Syntax and semantics 16

4.1.1 Modularity 17

4.1.2 Constants and basic types 19

4.1.3 Types 20

4.1.4 Variables 22

4.1.5 Expression evaluation. Operators. 24

4.1.6 Sentences 24

4.1.7 Procedures and functions in ATL 27

4.1.8 External functions and procedures 31

4.1.9 Comments 32

4.1.10 Other constructions 33
4.2 The ATLAS Command Subsystem 33
4.2.1 The `Command Subsystem` driver 34
4.2.2 The ATL compiler 34
4.2.3 The Virtual Machine 36
4.3 The *"dirty"* variables mechanism for the ATLAS asynchronous calls 50
4.3.1 General description 50
4.3.2 The asynchronous call as a sentence 52
4.3.3 The asynchronous call as the right side of an assignment . 53
4.3.4 On differences between external and internal asynchronous calls 54
4.3.5 Accessing to a *dirty* variable 60
4.3.6 Possible deadlocks dealing with "dirty" variables 60

5 The Input Subsystem **63**
5.1 Generic Input Subsystem 63
5.2 Design and implementation of the ATLAS Input Subsystem . . . 64
5.2.1 Basic input data 64
5.2.2 X-events control 65
5.2.3 Extended Tcl/Tk 67
5.3 Extension proposal 69

6 The `distr` process **71**
6.1 Processes management 71
6.1.1 Distribution in ATLAS 71
6.1.2 The *heartbeat* mechanism 88
6.1.3 Killing a process 88
6.1.4 Recovering a process 89
6.2 Communications management 89
6.3 Input data management 89
6.4 The *events* mechanism 92
6.5 Journaling management 93

7 Communications and drivers **95**

7.1 The ATLAS communications protocols 95

7.2 The set of messages used 104

7.3 Some implementation details 106

7.4 Transparent data transfer: `Variables` 107

7.4.1 Wrapper structure for data and types 107

7.4.2 Making this design transparent to the developer 110

7.5 Process driver 111

7.5.1 Automatic code generation 112

7.5.2 Handling of ATLAS events 122

7.5.3 Requesting input data 122

8 Journaling **125**

8.1 The journal and its functionalities 126

8.1.1 Design 126

8.1.2 Some implementation details 131

8.1.3 The journal functionalities 134

8.2 Meta-journal 142

8.2.1 Translation Meta-journal → journal 143

8.2.2 Consistency checking 144

8.2.3 The journal API offered by `distr` 146

8.2.4 The `MJEditor` 146

9 Using the system **149**

9.1 Design process of an ATLAS application 149

9.1.1 The process interface 149

9.1.2 The process implementation 149

9.2 The API library 151

9.2.1 Miscelaneous 159

9.3 Utility processes 160

9.3.1 Process "`pròcessos`" 160

9.3.2 Process "`demandes`" 161

9.3.3 Process "`entrades`" 164

9.4 A toy example 166

9.4.1 Description of the *esferes* application 166

9.4.2 Tiny user manual 166
9.4.3 Technical documentation and code 167
9.5 Applications using ATLAS 188
9.5.1 *VolAtlas:* a volume modeling application 188
9.5.2 *Octrees:* a solid modeling application using extended octrees 189
9.5.3 *NewDMI:* a BRep modeling application 190
9.5.4 *Motlles:* a CAD system for plastic injection moulds . . . 190
9.6 Evaluation of the system. The developers opinions 191

10 Extensions 195
10.1 Synchronous requests of input data from application processes . 195
10.2 Overloading in ATL 196
10.3 Extending the GETDATA command to accept timeouts for its requests 196
10.4 The Command Substitution possibility 197
10.5 Miscellaneous 198

11 Conclusions and future work 201

A ATL Grammar Description 209

B Intermediate code instructions 213

Chapter 1

Introduction

New technologies are introduced for the construction of computer programs at a fast pace. Usually they carry with them coveted performance or quality enhancements. Developers are thus interested in using these technologies and also in combining several of them to take profit of different aspects addressed by them. But the use of this technologies requires a fair amount of specific knowledge by both the designers and programmers of the application.

This is true for example, for aspects like application distribution in different processes running in different architectures, fault-tolerance with respect to the network, reusability or extensibility.

We are also interested in aspects more specifically related to computer graphics applications like the possibility of keeping a history of an execution (strongly needed in most CAD systems), the use of symbolic data (to define object families in parametric design) or finding a solution to the problem of graphical input data which normally are based on volatile references, and this makes their portability difficult.

The problem then consists in making some of these technologies available to an application developer without the burden of a steep learning curve. To do this, this thesis offers some tools and methods to greatly simplify the construction of fairly sophisticated applications. Therefore it deals with the design and implementation of a software support platform (ATLAS) to facilitate the development of advanced applications, specially for computer graphics, inasmuch as the aspects mentioned above are included in the design. The more concrete objectives of ATLAS will be explained in the next section.

This thesis is thus geared to offer to developers the possibility of using new techniques (like those named before) without requiring any special skill on them, i.e. as transparently as possible, and to find the solutions that best fit the conjunction of those techniques.

1.1 Objectives

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

In terms of functionality, ATLAS has these objectives:

- *Low level of parallelism.* ATLAS applications feature several distinct processes running concurrently in the same or different machines in a network; processes encapsulate ATLAS components or user modules. The user implements routines that are accessible to other processes as remote procedures. ATLAS is able to find out which processes are available on each machine and decide the best machine to execute a process in terms of capability and load.
- *Interprocess communication.* Since the application is split in several processes, these need to communicate over the network and exchange data between possibly different architectures. This gives rise to many different issues that need to be addressed [1].
- *Standardized input model.* A great deal of effort in an application's development is spent in its user interface. ATLAS provides a uniform but flexible view of inputs that allows many different dialogue modes in a uniform way, somewhat related to Abstract Data Views [2].
- *Configuration and macro language.* A flexible way must be provided for a programmer to describe what is in each of his modules, and how it should relate to others, and also to define the dialogues of the application and its behaviour in a simple way.
- *Flexible journaling mechanism.* This mechanism records the actions of the work session in order to be able to repeat this work session at any other time. This is needed in incremental design applications (in particular CAD-applications). The mechanism supports undo's and redo's and enforces the consistency of data recorded in the journal after editions or modifications of it.
- *Fault tolerance.* Since the application is spread out among several hosts, it becomes more exposed to transient or permanent failures (of the communications or of any of the hosts involved). Fault tolerance for these failures is transparently provided based on the journaling mechanism and on heartbeat messages sent by all processes so that their status can be assessed.
- *Reusability.* Each user module is completely isolated from others (in a separate process) except through a well defined interface described in ATLAS's programming language. Thus new components can incorporate and use reliably old ones. Another important issue for reusability, specially in computer graphics applications, is the use of a general data format

to describe complex scenes. Different processes require to use the same data format to read and write graphical information in order that other processes can also access these data. This issue is not directly covered by ATLAS, but is being solved with the MDTL project [3] also developed within our group.

- *Support to processes of constraints solving and parametric design.* ATLAS also includes a global identification mechanism which helps check the consistency of the journal, and solves the problem caused by volatile references in graphical input data and the use of symbolic data required by processes of constraints solving.

At this point, we have repeatedly used the word *flexible* to describe some aspects of ATLAS. With this we mean that these aspects are not rigid, i.e. they offer different ways to do something, or offer the possibility to change some behaviour of the system. In other words, since the ATLAS main objective is to offer the maximum transparency to the applications developer, ATLAS automatically manages a lot of applications aspects by using a predefined behaviour by default. This would have made the system too rigid to be used for a large number of applications, even in computer graphics. To avoid this rigidity, ATLAS adds *flexibility* so that applications can change this default behaviour if they want.

1.2 Other related work

There are several systems that can be considered related work since they address some problems that ATLAS also addresses. Distributed Object Computing environments (CORBA [4, 5, 6], DCOM [7] or Java RMI [8]) are some of these systems, which combine object-oriented concepts with client/server technology to produce a framework for building modular and scalable distributed applications at a relatively high level of abstraction.

Giving more emphasis to the fault-tolerance topic there are the Isis toolkit [9], Horus [10], Transis [11] or Totem [12] which combine process group with other facilities like the virtual synchrony communication model [13] in order to achieve fault-tolerance of distributed applications. In the same way but using a metaobject architecture is the FRIENDS system [14], which is also oriented to make these facilities as transparent as possible for the applications. The Arjuna object-oriented programming system [15] addresses the fault-tolerance problem as well by providing the programmer with classes that implement atomic transactions, object level recovery, concurrency control and persistence.

There are also other systems focusing their objectives on the network processes communications and the parallel capabilities (PVM [16], Linda [17], D-Memo [18]). Most of them simulate a virtual machine covering the intricacies of the network communications.

All of these systems are addressing some of the issues enumerated in section 1.1, but none of them is designed to achieve all of ATLAS's objectives. Moreover, as they concentrate only on the problems they are addressing, they

don't take into account the others. This causes that if you try to combine solutions to different problems from these different systems, the results are often intricate and inefficient because the solutions are not designed to work together. (As an example it would be quite complicated to mix CORBA communications with the process group solution to the fault-tolerance problem because they are using very different designs for their architectures). At any rate to afford these solutions, the programmer is required to master all the techniques involved.

ATLAS is an evolution of a system described in [19]. ATLAS inherits some aspects of the architecture of the previous one, and adds robust and fault-tolerant network distribution transparently, a meta-journaling system, a much more flexible control language and an orthogonal design which affords much more flexibility to the applications built with it.

Table 1.1 shows a comparison with those systems closest to the ATLAS requirements. The different columns are the most relevant issues related to the ATLAS objectives.

Transparency, indicates the system hides its intricacies to the developer. This column should have been evaluated with more values than yes/no, but it is almost impossible to say how transparent one system is, with respect to another. Because of this we decided to say yes in cases where the system is hiding totally the communications mechanism, the location of the object to reference and is also easy to use in case the developer must know he is using a distributed system.

All systems in the table favour the *Reusability* and implement *Interprocess communications*. Only one of them still does not support an *Heterogeneous network*.

The *Fault-tolerance* is only fully supported by Arjuna and FRIENDS. CORBA has specified it but no implementation supports it yet. In ATLAS it is also considered but only in terms of failures of machines or communications.

By *Persistent data* we mean the system offers persistence without any work required by the developer.

Journaling is an objective for ATLAS but not for any other system.

The support for *multiple language implementation* is only given by CORBA and DCOM. In ATLAS it is possible because it can have, like CORBA, different automatic code generators for the different languages, but this is not an ATLAS objective by itself so it will remain as an extension possibility.

Since Linda is a parallel language and D-Memo an extension of it, this issue and the next two, *IDL* and *Command language*, are not applicable for these two systems, so they are not evaluated for them.

Having an *IDL (Interface Definition Language)* to describe the interface of objects or processes is not necessarily a positive characteristic but just a characteristic. Some systems not offering an IDL extract the interface description from the implementation itself, which can be even easier.

The extension of the IDL to be also a *Command language* is also an ATLAS specification requirement so it is not offered by the others. This is also the same for the *GUI facilities* but in this case Java/RMI and DCOM have intrinsic

Comparative to the most relevant systems

	Transparency	Reusability	Interprocess communication	Heterogeneous network	Fault-tolerance	Persistent data	Journaling	Any language implementation	Interface Definition Language	Command language	GUI facilities
ATLAS	YES	YES	YES	YES	Of communi-cations	NO	YES	Possible (C++)	YES	YES	YES
CORBA	YES	YES	YES	YES	Not yet	NO	NO	YES	YES	NO	NO
JAVA/RMI	YES	YES	YES	YES	NO	NO	NO	NO (Java)	NO	NO	YES
ARJUNA	YES	YES	YES	YES	YES	YES	NO	NO (C++)	NO	NO	NO
DCOM	NO	YES	YES	Not yet	NO	NO	NO	YES	YES	NO	YES
Linda	NO	YES	YES	YES	NO	NO	NO	N/A	N/A	N/A	NO
D-Memo	YES	YES	YES	YES	NO	NO	NO	N/A	N/A	N/A	NO
FRIENDS	YES	YES	YES	YES	YES	YES	NO	NO (C++)	NO	NO	NO

Table 1.1: Comparison table with several systems

facilities for designing GUI, therefore they also offer it even though it is not one of their objectives.

This comparison shows that no other system covers all of the ATLAS' objectives; more specifically none of them addresses the *standardized input model*, the *configuration and macro language*, the *journaling mechanism* and the *support to processes of constraints solving and parametric design*. ATLAS has thus been designed to fill up this void in the manner most fitting for the development of interactive (computer graphics) applications in a research environment.

Chapter 2

Interactive Computer Graphics applications

2.1 On the specific problems of ICGA

We want our system to take special care with some particular irks of Computer Graphics applications, especially when they are interactive. Applications of this kind include of course all sorts of Computer Aided Design or Styling systems, but also applications used in Computer Aided Animation, Digital Effects, Medical Imaging, Virtual Reality, Enhanced Reality and Surgery Planning and Assistance. This list does not intend to be exhaustive. It does intend to make the point that although these applications may seem a narrow niche, they encompass a large variety of problems and software, making more sense to search for solutions addressing their specific problems.

These problems, from the point of view of ATLAS, are mainly two:

- Volatile references for data values.
- Sharing information between a constraint solving system and a user application

We shall next consider each of these two briefly, and then discuss the way in which ATLAS addresses these problems.

2.1.1 Volatile references

In a Computer Graphics application, not surprisingly, a great deal of user input comes in the form of geometric entities and coordinates. To make the system user friendly, systems universally adopt, to as large and extent as possible, a direct manipulation paradigm, whereby the user picks items with the mouse, then moves the mouse to express his intent. Several modifiers may be used to add expressiveness to this simple scheme.

ATLAS needs to deal with all the data representing these user actions for its journaling service to the application, and must be able to replay these "acts" in a meaningful way to a user module when attempting to recover from a network or machine failure.

A problem arises because the input data (essentially positions of the mouse together with buttons pressed and other modifiers) are given in device coordinates: they represent coordinates in pixels with respect to a standard corner of the window (typically the upper left-hand corner). Applications however deal with models that use their own coordinate systems, usually called world coordinates. For example in medical imaging the coordinates may be associated with real physical dimensions, or with the sampling frequency. In CAD applications they will almost always represent the real coordinates of the points of the part if it had been manufactured and placed in a standard position, in a set of units specified by the user. The relation between these world coordinates and the device coordinates is computed by the application when needed. This relation changes during the execution for diverse reasons. For example if we zoom in we are really changing the scaling from world coordinates into device coordinates (see [20], for example, for a more detailed discussion on this).

Since the conversion happens under control of the application, and may be changed by events out of ATLAS' control, simply playing back pixel coordinates may fail. For instance replaying the journal on a different workstation with different screen resolution may produce different window sizes, and different world-to-device transformations, resulting in different meaning for the same pixel coordinates! (see figure 2.1).

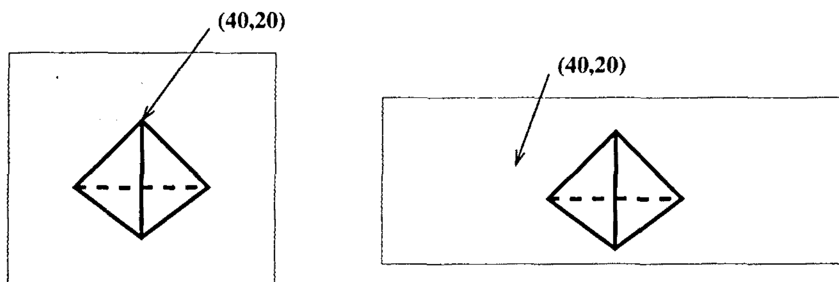


Figure 2.1: Example of a possible different meaning for the same input pixel if the window is resized.

2.1.2 Sharing Information

It is not really so much about sharing information but about sharing the meaning of information. Each user module in an ATLAS application will have its own data structures representing the models it handles. Asking it to export even part of the semantics of its data would require complicated procedures (in order to achieve some generality) and would require especial attention by the programmer, who would have to arrange for his code to export this information in the scheme adopted. It is likely that any such mechanism would also establish limitations to what can be expressed, and the least limitations would come at the cost of expensive parsing and analysis of this information.

Nonetheless, user modules need to be able to exchange information about their data in a way that is simple (i.e. places a small burden on the developer and on the application), yet it is able to support, for instance, the specification of a geometric constraint problem to a generic tool available in the system, in such a way that the tools answer (its solution to the constraint problem) can again be mapped into the module's own data in a simple and robust manner.

2.2 ATLAS' data labels

To address the issues just discussed, ATLAS uses a simple device. It features a ticket dispenser that can issue a unique system-wide identifier to anybody who asks for one. When user input is processed within ATLAS, each datum is attached a fresh ticket. This association is stored in the journal, and is also sent to the user process along with the datum.

User processes themselves may require from ATLAS as many tickets as they deem necessary through an API call (`atl_get_ticket` –explained in detail in chapter 9). To get the full benefit of the system, they will usually use plenty of them, and store them in association with relevant data that the user may interact with. A polyhedra modeler, for instance, might have one such ticket associated with each vertex, edge or face of each polyhedron in its database. Furthermore, user processes may instruct ATLAS about the "true meaning" of a user input through a call to `atl_substitute_ticket(t1,t2)`. This tells ATLAS that the datum that had ticket `t1` has been interpreted by the application to mean the entity to which it has attached ticket `t2`. In future instances where it is needed, as when recovering a process or replaying the journal, ATLAS will send at the point where it originally sent `t1`, a special input message with no data but the ticket `t2`, which the application will understand to mean the entity it has attached that ticket to. From the application point of view the handling of the special inputs can be easily done using overloading, providing one implementation for, say, a point in device coordinates, and another for a ticket.

Likewise, when posing a problem to an external geometric constraint solver, an ATLAS application needs only send the present coordinates/components of the entities involved, paired with their tickets (that it has previously obtained from the centralized server). The solver then returns its result in the form of instructions: "set item whose ticket is `t1` to these new coordinates...". This road has not been pursued further because solvers being developed by other groups in our lab have not been available until recently. We would expect to be able to integrate them into ATLAS in the form of generic services in the future.

These aspects of ATLAS have been considered in this thesis as a journaling functionality (*Global data identification*) discussed in section 8.1.3, where a somewhat lengthier and detailed discussion, along with some examples, may be found.

Chapter 3

The ATLAS components

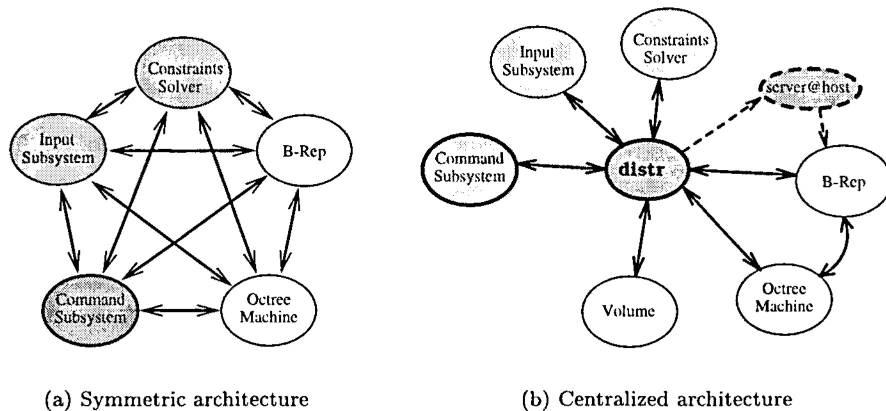
ATLAS is a multi-process platform. It can be seen as a collection of processes, running in several machines in a (local area) network. As such, its own components may be (and actually are) spread out in several distinct processes. Several decisions had to be made early on, therefore, as to the way in which ATLAS processes would interoperate and the way in which the ATLAS kernel's responsibilities would be spread among different processes. Some of these issues might have required some experimentation to assess exactly their virtues and downsides, but the cost of the project as a whole had to be kept in mind too: we wanted to be able to construct a solid system, or at least a reasonably mature prototype within the scope of this work. Therefore, and because that was not the focus of our work, we admittedly have made decisions in these areas without complete or exhaustive experimentation, but based on reasonable assumptions and considerations. In this brief chapter we intend to present some of these design issues and decisions, together with a first, bird's eye, view of the system.

3.1 Centralized vs. symmetric architecture

At a very first step, the alternative presented in figure 3.1 had to be resolved. We could conceive of ATLAS as an egalitarian system, with all processes having an equal standing and knowing about the whole network of processes (figure 3.1(a)), or we could choose to have some central hub managing all communications between processes (figure 3.1(b)). Both options had pluses and minuses, and there were of course other mid-way solutions that one could imagine.

The symmetric architecture (figure 3.1(a)), where all processes have identical standing with respect to the system, has the attractive of its orthogonality, and allows for very resilient fault-tolerance mechanisms to be devised: since all processes share some form of vision of the whole application, any one can take it upon itself to trigger the recovery from failure of a part of the system that it detects is not working.

However, the complications emanating from this idea are also severe. The problem of consensus among the processes is an example. If the view of the state



(a) Symmetric architecture

(b) Centralized architecture

Figure 3.1: Two different alternatives for the ATLAS architecture

of the system is not reliably shared by the component processes, a very complex set of cases arises, and system behaviour may soon be all but predictable. This problem has been studied on its own in the literature (for a good survey see [21]). At any rate, any conceivable solution of it would load each ATLAS process with heavy machinery just to maintain the state of consensus. Other difficulties would include establishing who manages each of ATLAS' services, and how much information related to each service has to be placed in a common pool (probably meaning replicated for all processes in the application) in order to achieve our goals.

Preliminary study of all these issues made us quickly bend towards the other extreme: the architecture depicted on the other side of the figure 3.1 (3.1(b)). Here a central process acts as a communications hub, maintaining open channels of communication with each and every ATLAS process in the application. This central process, dubbed `distr`, should be kept as simple as possible, and extra care should be placed on its development, in order to make it robust. It will be the place were all information relevant to the application recovery in case of failure resides, and therefore failures of `distr` itself cannot be recovered. This relaxation of the failure-resistance objective seems reasonable given the enormous impact in the system's weight that it has: many subsystems become a lot simpler to devise once this view is adopted. Moreover, being a standard component on which one can spend extra efforts to ensure its stability, the impact on system's recovery should be small.

In fact ATLAS failure resistance is mainly focused on failures of the communications or the systems in which the processes are being run. When a user wants to start an ATLAS session, he really starts `distr` and it starts up the rest of the system; typically `distr` is run in the user's workstation, and therefore failures of the kind mentioned above make it pointless to recover the session anyway.

3.2 The failure resistance (crash fault-tolerance) in ATLAS

Once decided for the centralized architecture, the decisions and managements of ATLAS fault-tolerance lay on the `distr` central process. The fault-tolerance problem has been studied in the literature at length. A recent survey can be seen in [22] which summarizes the necessity of two important steps for a system being fault-tolerant: the *"detection"* of faults and the *"correction"* of them when detected.

In the ATLAS fault-tolerance, which is only for crashes of processes or communications, the *detection* (or maybe better called *suspicion*) is performed by a *heartbeat* mechanism (explained in chapter 6). This mechanism allows `distr` to detect (or suspect) when a process or its communication with `distr` crashed. The *correction* relies on the information kept in the journal and is performed by the *processes recovery* which is an ATLAS journaling functionality (explained in chapter 8).

From now on, these two parts of the ATLAS fault-tolerance will be treated independently in the thesis. So this concept of fault-tolerance containing both parts is not going to be discussed again.

3.3 A bird's eye view of ATLAS

Because we have decided for a centralized architecture (figure 3.1(b)), the central process `distr` is the most crucial component in the architecture. It acts as a communications center for the duration of the application execution and is charged with starting the system, relying messages, keeping tabs on the status of all processes in the application at a given time, requesting the addition of new processes or mandating the finalization of running processes, keeping the journal and managing re-executions when needed.

Each process connected in an application needs a communications driver to be able to manage the interchange of messages with the rest of the application. The execution of processes relies on a paradigm similar to the *remote procedure call* paradigm, therefore a process is considered as a set of routines to do the process related work plus a communications driver. Taking information of the processes interface defined in the ATL control language (see below), the ATLAS automatic code generator is able to generate the necessary code to implement the communications driver for the process. Therefore, it gives the desired transparency to the developer who does not have to know about the communications mechanism for his application processes.

A direct communication between two processes is also considered in the architecture (as shown in figure 3.1(b) between the `B-rep` process and the `Octree Machine` process). It allows the interchange of big volumes of data between processes. About this special communication, `distr` will only know the times of its creation and destruction, in order to be aware of the possible "domino effect" in case one of the processes taking part in this communication crashes. When a

process or its communication with `distr` crashes, the processes recovery management can determine whether this process at the moment of the crash had an opened direct communication or not. In the affirmative case the other process involved in the direct communication has to be also recovered in order to reach again the right status of both processes. However, this kind of communications has not been included yet in the present prototype, so its management is not explained in detail in this thesis except for the description of its protocols with `distr` (chapter 7).

The ATL language is a modular language which is used by the developer for different purposes: to configure the application; to describe the interface of a process; to define the interaction among processes and the dialogues of the application; and also to facilitate the developer a rapid improvement and debugging of his ATL code because it can be modified and recompiled at run time. It can be used also as a powerful macro language by the final user to control the execution or customize the environment.

The interpreter of the ATL language is the `Command Subsystem`, and it allows interactivity and is able to manage both synchronous and asynchronous routine calls, offering to the user a certain level of parallelism.

Another important aspect of ATLAS is the separation between computing processes and data input processes. Although it is not compulsory –some modules may incorporate their own interface if needed–, it favours the reusability and allows the developer to build his application without taking care of the user interface because he can reuse another one available.

To this end, `distr` is also in charge of matching the input data coming from a user interface process with the corresponding data requests coming from the computing processes when they need input data. This matching is done asynchronously by `distr`, who also registers this match in the journal in order to be able to reproduce it if needed.

The last fundamental ATLAS component is the generic `Input Subsystem`. It defines a general user interface and may be charged with managing interaction with other windows created by the application (both X-windows and Tcl/Tk [23] interface windows). It is implemented as a user process simplifying its replacement by a new one if required.

Other utilities are also included, but are more anecdotical (i.e. not strictly necessary to run an application). However, in some cases they can be used to allow the advanced developers to overrule some defaults or use ATLAS at a lower level to gain control of how things are handled. They also show that the handles offered to interact with `distr` allow for enough extensibility.

The following chapters explain in detail all of these ATLAS components and also the connections existing among them to build the whole system.

Chapter 4

The ATL language, its compilation and interpretation

The controlling device that orchestrates the collaboration between all the application processes is the `Command Subsystem` which is an interpreter for the ATL language.

ATL is an imperative, modular language used by the developer for several different purposes: to configure the application –stating which processes belong to it; to describe the interface of a process indicating the public routines that can be called by other modules; to define the interaction among processes and the dialogues of the application; and also to facilitate the developer a rapid improvement and debugging of his ATL code because at run time the developer can modify and recompile modules and the changes become effective immediately.

As the ATL language is used to define the process interface, it is similar in some sense to the Interface Definition Language (IDL) in the OMG CORBA specification. But ATL is also a programming language that allows the definition of commands to configure the interaction among processes or make some easy calculations with data to be used as a parameter, for example.

To the final user the ATL language is also useful because he can use it as a powerful macro language to better control the execution, or customize the environment.

Although this role could have been relayed to an existing scripting language (Guile [24], Perl [25], Python [26], Tcl [23], ...) it was deemed not appropriate as we wanted a great deal of control on the semantics of user scripts. Furthermore the close intertwining of this scripting engine with other components of ATLAS made it unclear to which extent the use of an external scripting language would save work. It is true however that using a more standard scripting engine would have saved (some) users the need of learning yet one more. For this reason, in our implementation we have used Tcl/Tk to add widgets to the user interface (see Chapter 5: *The* ATLAS *Input Subsystem*).

4.1 Language description: Syntax and semantics

As an example, figure 4.1 shows a portion of an ATL module. This example is included here as a first illustration of the flavour of ATLAS, but does not include very aspects of the language, that we shall now discuss in detail.

```
USE se, demandes;
// Type definitions
#deftype ::esferes_Button_pressed_event se::Button_pressed_event;
EXPORT #deftype Point STRUCT
                        px -> real; py -> real; pz -> real;
                      ENDSTRUCT;
EXPORT #deftype ColorRGB STRUCT
                           r -> real; g -> real; b -> real;
                         ENDSTRUCT;
EXPORT #deftype Sphere STRUCT
                         center -> Point; rad -> real; color -> ColorRGB;
                       ENDSTRUCT;
PROT
  EXTERN FUNCTION GetWindowId () RETURNS integer;
  EXTERN PROCEDURE AddSphere (Sphere sph);
  EXTERN FUNCTION GetColor (::esferes_Button_pressed_event ev) RETURNS ColorRGB;
  EXTERN PROCEDURE RemoveSphere (::esferes_Button_pressed_event ev);
  EXTERN FUNCTION GetSphere (::esferes_Button_pressed_event ev) RETURNS Sphere;
  EXTERN PROCEDURE ChangeColorSphere (::esferes_Button_pressed_event ev, ColorRGB c);
ENDPROT
   ...

EXPORT PROCEDURE AddDefault () IS
  Sphere sph;
  sph.center.px = 0; sph.center.py = 0; sph.center.pz = 0;
  sph.rad = 1;
  sph.color.r = .5; sph.color.g = .5; sph.color.b = 1;
  AddSphere (sph);
ENDPROCEDURE

EXPORT PROCEDURE Remove () IS
  RemoveSphere (GETDATA("Select the sphere"));
ENDPROCEDURE

EXPORT PROCEDURE ChangeToColor (real r, real g, real b) IS
  ColorRGB col;
  col.r = r; col.g = g; col.b = b;
  ChangeColorSphere (GETDATA ("Select the sphere to change color"), col);
ENDPROCEDURE

EXPORT PROCEDURE Distance () IS
  Sphere sph1, sph2;
  esferes_Button_pressed_event bp1, bp2;
  sph1 = GetSphere (GETDATA ("Select the first sphere"));
  sph2 = GetSphere (GETDATA ("Select the second sphere"));
  real dcent, dsurf;
  dcent = sqrt(fabs((sph1.center.px-sph2.center.px)*(sph1.center.px-sph2.center.px)
                   +(sph1.center.py-sph2.center.py)*(sph1.center.py-sph2.center.py)
                   +(sph1.center.pz-sph2.center.pz)*(sph1.center.pz-sph2.center.pz)));
  dsurf = dcent - sph1.rad - sph2.rad;
  PRINT ("Distance between centers = %f", dcent);
  PRINT ("Distance between surfaces = %f", dsurf);
ENDPROCEDURE
   ...
```

Figure 4.1: Portion of the "esferes.atl" file, the ATL module of the toy application explained in chapter 9.

The grammar of the ATL language is described in appendix A.

4.1.1 Modularity

The ATL command language defined in ATLAS is a modular language where a module is a file written in this language. The name of this file must have an extension **.atl** and the file name without this extension is considered the module name. Each module can be compiled individually if it does not depend on any other module. The sentence USE is used in the language to control the inter-modules dependences. This sentence used in a module tells the compiler this module will use exported areas from the modules indicated in the sentence.

Syntax: "USE" module_name ("," module_name)* ";"

This sentence causes the automatic compilation of the indicated modules if these modules have not been compiled yet. Every compilation of a module provokes the automatic execution of the procedure *"main"* at the end of the compilation if this procedure has been defined in the module (see subsection 4.1.7).

The module name (the name of the file containing it) must be a correct identifier for the language itself. In case the module includes the interface declaration of a process of the application, this name must be also the name of the process. The range of identifiers which are accepted by the language is represented by the following regular expression:

[a-zA-Z][a-zA-Z0-9_]*

Every entity (type, variable, function or procedure) defined inside a module belongs by default to the module scope (it is only visible inside the module). But inside a module, it is also possible to define entities belonging to a global scope (with a global name), or exported by the module (visible also on the global scope). The name of the entities exported by the module will be prefixed with the module name and two colons (as in C++ classes). As both of them (global and exported entities) are visible on the global scope, any other module can access to them.

In order to make a global definition the syntax to be used is to prefix the name with "::", indicating to the compiler that this definition is to be included in the global scope and visible for everybody. A module can access to a global entity either by using its simple name —if there isn't any other more internal definition with the same name— or using the name prefixed by "::" as in the definition.

Using global definitions a lot is not recommended, because it can provoke many cases of collision among different modules because of the use of equal names.

Those entities belonging to the module (defined into it and not global) are introduced into the symbol table with their name prefixed by the module name followed by "::". To make them public, their definition must be preceded by the keyword "EXPORT". This tells the compiler they must be visible in the global scope. Therefore any other module can access them using *"module_name::name"*.

Scope management: There are three different scope levels, the global scope, the module scope and local scopes. In the global scope there are those

entities defined explicitly as global entities and those exported by the module. In the module scope there are those entities defined in the module but outside any block (function or procedure). Those entities defined inside functions or procedures describe local scopes (one for each block).

When an access is done inside a local scope, the compiler looks for it in the symbol table and if there is more than one entity with the same name it uses the one belonging to the most internal scope visible from there.

Example: file "module1.atl".

definitions:

#deftype type1 integer	⟶	Defines the type *'module1::type1'* in the module scope.
#deftype ::type2 **VECTOR**[3] **OF** real	⟶	Defines the type *'type2'* in the global scope.
PROCEDURE proc1 () **IS**	⟶	Defines the *'module1::proc1'* function in the module scope.
{ sentences }	⟶	Everything defined inside a block belongs to local scope.
ENDPROCEDURE		
type2 var1;	⟶	Defines the *'module1::var1'* variable with type *'type2'* (global) in the module scope.
EXPORT FUNCTION fnc1 () **RETURNS** integer **IS**	⟶	Defines the *'module1::fnc1'* function in the global scope.
{ sentences }		
ENDFUNCTION		
EXPORT #deftype type3 string	⟶	Defines the *'module1::type3'* type in the global scope.

access:

PROCEDURE examp () **IS**		
var1 = ... ;	⟶	Assigns a value to the module variable *'module1::var1'*.
var3 = ... ;	⟶	Assigns a value to the global variable *'var3'*. It must be global because it has not been defined in the module.
::type2 var4;	⟶	Access to the global type *'type2'* to define the local variable *'module1::var4'*.
moduln::proc3 ();	⟶	Attempt to access to a procedure *'proc3'* which must be exported by the module *'moduln'* and which is in the global scope with the name *'moduln::proc3'*.
ENDPROCEDURE		

Analogous to the "USE" sentence, there are also an "UNUSE" sentence and a "REUSE" sentence (not allowed to be used inside procedures or functions).

Syntax: "UNUSE" module_name ("," module_name)* ";"
"REUSE" module_name ("," module_name)* ";"

The "UNUSE" sentence is useful to undo the compilation done by the "USE" sentence, therefore its effect is to eliminate every symbol of the symbol table that had been added by the corresponding module compilation (symbols defined or declared in that module).

The "REUSE" sentence is useful to recompile modules, because it allows to redefine all the internal definitions on the module. Its effect is to eliminate every symbol of the symbol table except those being prototypes of external routines (see section 4.1.8) and after the elimination it recompiles the ATL module. This allows the developer to recompile modules interactively.

As the "UNUSE" and "REUSE" sentences are causing the undefinition of the symbols defined in this module, they must be carefully used because other modules can have references to these symbols in their commands that can provoke an error on the application execution.

The sentences "USE" and "UNUSE" are also ordering an automatic execution or killing of the process whose module is being used (in case the module is representing an external process). Therefore it must be noticed that an "UNUSE" of a module which is representing a process will cause the ending of that process. It doesn't happen with the "REUSE" sentence, but on the other hand if the "REUSE" sentence has an error and doesn't finish correctly the module will not be recompiled and it can cause execution errors if the other modules are using some entities declared in this module.

The "REUSE" sentence will finish with error if the prototype of the external routines (in case it is a module representing a process) have been changed. This is because since the process was being executed before the "REUSE" sentence, it has the external routine prototypes as defined by the first "USE" sentence. Any change on these prototypes cannot thus be accepted by the compiler.

4.1.2 Constants and basic types

There are four basic types recognized by the language:

integer → It has an integer value and its internal representation is a long in C++. The syntax accepted is a sequence of digits [0..9]. All the arithmetic and comparison operators are valid for integers.

real → It has a real value and its internal representation is a float in C++. The syntax accepted for a constant of this type is *digits.digits* where *digits* is a sequence of digits [0..9] which can be empty for one of the two cases. The same as with integers, all the arithmetic and comparison operands are valid for reals.

boolean → It has a boolean value (true or false) and its internal representation is a bool in C++. It only accepts as constant values *TRUE, true, FALSE* and *false*. The operators accepting this type in their operands are the logic operators and the equal and not equal relation operators ('==' and '! =').

string → It has a string of characters as its value and its internal representation is a String of the gnu library of C++. Any set of characters delimited by quotes is accepted and the quotes itself are also accepted if they are preceded by the backslash escape. In that case the compiler internally takes out the backslash escape and builds the C++ String without it. The accepted operations with strings are the equal and not equal comparison operators and the add operation ('+') which concatenates the two strings.

4.1.3 Types

Apart from the basic types just described, the ATL language also accepts programmer defined types built from other types using the struct, vector or alias type constructors.

In ATL a "struct" is defined by a set of fields (of equal or different types) and a "vector" by an array of certain fixed number of elements of the same type.

There is also the possibility of defining types as aliases of *_atl_unknown*. These types can be used in the module to define variables which are needed only to pass information from a routine to another, and the internals of these types do not need to be known inside the module. These types won't be accessible by the compiler nor the Virtual Machine, so they can only be used as parameters or as return values. The *_atl_unknown* types cannot be used as part of another structure or vector type.

The required syntax for the internal description of types is:

```
type  <--   "STRUCT"
                ( field_identifier "->" type ";" )+
            "ENDSTRUCT"
           | "VECTOR" ( "[" number_elems "]" )+ "OF" type
           | predefined_type
```

where "predefined_type" is the name of a type previously defined (already in the symbol table) or a basic type.

And the syntax to define a new type is:

"#deftype" type_identifier (type | "_atl_unknown")

A type definition can be done at any point of the command language code except inside blocks (functions and procedures) and it will be considered defined from that point on.

As an example, a well defined type could be:

```
#deftype new_type  STRUCT
                     field1 -> integer;
                     field2 -> VECTOR[3][5] OF  STRUCT
                                                  f1 -> real;
                                                  f2 -> string;
                                                ENDSTRUCT;
                     field3 -> boolean;
                   ENDSTRUCT
```

The type checking used by the ATL compiler is a type checking by name, i.e. two types are equal only if they have the same name. Therefore, when type checking must be done, the types involved must be defined and have a name. There is also the possibility of defining variables of a type "without name". These are variable definitions where the type is not a name but directly a type definition. In this situation the compiler generates a new name for this type but this name is unique and only identifies the type of the variables declared on the same declaration line.

Since the type checking by name is very strict, ATL also accepts type *castings*. Some of these castings (implicit) will be done automatically by the compiler and the others (explicit) must be explicitly ordered by the programmer.

First of all, we should define what we will call the *"effective type"* of a type. The *effective type* represents the same structure as the type but combining only primitive types (basic types). When the type has a component which is an alias of another type, the *effective type* substitutes this component by the *effective type* of the component, so, finally it is just the same structure with basic types composing it. As an example, if we have the type "usertype" defined as:

```
#deftype myint     integer
#deftype point     VECTOR[3] OF real;
#deftype usertype  STRUCT
                     field1 -> myint;
                     field2 -> point;
                     field3 -> boolean;
                   ENDSTRUCT
```

the *effective type* for this "usertype" would have the following structure:

```
STRUCT
  field1 -> integer;
  field2 -> VECTOR[3] OF real;
  field3 -> boolean;
ENDSTRUCT
```

- **Implicit castings:**

  Type conversions that the compiler can manage automatically when there are different types in an operation.

  The supported implicit castings in ATL are:

  - casting from "integer" to "real"
  - casting from any type whose *effective type* is a basic type to that basic type

Examples:

Given the following definitions:

```
#deftype typ1 integer
#deftype typ2 integer
#deftype typ3 tip1
integer enter;
typ1 t1;
typ2 t2;
typ3 t3;
real rr;
```

we will have:

```
enter = t1 + 3;          OK!
enter = t2 + t1 * t3;    OK!
rr = t2 + enter;         OK!
t2 = t1;                 ERROR!
```

where the last case is an error because t2 is not of a basic type, therefore typ1 cannot be implicitly converted to typ2.

- **Explicit castings:**

  Type conversion asked explicitly by the programmer. Its syntax consists on putting, just before the expression whose type we want to convert, the name of the type to which it must be converted enclosed in brackets:

  "{" type_name "}" expression

  This syntax asks the compiler to convert the expression type to the type "type_name". The compiler then has to check if this explicit conversion is possible and this is made by testing if the internal structures of both types are equal or implicitly convertible (for cases covered in the previous bullet).

4.1.4 Variables

The ATL command language allows variable declarations of any type described above. In fact, we can say that a variable is an instantiation of the type definition tree that defines it whose leafs can contain values.

A variable declaration can be done at any point of the code, but there will be differences depending on the position of the declaration

- ***between blocks*** → it is out of any block (procedure or function). In this case the variable will belong to the module scope (in case the "EXPORT" or "::" tokens are not used), or to the global scope otherwise (see subsection 4.1.1).

  A variable declared in global scope will be visible from its declaration to the end of the execution. A variable declared in module scope will be

visible also from its declaration but only to the end of the module where it has been declared.

Syntax: { "EXPORT" }
type { "::" } var_name ("," { "::" } var_name)* ";"

where 'type' can be either an already defined type name or a type description "without name" (see subsection 4.1.3).

- ***inside a block*** → it is declared inside a block (procedure or function). In this case the variable will belong to the local scope corresponding to the block and will be visible from its declaration and only to the end of the block definition.

Syntax: type var_name ("," var_name)* ";"

The syntax used to refer to a field of a struct or an element of a vector is:

variable_name ("." field_name | "[" integer_expression "]")*

where 'field_name' is the name of the desired struct field and the 'integer_expression' can be any expression evaluating to an integer.

Examples:

We can do things like:

```
#deftype type1  STRUCT
                  field1 -> integer;
                  field2 -> VECTOR[3][5] OF  STRUCT
                                               f1 -> real;
                                               f2 -> string;
                                               f3 -> integer;
                                             ENDSTRUCT;
                  field3 -> boolean;
                ENDSTRUCT

type1 var1;

PROCEDURE example () IS
  var1.field1 = 34;
  integer index;
  index = (var1.field1-10)%5;
  var1.field2[var1.field1%3][index].f2 = "Myname";
  var1.field3 = var1.field1/2 == var1.field2[0][0].f3;
ENDPROCEDURE
```

4.1.5 Expression evaluation. Operators.

An ATL expression can be a constant (of any basic type), a variable, a function call or any combination of them allowed by the operators which are combining them. Every expression has an associated type that is the expression evaluation result type. This type is known at compilation time. Therefore the type checking needed to know if an expression is correct or not can be done also at compilation time.

The following table shows, with a decreasing priority order, all the operators that can be used in an expression and the types allowed for their operands and result.

<table>
<tr><th>Operator</th><th>opnds</th><th>Operand types</th><th>Return types</th></tr>
<tr><td>'-'</td><td>1</td><td>integer,real</td><td>integer,real</td></tr>
<tr><td>'!'</td><td>1</td><td>boolean</td><td>boolean</td></tr>
<tr><td>'*'/'/'</td><td>2</td><td>integer,real</td><td>integer,real</td></tr>
<tr><td>'%'</td><td>2</td><td>integer</td><td>integer</td></tr>
<tr><td>'+'</td><td>2</td><td>integer,real,string</td><td>integer,real,string</td></tr>
<tr><td>'-'</td><td>2</td><td>integer,real</td><td>integer, real</td></tr>
<tr><td>'=='/'!='</td><td>2</td><td>integer,real,string,boolean</td><td>boolean</td></tr>
<tr><td>'<'/'<='/'>'/'>='</td><td>2</td><td>integer,real</td><td>boolean</td></tr>
<tr><td>'&&'/'||'</td><td>2</td><td>boolean</td><td>boolean</td></tr>
</table>

Because of the fact that an implicit casting from integer to real is possible, these two types can be operated together giving a real as a result type (see subsection 4.1.3).

4.1.6 Sentences

A sentence is an action without any return result. The different sentences recognized by the language are:

- the *empty sentence*, with syntax ";", without any side effect (equivalent to skip);
- an *assignment sentence* which allows to assign the result of evaluating an expression to a variable of a compatible type;

  Syntax: variable "=" expression ";"

  where 'variable' can be the whole variable or the definition of an access to one of its parts (field or element);
- a *conditional sentence* which evaluates a boolean expression and depending on whether it is true or false it executes one of two sets of sentences;

  Syntax: "IF" expression "THEN" sentences
{ "ELSE" sentences }
"ENDIF"

where the type of the expression result must be boolean and the branch of sentences ELSE is optional. The execution effect of this sentence is to execute the branch THEN when the expression result is true and the branch ELSE (if it has been defined) or nothing (if not) when the expression result is false;

- a *multiple conditional sentence* which evaluates an expression to decide which branch to execute and compares the expression result with the constant that defines the branch. It executes the first branch that complies with the condition;

  ```
  Syntax:   "CASE" expression "IS"
                  ( "WHEN" constant "DO" sentences )*
                  { "OTHERWISE" sentences }
            "ENDCASE"
  ```

  It can contain any number of branches WHEN but only one branch OTHERWISE which is optional. The program will execute the sentences of the OTHERWISE branch only if no other constant of a WHEN branch is equal to the expression result. If none of the WHEN branches complies with the condition and there is no OTHERWISE branch this sentence won't execute any of its branches;

- two *iteration sentences* which evaluate a condition (boolean expression) and execute the sentences block repeatedly until the condition becomes false;

  a) WHILE sentence:

  ```
  Syntax:   "WHILE" expression "DO"
                  sentences
            "ENDWHILE"
  ```

  For each iteration the expression is evaluated (it must be a boolean expression) and if the evaluation result is true the sentences are executed and the control goes back to the expression evaluation to repeat the process. When the evaluation result is false the sentences are not executed and the iteration finishes.

  b) FOR sentence:

  ```
  Syntax:   "FOR" "(" assignment ";" expression ";" assignment ")"
                  "DO" sentences
            "ENDFOR"
  ```

  First of all, the first assignment sentence is executed and after that the iteration starts evaluating first the expression (boolean expression) and executing the sentences if the result is true. In this case, before going back to the expression evaluation (the following iteration), the second assignment sentence is executed. The iteration process ends when the expression result is false.

- an *output sentence* which allows to print data in an input subsystem specially created output window. This sentence accepts a string as a first parameter which has a format similar to the C *"printf"* sentence but allowing only one other parameter, i.e. only one value is accepted into the specified format.

  Syntax: "PRINT" "(" STRING "," variable ")"

  where STRING is a string constant that only permits one character '%' inside it which indicates the exact point where the second parameter variable value must be put.

- a *grouping sentence* which given a module name not corresponding to the module being compiled, causes local names to be resolved in the scope of that module if they are not found in the local scope.

  ```
  Syntax:   "WITH" module_name "DO"
                     sentences
            "ENDWITH"
  ```

  where 'module_name' will be the module name to prefix the unknown call sentences in this block.

  In the following example, the effect produced in a function or procedure call which is done inside a grouping sentence is shown:

  Example: file "mod.atl"

  ```
  FUNCTION func1 (real i, real j)
    RETURNS real IS
            ⋮
  ENDFUNCTION
  PROCEDURE example () IS
    real var1;
            ⋮
    WITH process_ext DO

      var1 = func1 (i,j);          →  Causes the function call:
                                      'mod::func1(i,j)'.
      proc2 ();                    →  Causes the procedure call:
                                      'process_ext::proc2()' because
                                      there isn't a 'mod::proc2()'.
      func2 (var1);                →  Causes the function call:
                                      'process_ext::func2 (var1)'.
      var1 = ::global_func ();     →  This is not modified because
                                      it is an explicitly global func-
                                      tion call. It causes the func-
                                      tion call: 'global_func ()' in
                                      the global scope.

    ENDWITH
  ENDPROCEDURE
  ```

A nested use of this grouping sentence is also accepted. In this case the compiler uses a stack where it pushes and pops the module names depending on the grouping sentence arriving. The bottom of this stack is always the module being compiled. Names are prefixed with module names from this stack from the bottom up until they are found in the symbol table.

There are also other sentences like the *procedure or function call sentence* and the *return sentence* that will be presented in next subsection.

4.1.7 Procedures and functions in ATL

Definition:

Functions and procedures are the execution blocks of the module. Their definition is different for a function or a procedure.

A procedure is defined with a name, a set of parameters and a block of sentences:

Syntax: { "EXPORT" } { "ASYNC" }
"PROCEDURE" { "::" } nom_proc "(" param_list ")" "IS"
sentences
"ENDPROCEDURE"

where 'param_list' is the set of parameters of the procedure which is composed by a list of parameter definitions separated by commas. Each parameter definition has a type name (that must be previously defined) and a given name for the parameter. An example of a procedure definition can be:

PROCEDURE proc (integer x, typel parl, boolean r) **IS**
{ sentences }
ENDPROCEDURE

The optional tokens "EXPORT" and "::" are defining the scope where the procedure will be defined (see scope definition details in subsection 4.1.1).

The "ASYNC" token is also optional and indicates that this procedure can be called asynchronously (without stopping the execution of the calling routine).

To define a function, in addition to the name, the set of parameters and the sentences block, a type determining the return value type of this function is also required:

Syntax: { "EXPORT" } { "ASYNC" }
"FUNCTION" { "::" } func_name "(" param_list ")"
"RETURNS" type_name "IS"
sentences
"ENDFUNCTION"

where 'param_list' is the same as in procedures and 'type_name' is the required type for the function return value. As an example we can define:

FUNCTION func () RETURNS boolean IS

{ sentences }

ENDFUNCTION

Call and return sentences:

The *call sentence* will have the same syntax for both functions and procedures. The effect produced by this sentence is also the same for both cases because even though a function is returning a result value when this function is called without getting the result value it is treated as a procedure call and the return value is lost. The return value of a function will be considered when this function call is taking part in an expression.

Syntax: { "ASYNC" } routine_name "(" expr_list ")" ";"

where 'expr_list' are the expressions used to pass the actual parameters to the call, and the "ASYNC" token is used to make an asynchronous call (which requires that the routine was declared with the possibility of being asynchronous with this token). Of course, a routine defined synchronous cannot be called as asynchronous, but the reverse is permitted, so the user can call a routine defined asynchronous to be executed synchronously (by not using the ASYNC token when calling the routine).

An asynchronous call is only accepted as a sentence itself (without taking care of the result if it has one) or as the r-value of an assignment.

A function or procedure call is also accepted at the blocks level (out of any procedure or function body). Its effect is the immediate execution of this call unless it has not been totally defined yet, which will cause an error.

The *return sentence* is also different depending on where it is used (function or procedure). In a procedure the return sentence is only an order to finish the procedure execution at this point, without any value to return. In a function this return sentence is also an order to finish the function execution but it must have also a return expression which evaluates to the value the function must return. The definition of this return sentence is:

```
return  <--  "RETURN"
           | "RETURN" expression
```

In a procedure the return sentence is optional and if it doesn't exist the procedure execution ends at the end of the sentences block. On the contrary, a function must have at least one return sentence in order to indicate the function return value. It is an error for a function to terminate without executing a return sentence.

Parameters:

With respect to the parameters for these functions and procedures, the language accepts these parameters to be passed by reference or by value. The difference is given in the definition and (as in C++) if the parameter is passed by reference its name is preceded by the character '&'. The parameters definition is then:

param_list ← { param ("," param)* } :→ the parameters list can be empty

param ← type_name { "&" } par_name :→ the '&' character is optional

When a parameter is passed by value the sentences block works with a copy of the variable passed as a parameter, but when a parameter is passed by reference the sentences block works with a reference to the variable passed as a parameter, therefore any change made to this variable inside the function or procedure will be effective also outside the function or procedure block.

When the compiler is looking at a sentence call, it must make type checking for each parameter passed (comparing with the function or procedure definition), but it also checks that a parameter defined to be passed by reference is a variable, because something which is not a variable (constant, expression result or function call result) can not be referenced with an address. In this case the compiler will give an error message.

Prototypes:

The language also includes function and procedure prototypes to enable calls to functions (or procedures) to precede their definition. The most obvious use of this is for two functions (or procedures) which are cross-recursive (A calls B which again calls A).

The prototypes must be defined inside a special block used to define prototypes and its syntax is the head of the function or procedure itself.

```
Syntax:   "PROT"
            ( { "EXPORT" } { "ASYNC" }
              "PROCEDURE" { "::" } proc_name "(" param_list ")"
                    { "INVERSE OF" name } ";"
            | { "EXPORT" } { "ASYNC" }
              "FUNCTION" { "::" } func_name "(" param_list ")"
                    "RETURNS" type_name
                    { "INVERSE OF" name } ";"
            | ";"
            )+
          "ENDPROT"
```

The definition of parameters in a prototype is only used for type checking when the function or procedure is called. The prototype definition causes the creation of an entry for the function in the symbol table, allowing this symbol to be used in a call sentence before the function or procedure called is totally defined.

There is also the possibility of declaring that a function or procedure has an inverse (also a function or procedure). This declaration is only allowed in the prototype definition and the parser checks, at the end of the prototypes block, that the prototype of the inverse function or procedure (whose name has been put after "INVERSE OF") has also been declared. The two prototypes

must be compatible in parameters, i.e. the two prototypes have the same signature. This is used to optimize the journaling UNDO functionality (explained in section 8.1.3)

The GETDATA function:

The GETDATA function is a special function known by the compiler which requests input data of a certain type.

Syntax: "GETDATA" "(" string-expression ")"

where the expression must evaluate to a string and represents a set of characters shown to the end user to identify the request.

This function is accepted by the compiler only in two places:

- at the right side of an assignment sentence which causes a request for an input data having the same type as the variable to be assigned,
- as a parameter of a function or procedure call which causes an input data request having the type corresponding to this parameter. This case is not allowed if the parameter is passed by reference.

The execution of this function causes a request of input data to the system and the corresponding command execution waits until this input data is given by the system.

In the current version of ATLAS, the GETDATA function sends a request which does not have expire time (with a -1 as timeout value –see section 6.3). A possible extension to allow the use of other interaction modes (different timeout values) in the GETDATA function is presented in chapter 10 (section 10.3).

This GETDATA function is not asynchronous in itself even though it is based in the asynchronous input data mechanism offered by ATLAS (see section 6.3). Although it causes the command execution to wait until it gets the input datum requested, it does not stop the whole system. Other commands can be executed while this one is waiting for an input datum.

The "main" procedure:

The "main" procedure can be seen as an initialization procedure of the module because it is only executed once after the successful compilation of the module. This procedure doesn't have any parameter and cannot be called neither from any other function or procedure nor directly from the interactive Command Subsystem like the other procedures or functions can. Its prototype is fixed:

PROCEDURE main ();

It can be defined as any other procedure.

There are no restrictions on its internal sentences block because its behaviour is the same as any other procedure in the module. The only restriction is that the module cannot define another procedure or function using the same name ("main") with a different prototype.

Other predefined global functions:

There is also the possibility of calling some functions which are defined as global functions of the system. These functions include some mathematical functions, translations from integer or real to string and vice-versa, and the possibility to produce input data directly from an ATL command.

In particular these ATL default global functions are defined with the following prototypes:

– Mathematical functions:

real **exp** (real);	real **asin** (real);
real **fabs** (real);	real **acos** (real);
real **sqrt** (real);	real **atan** (real);
real **log** (real);	real **sinh** (real);
real **log10** (real);	real **cosh** (real);
real **sin** (real);	real **tanh** (real);
real **cos** (real);	real **atan2** (real, real);
real **tan** (real);	real **pow** (real, real);

– Translation functions:

integer **atol** (string);	string **ltoa** (integer);
real **atof** (string);	string **ftoa** (real);

– Input data functions:

atl_send_input ('any_type');

where 'any_type' can be any type previously defined. This function assigns -1 as a timeout value to the input data (see section 6.3).

atl_send_input_tout ('any_type', integer);

where the second parameter is the timeout value assigned to the input data.

Therefore, in order to produce an input data with a non -1 timeout value, you must use the 'atl_send_input_tout' call instead of the 'atl_send_input' call. Except for this difference, the ATLAS 'atl_send_input' call works exactly as the one with the same name in the ATLAS utility library described in chapter 9.

4.1.8 External functions and procedures

Usually an important part of any ATL module code consists of calls to functions or procedures which are defined externally to the Command Subsystem. These are the functions that constitute the core of the application. They must be implemented in the application process and declared as external prototypes in the ATL module.

Module associated to a process:

In order to be able to identify the external functions or procedures, the ATL compiler must find the needed information in the symbol table. This information is given in a module *associated* to the external process. This module must have the same name than the process and must declare the external functions and procedures that the corresponding process implements and offers to be public, this means it should declare the *interface* of the process. To declare an external function the keyword "EXTERN" is used and it can be followed by the keyword "ASYNC" if this function can be called asynchronously. This declaration, as the others already explained, must be placed inside the prototypes block.

```
Syntax:   "EXTERN" { "ASYNC" }
             ( "PROCEDURE" proc_name "(" param_list ")" ";"
             | "FUNCTION" func_name "(" param_list ")"
                  "RETURNS" type_name ";"
             )
```

Although this *associated* module is declaring the required external functions for the external process, it is also an ATL module and can declare anything the other Command Subsystem modules can.

This association between an ATL module and an external process allows the application programmer to encapsulate in modules the processes together with command language tools that use these processes.

Parameters:

An external function call provokes an execution of this function in the process that implements it. The parameters of this call must, then, go through the network because the process can be executed on another machine. In order to avoid the problem of having different representations of data in different architectures, ATLAS relies on the external data representation (XDR). This representation of data is totally managed by the system and the application developer doesn't have to be aware of it.

The parameters in this kind of function calls can be also passed by reference, but in this case it doesn't make sense to pass to another process a physical address. The effect of a parameter passed by reference to an external routine is to use a mechanism of *copy and return* of the corresponding parameter. The variable is copied to the process, the function modifies it and it is returned with the new value to the Command Subsystem to substitute the old value by the new one. It follows a *"copy-in copy-out"* paradigm.

4.1.9 Comments

The compiler of the ATL language also accepts comments inside the module. These comments are like those accepted in C++. A comment can start with "//" and finish at the end of the line, or start with "/*" and finish with "*/". Comments are accepted at any point of the module definition where a space would be acceptable, and are skipped by the compiler.

4.1.10 Other constructions

There is an extra sentence not included as part of the language proper, but which must be recognized also by the compiler because it is him who interprets all user commands.

Syntax: "KILLPROC" proc_name ";"

This "KILLPROC" sentence is a sentence of the language more oriented to the developer than to the final user. This sentence causes the `distr` process kill the 'proc_name' application process, and can be useful at developing time when a process is inside an infinite loop, for example.

"KILLPROC" is only allowed as an interactive sentence, i.e. is a sentence to be introduced interactively by the user (developer), it is not allowed to be inside a module file.

4.2 The ATLAS Command Subsystem

The `Command Subsystem` is the ATLAS component charged on interpreting the ATL language. It is divided in three independent processes connected to each other (figure 4.2):

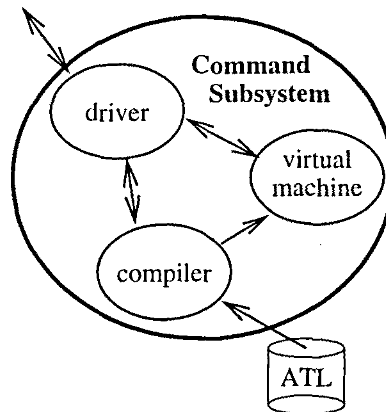


Figure 4.2: Command Subsystem internal scheme.

- The communications driver is the one that manages the communication between the Command Subsystem and the rest of the ATLAS system. It is the only part communicating with the master process `distr`. This driver has also direct communication with the other two parts of the subsystem.
- The compiler of the ATL language is able to read the ATL code both interactively from the user (it comes from an Input Subsystem through the `distr` process) or from a file written in that language.

The compiler compiles the ATL language code and generates an intermediate code that will be interpreted by the Virtual Machine (the third component).

- The Virtual Machine is the component in charge of the execution of the ATL code. Its purpose is to interpret the intermediate code generated by the compiler. Among its functionalities it keeps information on the external routine calls which have to wait for return values, generates requests of input data to the system and is able to manage different command executions as different execution *threads* [1] (allowing certain level of asynchrony).

  It is also able to manage asynchronous calls, by using a mechanism of "dirty variables". It tags all output parameters or return values of an asynchronous call as "dirty", and any attempt to use one of them as an r-value freezes the executing thread. Thus asynchronous calls may be issued and other portions may properly await their completion in a transparent manner.

4.2.1 The `Command Subsystem` driver

The driver for this ATLAS component is like the driver for any other ATLAS process (explained in chapter 7). In this case the management it does is a little bit different because it just acts as a message routing process handling the communications with the compiler and the Virtual Machine processes. When the driver receives a command message from the `distr` process it sends it to the compiler to be compiled and interpreted. Messages coming from `distr` which are not commands are directly sent to the Virtual Machine, and messages arriving from the Virtual Machine are directly sent to the `distr` process.

The compiler and Virtual Machine also hold a direct connection between them used to serve executable code to the Virtual Machine (this is explained in the Virtual Machine subsection below).

4.2.2 The ATL compiler

The ATL compiler is able to receive the orders interactively, and considers any routine call made out of a block (outside any function or procedure definition) as an execution order causing its immediate execution by the Virtual Machine. This facilitates a rapid improvement and debugging of code and also offers the final user the possibility of using ATL as a macro language.

Parsing the language

The PCCTS package (Purdue Compiler-Construction Tool Set) [27] has been used to generate the ATL compiler. The parser generator ANTLR (included in PCCTS) is able to generate C++ output. It has three things that make it

[1] They are not low level threads, just different commands being executed concurrently, i.e. when one of them must wait for any response another one can start its execution and so on.

generate strong parsers: $k > 1$ lookahead, semantic predicates (the ability to have semantic information direct the parse), and syntactic predicates (selective backtracking).

Since PCCTS also includes a template for symbol table manager, we just added the specified fields for the sort of registers that the ATL compiler needs.

Each register of the symbol table has a general description with a field pointing to a different object depending on the type of the register and the contents of it. Figure 4.3 shows schematically the structure of these registers.

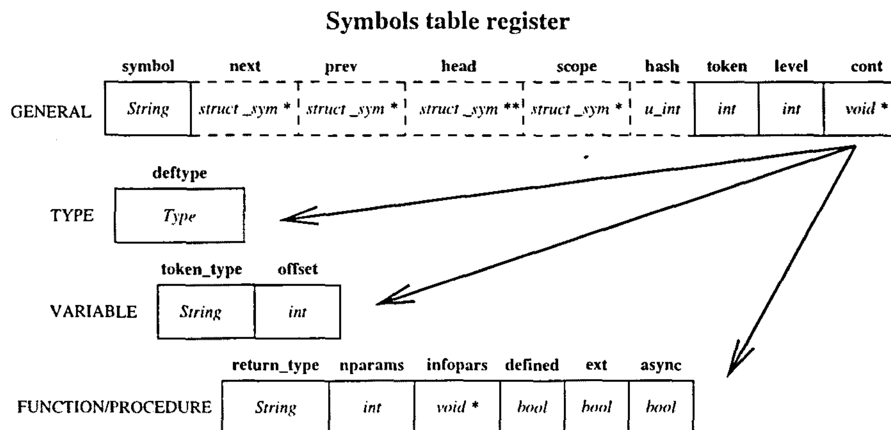


Figure 4.3: structure of a symbol table register

The contents of the different kinds of registers are:

Type: When a type is defined the compiler only needs to keep the *"Type"* object built for it.

Variable: For a variable declaration, it needs to keep the name of the type of the variable (for type checking) and the offset to the variable in the stack in case it is a local variable.

Routine: When a function or procedure is defined the data kept in the symbol table are: the return type of the routine (it is "_void" for procedures), the number of parameters, the information of these parameters (type name and flag indicating whether it is passed by reference or not), a flag saying whether the routine has been defined or just declared, a flag for being or not an external routine, and another flag to say whether it is declared asynchronous (allowed to be called asynchronously).

Intermediate code generation

The intermediate code language has been defined as a finite number of independent instructions which can carry none, one, two or three operands. These instructions can be seen as similar to the byte codes of Java [28, 29] or Basic or

as an assembler language. These instructions will be interpreted by the Virtual Machine process.

The intermediate code is generated by the compiler from the ATL language code in order to be executed sequentially and each instruction independently of the others. This execution is totally managed by the Virtual Machine (see next subsection).

As an example, figure 4.4 shows the intermediate code generated from a block of sentences in ATL. For this example it is assumed that `'sum'` is declared in the module scope and `'i'` is a local variable of the routine implementing this code block.

```
                                           GETN   "sum"
sum = 0;                     -->           PUSH   0
                                           MOVI
                                           GETV   offset
i = 1;                       -->           PUSH   1
                                           MOVI
                                     a:    GETV   offset
                                           PUSH   10
WHILE i <= 10 DO             -->           LEQI
                                           BRF    b
                                           POP
                                           GETV   offset
                                           PUSH   2
                                           MOD
   IF i % 2 == 0             -->           PUSH   0
                                           EQI
                                           BRF    c
                                           POP
                                           GETN   "sum"
                                           GETN   "sum"
                                           GETV   offset
      THEN sum = sum + i;    -->           ADDI
                                           MOVI
                                           BR     d
                                     c:    POP
                                           GETN   "sum"
                                           GETN   "sum"
      ELSE sum = sum - i;    -->           GETV   offset
                                           SUBI
                                           MOVI
                  ENDIF
                                     d:    GETV   offset
                                           GETV   offset
   i = i + 1;                -->           PUSH   1
                                           ADDI
                                           MOVI
                                           BR     a
ENDWHILE                     -->     b:    POP
```

Figure 4.4: Example of the intermediate code generation for a block of ATL sentences

This intermediate code instructions sequence generated by the compiler is stored into a code table which will be sent to the Virtual Machine to be executed. Each function or procedure defined will have its own code table.

4.2.3 The Virtual Machine

The Virtual Machine is the `Command Subsystem` component which interprets the intermediate code generated by the compiler. It controls the interaction among

the application processes, which is programmed by the developer in ATL, and also has a direct communication with both the compiler of the ATL language and the Command Subsystem communications driver.

The communications driver of the Virtual Machine listens from both communication channels. The communication with the compiler is only one way, the compiler sends messages to the Virtual Machine but not the other way. The communication with the Command Subsystem driver is in both directions, the Virtual Machine driver receives from and sends to this communication channel.

Information from the compiler

The information the compiler sends to the Virtual Machine is related to two main tasks:

- compilation task → messages related with this task are those giving information about the compilation of different elements that must be known at execution time (global variables declaration, procedures or function definition, etc),
- execution task → since the compiler is able to receive orders interactively, it must generate messages requesting execution which have to be interpreted by the Virtual Machine.

The structure encapsulating the communication channel with the compiler can be seen in figure 4.5, where the base class *"Comunic_Pipe"* is an abstract class encapsulating the communication channel descriptor and the ATLAS messages reception management in case of a pipe channel.

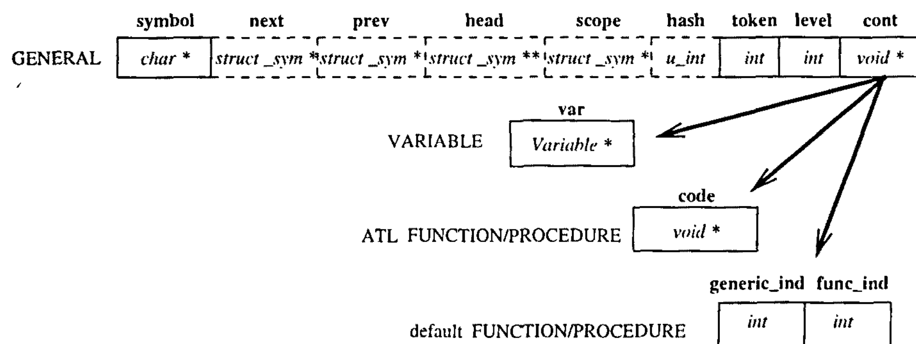
```
class Comunic_Comp : public Comunic_Pipe
{
    char missgen;                 // distinguish between the two kind
                                  // of messages
    taula<Instruccio> *coditmp;   // pointer to the intermediate code
                                  // table being generated at any moment

    void Handle_Message (Message *miss);       // message treatment
    friend void interpret (ExecStep &estat);

  public:
    Comunic_Comp (int fd) : Comunic_Pipe(fd) {}
};
```

Figure 4.5: The *"Comunic_Comp"* class interface

The Virtual Machine symbol table

The messages sent by the compiler during compilation are needed by the Virtual machine to keep some information only known at compilation time. This information is stored in the Virtual Machine symbol table, which is smaller than the compiler's, and the information stored is different. This symbol table will also keep information corresponding to those default functions recognized by the ATL language (more about this in *"Default functions management"* below).

The different registers of this symbol table are for variables and functions or procedures. The first correspond to global variables and keeps the address where the variable is stored. The others correspond to functions or procedures that can be: defined in the ATL language, which keep the address of the intermediate code table that implements the procedure or the function; or the "default" procedures or functions which are those known by the language without having to be defined in any module (mathematical functions, translations from "integer" or "real" to "string" and vice-versa and procedures to produce input data to be sent to the `distr` process). These "default" procedure or function registers store two integers representing the indices of the tables that store the addresses of the functions to call (see *"Default functions management"*). The symbol table does not need to keep information about external functions because the compiler includes it in the operands of the intermediate code instruction which triggers the call.



The field 'level' of the general register is used to indicate if a variable or a function/procedure has been temporarily deleted from the symbol table. This situation can be produced, for example, by an UNUSE of a module, and the Virtual Machine keeps the same address for the register in order to avoid problems with other modules referencing this address, because the user can load again the module afterwards (eg. with USE).

The messages that the compiler sends to generate and update this symbol table and which are handled by the *Handle_Message* method of the *"Comunic_Comp"* class, are the following:

DECL_FUNC: *Function declaration* message. This message is sent by the compiler when a prototype of a function or a procedure internal to the Command Subsystem is parsed. It contains the name of the function or procedure.
V.M. behaviour:[2] First it checks if there is a register in the symbol table with this name (it can be a repeated prototype or it could have been deleted because of an UNUSE of the module –see the REMOVE message in the next section). If it already exists it revalidates its definition (sets its field 'level' to 1) and points the field 'code' to NULL. If it didn't exist, it creates a new

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

function type register to the symbol table and initializes its 'level' field to 1 and its 'code' field to NULL.

DECL_VAR: *Variable declaration* message. This message is sent by the compiler when a global variable is declared. It contains the name of the variable, the name of its type and its *effective type.*
V.M. behaviour: First it checks if there is a register in the symbol table for this variable. If it already exists it revalidates its definition (sets its field 'level' to 1) and creates the *Type* and the *Variable* assigning its address to the field 'var'. If it didn't exist, it creates a new variable register in the symbol table, initializes its 'level' field to 1 and creates the *Type* and the *Variable* and assigns it to the 'var' field.

DEF_FUNC: *Function definition* message. This message is sent by the compiler when it detects that the definition of a function or procedure internal to the Command Subsystem is going to start. It contains the name of the function or procedure.
V.M. behaviour: First it checks if there is a register in the symbol table for this function. If it already exists and it is already valid (its 'level' field is 1) it just needs to assign the 'code' field to the intermediate code table where the corresponding instructions will be stored, if the register was not valid it has to create also the function part for this register and validate it (set its 'level' field to 1). On the other hand if the register didn't exist it has to be created and set with the corresponding values.
Finally the address of the code table created is also assigned to the temporary code table of the communication wrapper ('coditmp' field of *"Comunic_Comp"*), because the instruction messages following this one will be part of this code table just created for this function. The value of the flag 'missgen' of *"Comunic_Comp"* is set also to DEF_FUNC.

INSTRUCT: *Intermediate code instruction* message. This message is sent by the compiler each time it generates an intermediate code instruction. It contains the instruction and its operands.
These instructions always arrive after a function definition message or an execution message (see *"The execution messages"* below). In the first case they are instructions to be stored into the code table of the function and in the second case they are instructions to be stored into a temporary code table to be executed.
V.M. behaviour: First of all a preprocessing is needed in order to change the operand in two cases (see also the intermediate code instructions description in appendix B):

CINT (code = 42)	the compiler sends the name of the function to be called and the Virtual Machine has to change it depending on the function to be called. In case the function is defined in the ATL language the operand must be changed to have the address to the symbol table for this function; but in case it is a default function the preprocessing has to change the instruction to the CDEF instruction (code = 58) and with the indices for this function (stored in the symbol table) as operands. This last instruction will call the corresponding default function when executed.

GETN (code = 53) the Virtual Machine changes the name of the global variable for its address stored in the symbol table

After this preprocessing the instruction is appended to the code table currently being created.

Dummy: Empty message. This message is sent by the compiler to indicate that the code table being sent instruction by instruction is finished.

It checks the flag 'missgen' of *"Comunic_Comp"* to decide whether it was defining a code table for a function or it was receiving a code table to be executed.

V.M. behaviour: The flag 'missgen' of *"Comunic_Comp"* is set to DUMMY and the 'coditmp' field is set to NULL.

In case it was receiving a code table to be executed the Virtual Machine also creates a new *"ExecStep"* object and the routine *interpret* is called to execute this code (see below).

The execution messages

The execution messages are those involved in requesting the execution of some code to the Virtual Machine. They come from the compiler as a result of commands entered by the user or of the compiler's own processing.

Besides the already discussed 'INSTRUCT' and 'Dummy' messages, which can also be execution messages, there are the 'EXECUTE' and 'REMOVE' messages which can only happen in connection with an execution.

EXECUTE: *Execution order* message. This message is sent by the compiler when it detects that a code table must be executed (function or procedure call between blocs or execution of the 'main' procedure). It doesn't contain data.

V.M. behaviour: It generates a new code table at the 'coditmp' field (this code table will be temporary and is only used for this execution), and updates the 'missgen' flag of *"Comunic_Comp"* to EXECUTE.

REMOVE: This message is sent by the compiler in three cases and includes a flag to distinguish among them. The cases are:

- when it has detected any compilation error and must remove some entries from the symbol table;
- when an UNUSE or a REUSE for a module is seen and it has to eliminate an intermediate code table from the symbol table;
- when an UNUSE or a KILLPROC for a module is seen and it has to indicate the Virtual Machine that the process is dead.

In the first two cases, besides the case-selection flag, the message contains the name of the entity to be removed from the symbol table. In the third case it contains the name of the process.

V.M. behaviour: In the first two cases the symbol table must be modified. If it is a compilation error (first case) it removes the register totally from the symbol table, but in the second case it only removes its contents and marks the register as not valid ('level' field set to 0).

Besides the necessary changes to the symbol table, in the second and third cases the possibility of having some executions of this module waiting for some answers from the system exists, so these stopped executions cannot continue if the code table being executed is removed from the symbol table. Same is true for those executions waiting for external returns from the process associated to the module being UNUSEd, even if they do not have to execute any internal code table, because the return may never come from that process.

V.M. behaviour: This elimination of pending executions is done by the routines *neteja_llistes* and *neteja_externes*.

The first one receives the register of the symbol table to be treated. First this routine checks if this register is for a function/procedure and otherwise returns. If it is a definition of a function/procedure –and it is not a default function– it takes the pointer to the code table to be eliminated and looks for it at waiting lists comparing it with every code table in each waiting execution. When it finds a coincidence:

- If it was in the list of executions waiting for input data, it generates a message of "removing request" with the identifier of the request and sends it to the `distr` process. Then it deletes the stacks for this execution and removes the *WaitingInput* from the list (see *"Data request"* below).
- If it was into the list of executions waiting for results of external functions (synchronous) it deletes the stacks for this execution and removes the *ExecStep* object from the list (see *"The execution"* below). The cases for asynchronous calls are explained in next section (4.3).

After this it checks if the Virtual Machine was waiting to finish and these were the last executions to manage and in this case it finishes the Virtual Machine execution.

The *neteja_externes* routine receives the name of the module and looks for those executions awaiting results from external routines. If the external routine it is waiting for is a routine of this eliminated process it removes the *ExecStep* from the list (in case the routine was synchronous) and deletes it. The case for asynchronous calls is treated in section 4.3. This routine also checks if the Virtual Machine was waiting to finish and acts as the others.

The intermediate code structure

The intermediate code language is a finite set of independent instructions that can use zero, one, two or three operands depending on the instruction. These instructions are similar to the instructions of an assembler language.

Internally the intermediate code instructions are represented by a class, *"Instruccio"*, containing a code indicating the instruction and three operands that can be instantiated or not depending on the instruction represented. The methods offered by this class are basically the different constructors (with zero, one, two, or three operands) and the access methods to the instruction code or to any of its operands (see figure 4.6).

```
class Instruccio
{
    short codi;                          // instruction code
    Operand oper1,oper2,oper3;           // operands

  public:
    Instruccio () {}
    Instruccio (short cod) : codi(cod) { }
    Instruccio (short cod, Operand &op1) : codi(cod), oper1(op1) { }
    Instruccio (short cod, Operand &op1, Operand &op2)
       : codi(cod), oper1(op1), oper2(op2) { }
    Instruccio (short cod, Operand &op1, Operand &op2, Operand &op3)
       : codi(cod), oper1(op1), oper2(op2), oper3(op3) { }

    short Codi () { return codi; }
    Operand & Getoper (int i) { switch (i) { case 1: return oper1;
                                             case 2: return oper2;
                                             case 3: return oper3; }
                              }
};
```

Figure 4.6: The *"Instruccio"* class interface

An operand is represented by another class *"Operand"* which is a base class for the classes representing the different types of operands. This base class contains the name of the operand type, a flag saying whether it is a constant or not and a union of fields representing the different data types that an operand can represent (see figure 4.7). Each derived class will access the corresponding field of the union.

This class uses two C++ classes *"Variable"* and *"node"*, which allow it to keep a variable into the tree structure of its type. These classes are more extensively used for the representation of data going through the communications mechanism because they permit the encapsulation of any type of data (more extensively explained in chapter 7), but they are also used by the Virtual Machine to store and access data.

- The *"Variable"* class contains the name for the variable, a *"Type"* instance for its type and a pointer to a *"node"* object which is the root of the tree representing the variable. The *"Type"* class encapsulates the type definition as well as its *effective type* which expresses the type in terms of primitive types.

  An important method in this class is the creation of the tree that mimics the type's structure, made up of objects of the class *"node"*.

- The *"node"* class is an abstract class with different kind of nodes derived from it. They are a node for a structure having a set of node pointers for its fields, a node for a vector having an array of node pointers for its elements, and four nodes for the basic types (integer, real, string and boolean) containing the values of the atomic components of the variable.

The classes derived from *Operand* are the following:

"Operand_int" ⟶ Operand containing an integer constant.

"Operand_float" ⟶ Operand containing a float constant.

```
class Operand
{
  protected:
    String tipus;                // type name of the operand
    int switch_t;                // constant/pointer to a node/pointer to a Variable
    union  { Variable *ptrvar;
             void *punter;
             io_abstract *varbl; // can contain an io<node *> or just an io_base if it
                                 // is only a tag (atl_ticket)
             long enter;
             float real;
             char *string;
             bool boolea;
   } contingut;

  public:
    Operand () { tipus = ""; }
    Operand (const Operand &op);              // make a copy of the operand
    Operand &operator = (const Operand &op);  // reserving space in case
                                              // it is a 'char *'
    String Gettipus () const { return tipus; }
    int Switch_t () const { return switch_t; }
    bool Isconst () { return (switch_t==0); }     // the operand is a constant
    bool Isnode () { return (switch_t==1); }      // the operand is an io_base pointer
    bool IsVariable () { return (switch_t==2); }  // the operand is a Variable pointer
    void Buida ();       // removes the content of the operand, when it is a "string"
                         // it deletes the 'char *' and when it is an io_abstract pointer
                         // it decreases its references (also for the node if needed)
    ~Operand () { if ((switch_t==0) && (tipus=="string") &&
                      (contingut.string!=NULL))
                    delete [] contingut.string; }
    friend ostream& operator << (ostream& s, const Operand &op);
};
```

Figure 4.7: The *"Operand"* class interface

"Operand_string" ⟶ Operand containing a string constant.

"Operand_bool" ⟶ Operand containing a boolean constant.

"Operand_var" ⟶ Operand containing a pointer to the wrapper structure defined to deal with *global data identifiers* (see *"Global data identification"* in chapter 8). In this case it wraps a *"node"* pointer.

"Operand_punter" ⟶ Operand containing a void pointer. This kind of operand is used to keep the pointer to the intermediate code table at the activation blocs in the execution stack (see subsection *"Stacks management"* below).

"Operand_ptvar" ⟶ Operand containing a pointer to a *Variable.*

This *Operand* structure is also used in the stacks management of the execution (see next subsection).

The intermediate code that the compiler generates and that must be executed by the Virtual Machine is structured as a set of code tables (one for each procedure or function) which are tables of instructions of this intermediate language.

The execution

When a 'Dummy' message arrives after a code table has been received for an execution, an *ExecStep* object is created and passed to the *interpret* routine to be executed. This *"ExecStep"* class encapsulates an execution status in the Virtual Machine. It keeps pointers to the code tables that become part of this execution (i.e. they have been called from this execution) and pointers to the execution and temporary stacks. It also keeps the index of the code table where the execution is at the moment and the base pointer to the execution stack (lenght of the stack once the activation bloc is stored). The definition for this class is shown in figure 4.8:

```
class ExecStep
{
    DLList<taula<Instruccio> *> l_code;  // code tables in the execution
    VStack<Operand> *tmpstack;
    VStack<Operand> *execstack;
    int base_ptr;
    int index;

    // Atributes to manage internal asynchronous calls
    bool async;                // flag for the first call in this thread
    int callscount;            // counts the nested calls in the thread
    bool justnew;              // this thread has been just created
    int code_call;

    // Atributes to maintain the execution thread number for the journaling
    unfourbytes thread;

    // Atributes to preserve some aspects of the execution environment that can be
    // used after a stop in this execution
    OpenedCall *towait_actiu;
    DirtyVar *pervalidar;

    void actualize (int b, int i)        // update the status with the new data.
          { base_ptr = b; index = i; }
    void add_table (taula<Instruccio> *t) { l_code.prepend(t); }
    void sub_table () { Pix i = l_code.first(); l_code.del(i); }
    // functions that need a direct access to the structure
    friend void interpret(ExecStep &estat);
    friend void interp_tracta_dades (AnswerData dades);
    friend void interp_tracta_ret_valor (ReturnValue &dades);
    friend void interp_tracta_mort (String nomproc);
    friend void neteja_llistes (Sym *a);
    friend void neteja_externes (String nom);
    friend void interp_tracta_ret_void (ReturnVoid dades);
    friend void interp_tracta_ret_param (ReturnParam &dades);
  public:
    ExecStep (taula<Instruccio> *t, int code, unfourbytes th);
        // initializes a status from a code table to be executed.
    ExecStep (taula<Instruccio> *t, int code, VStack<Operand> &tmp, unfourbytes th);
        // initializes a status copying the temporary stack (for asynchronous calls)
    OpenedCall *Towait_actiu () const { return towait_actiu; }
    void Activa_towait_actiu (OpenedCall *oc) { towait_actiu = oc; }
    ~ExecStep ();       // deletes all pointers different of NULL.
};
```

Figure 4.8: Interface for the *"ExecStep"* class

The *interpret* routine receiving a reference to an *ExecStep* object interprets (executes) the intermediate code. To start an execution the *ExecStep* is initialized (empty stacks and index and base pointer 0). An execution can be stopped waiting for some data coming from elsewhere, so in these cases it sets the execution status (encapsulated in an *ExecStep*) to be able to continue the execution later at the same point. Only the 'base_ptr' and 'index' values need to be

set, as the rest are references.

The implementation of this routine is just a loop ending when it finds the instruction END and executing each instruction in the code table being interpreted (see appendix B). When there is an instruction that has to stop the execution, the treatment for this instruction updates the *ExecStep* with the information needed to know where it will have to restart the execution (see subsection *"Communications with the Command Subsystem driver"*) and returns from the routine.

This implementation of the interpreter allows for different commands to be concurrently executed. At each point in time, one will be active, while the others are suspended awaiting input or external routines.

Stacks management

The Virtual Machine uses two different stacks in the execution of the intermediate code. One of them, the *"temporary stack"*, is used for the intermediate computations and assignments, and the other, the *"execution stack"*, is used to store all the activation blocs of the execution and the parameters and local variables of the functions or procedures called as well. The order in which the parameters are going to be stored into the execution stack is the same order as they appear in the calling sentence, or the function declaration. So, the order is: first the `param_1`, then the `param_2`, and so on until the `param_n`. The scheme in figure 4.9 shows how data are stored into the execution stack.

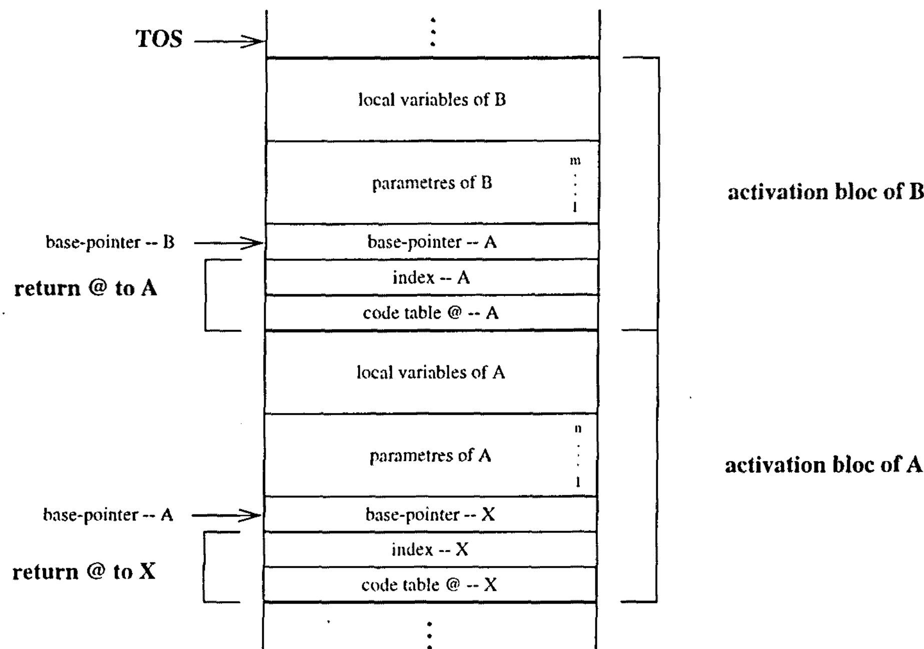


Figure 4.9: Scheme of the use of the execution stack

In appendix B the description of all intermediate code instructions that the Virtual Machine accepts is explained. For each instruction it explains also its operands and the effect of its execution.

Default functions management

The ATL language offers the possibility of using some mathematical functions inside the function or procedure definitions so that they are directly executed by the Virtual Machine. It also offers functions to translate from "integer" or "real" to "string" and vice-versa (which will be useful to send numerical data to the Tcl interpreter, for example), and the ATLAS API routine "atl_send_input" that allows the user to produce input data from the language. All of these will be generically referred to as the ATL default functions.

The prototypes of the predefined functions in the language have been already described in *"Other predefined global functions"* in section 4.1.7.

Since all these functions do not have the same parameters, and the parameters must be taken from the temporary stack, the treatment needed for each function will be different depending on the number and type of the parameters. To manage these differences a table is used to store generic functions. These generic functions are those that the interpreter of the Virtual Machine actually calls, and they all use two parameters: an index into a table of functions and a reference to the temporary stack. All functions called through one generic function share the same prototype.

The second table of functions is initialized with the above standard library functions, plus two specific functions to generate input data to ATLAS. These functions will create an *InputData* message from the *Variable* received as a parameter and will send it to the `distr` process producing an input data with the appropriate timeout.

This method can be easily extended to provide other utility functions to the interpreter.

Communication with the `Command Subsystem` driver

The communication between the Virtual Machine and the Command Subsystem driver channels the communication between the Virtual Machine and the ATLAS `distr` process. Therefore all the messages going from the Virtual Machine to the Command Subsystem driver are finally sent by the driver to the `distr` process, and when the Command Subsystem driver receives a message from the `distr` process which has to go to the Virtual Machine it just redirects the message to it. So in fact the communication between the Virtual Machine and the Command Subsystem driver is used as a bridge for a communication between the Virtual Machine and the `distr` process.

This communication channel is implemented physically by a stream socket (the ACE-wrappers library is used for this implementation), and the *"Com_sc_int"* class encapsulates the stream socket and the messages reception management used by ATLAS for its socket communications. It also contains a stream associated to the channel descriptor at creation time (see figure 4.10).

The messages sent by the Virtual Machine to the `distr` process are:

RequestData: Produced by the execution of the instruction 'REQD'. It causes a request of input data to the system.

```
class Com_sc_int : public MyEventHandler
{
    ACE_SOCK_Stream canal;      // communication socket
    Receiver_socket rebuts;     // main object for the reception of messages
                                // used by the ATLAS communications
    FILE *fd_stream;            // stream associated to the file descriptor

    void Handle_Message (Message *miss);    // management of the message
  public:
    Com_sc_int () : rebuts(this) { fd_stream = NULL; }
    Com_sc_int (ACE_SOCK_Stream c) : rebuts(this), canal(c)
         { fd_stream = NULL; }
    void set_fd (int fd);          // assigns the fd to the channel and
                                   // opens the fd_stream
    int envia (Message *miss);     // management to send messages to ATLAS
    ACE_HANDLE get_handle () const { return canal.get_handle(); }
    int handle_input (ACE_HANDLE fd);  // implements the use of the
                                       // messages reception management
};
```

Figure 4.10: Interface of the *"Com_sc_int"* class

CallRoutine: Produced by the execution of the instruction 'CEXT' and the instruction 'PRINT' (which calls to the *Sortida* routine of the Input Subsystem). It calls a routine external to the Command Subsystem.

Parameter: Produced by the execution of the instructions 'PEXV' and 'PEXR', and also by the instruction 'PRINT' (which sends the needed parameters for the *Sortida* routine call). It sends a parameter of a previous external call.

Error: Produced when an execution error occurs in the intermediate code. It can be any of:

- Vector index out of range
- Undefined function
- Function without return value
- Function removed from the symbol table
- Variable removed from the symbol table
- Invalid operator for empty variables [3]
- Attempt to access an empty variable
- Empty variables cannot be dirtied.

DelDemand: Produced when an order to remove a module (UNUSE) is sent by the compiler and there was an execution waiting for an input data. It causes the request be removed from the system.

InputData: Produced when the default function "atl_send_input" has been called to send an input data to the system (see subsection *"Default functions management"*).

In case of error messages, input data being sent to the system or a remove of a request, the Virtual Machine does not have to do anything except sending the message. But on the other cases sending the message is not enough, the Virtual Machine needs also a specific management in order to be able to receive answers to these messages.

[3] An empty variable is a variable which only contains an atl_ticket (see also section 8.1.3)

Data request

The effect of a request of input data to the Virtual Machine is that the execution producing the request stops waiting for the data it has asked for.

In order to manage this wait and allow the Virtual Machine driver to know what to do when the data arrive, the Virtual Machine uses an object class *"WaitingInput"* which encapsulates the request identifier (to recognize the data when they arrive) and a reference to the *ExecStep* object which keeps the execution status when the request was produced, in order to be able to continue this execution.

The Virtual Machine keeps a list of these *WaitingInput* objects where all executions waiting for input data are appended. One of the messages the Virtual Machine will receive from the `distr` process is then:

AnswerData: *Answer data to a request* message. This message is sent by the `distr` process when it has an input datum agreeing in type name with the request produced by the Virtual Machine. It contains the identifier of the input datum, the identifier of the request and the *Variable* encapsulating the input datum.
V.M. behaviour: The treatment routine for this message (*interp_tracta_dades*) receives the message as a parameter, searches into the *WaitingInput* objects list and generates an *Operand_var* setting its node pointer to the tree of the *Variable* (incrementing its reference count). It also pushes this operand onto the corresponding temporary stack (*ExecStep.tmpstack*), and executes the *interpret* routine passing it the execution status (*ExecStep*).
When the *interpret* routine returns, it removes the *WaitingInput* because it has been already treated and checks if the Virtual Machine was waiting to finish and this was the last execution to manage and in this case it finishes the Virtual Machine execution.

External routine calls

In order to produce a correct external routine call the compiler generates intermediate code first to produce the routine call, 'CEXT', and then to produce the code for each parameter 'PEXV' and 'PEXR' instructions depending on the case. Finally (in case it is non an asynchronous call) it generates a 'WAIT' instruction which causes the execution to stop waiting for the results of this external call.

The *OpenedCall* object encapsulates the name of the routine called, the identifier assigned to it and a list of *Variable* pointers where the parameters passed by reference must store their values when they return. This class also has methods to add parameters to the list, to remove a parameter from the list (when the result for it has arrived) and methods to iterate through the parameters list.

In case the call is synchronous the execution must stop (with the WAIT instruction) and the *ExecStep* has the pointer to the *OpenedCall* corresponding to this call. The Virtual Machine then keeps a list of *ExecStep* which contains all executions waiting for results of external calls (synchronous).

In case the call is asynchronous, the execution calling the external routine does not need to stop, so a list containing all the *OpenedCall* objects corresponding to the external asynchronous routines which have not returned yet is also kept by the Virtual Machine (see also section 4.3).

ReturnVoid: *Void return* message. This message is produced by the return of a void function which does not have parameters passed by reference. It contains the identifier of the routine called.
V.M. behaviour: The routine to treat this message (*interp_tracta_ret_void*) receives the message as a parameter and searches the *ExecStep* (for synchronous call) or *OpenedCall* object (for asynchronous call) for this routine. If it was in the synchronous calls it executes the *interpret* routine passing to it the adequate execution status (*ExecStep*).
If it was in the asynchronous calls list it just removes the *OpenedCall* from the list.
After this it checks if the Virtual Machine was waiting to finish and this was the last execution to manage and in this case it finishes the Virtual Machine execution.

ReturnValue: *Value returning* message. This message is produced by the return value of an external function called. It contains the identifier of the routine called and a pointer to the *Variable* containing the value returned.
V.M. behaviour: The routine to treat this message (*interp_tracta_ret_valor*) receives the message as a parameter and searches the *ExecStep* (for synchronous call) or *OpenedCall* object (for asynchronous call) for this routine. If it was in the synchronous calls list it generates an *Operand_var* setting its node pointer to the *Variable* (incrementing also its reference count), pushes the operand to the temporary stack of the corresponding execution (*ExecStep.tmpstack*), removes the *Variable* from the message and if there are no parameters to wait for (parameters passed by reference to this routine call) executes the *interpret* routine passing it the execution status (*ExecStep*).
In case there were parameters to wait for, it waits until all the expected values have arrived, before calling *interpret.*
When the *interpret* routine returns it removes the *WaitingCall* because it has been already treated and checks if the Virtual Machine was waiting to finish and this was the last execution to manage and in this case it finishes the Virtual Machine execution.
The explanation for the asynchronous call return value is included in the "dirty variables" mechanism explained in next section.

ReturnParam: *Parameter return* message. This message is returned by each parameter passed by reference to the external routine call. It contains the identifier of the routine call and a pointer to the *Variable* containing the result for the parameter.
V.M. behaviour: The routine to treat this message (*interp_tracta_ret_param*) receives the message as a parameter and searches the *ExecStep* (for synchronous call) or *OpenedCall* object (for asynchronous call) for this routine. If it was a synchronous call it sets the first pointer on the parameters list of the *OpenedCall* in the *ExecStep* to the *Variable*, then it removes the parameter from the list and if there are no more parameters to wait for, it executes the

interpret routine passing to it the execution status (*ExecStep*).
When the *interpret* routine returns it removes the *WaitingCall* because it has been already treated and checks if the Virtual Machine was waiting to finish and this was the last execution to manage and in this case it finishes the Virtual Machine execution.
In case there were more parameters to wait for, it waits until all the expected values have arrived.
The explanation for the asynchronous call is included in the "dirty variables" mechanism explained in next section.

Other messages from the **distr** *process*

The messages seen above are those the Virtual Machine receives as a response to other messages sent to the system before. But there are also other messages the Virtual Machine can receive from the Command Subsystem driver without being caused by the execution of the intermediate code. These messages are:

Dummy: Empty message. This message is the message sent to signal the completion of an XDR encoding of a variable. This message has no effect on the Virtual Machine execution but is used instead by its communication driver to complete the construction of the corresponding variable.

ExitExec: *Order to finish* message. This message is sent by the Command Subsystem driver to the Virtual Machine when it receives the command "quit".
V.M. behaviour: The routine to treat this message (*interp_cal_acabar*) checks if there are any executions waiting on the lists (for data or for returns). If the lists are empty the Virtual Machine execution finishes, but if they are not empty this routine sets a flag which indicates the execution should finish when the results or data waited for have arrived. The flag will be checked by the routines managing these results or data arrivals.

4.3 The *"dirty"* variables mechanism for the ATLAS asynchronous calls

4.3.1 General description

The general design for this mechanism is based on these main ideas:

- An asynchronous call implies the Virtual Machine does not have to wait for this call to finish.
- If the asynchronous call has parameters passed by reference or a return result assigned to a variable, all these variables are tagged as *"dirty"* variables when the routine is called.
- A *"dirty"* variable cannot be used as an r-value until it is *cleaned*. Therefore any other "parallel" execution of the Virtual Machine willing to use

one of these variables as an r-value is going to be frozen until the variable is again *clean*.

But if it is used as an l-value, even if it is still *"dirty"*, it becomes *"clean"* at this point.

- On the other side, when the asynchronous call is finished the variables *"dirtied"* by it are *cleaned* and executions waiting for these variables become ready to continue their work.

In this design an asynchronous call is only allowed as a sentence by itself or as the right side part in an assignment. Note that an asynchronous call in the middle of an expression is not accepted.

Having in mind this general design the mechanism can be described in more detail by dividing it into the different possible situations:

- *Dealing with the asynchronous call:*
  - this call alone as a sentence – no return result is needed
  - this call as the right side of an assignment
  - different implementations for external or internal asynchronous calls
- *Dealing with the possibly **"dirty"** variable:*
  - the variable as an l-value – used at the left side of an assignment
  - the variable as an r-value – inside an expression

In order to better understand the management done with variables in this mechanism, first we must clarify some ideas and define some concepts to be used in the explanation below:

- The ATL management of scopes (see section 4.1.1) considers three different scopes: global scope, visible from any routine in any module; module scope, visible from any routine in the same module; and local scope, visible only inside the routine owning this scope. There is then only one global scope, a module scope for each module and a local scope for each execution of a routine. The local and module scopes do not interfere each other.

- The entities marked *dirty*, in fact, are not the variables themselves, but the containers of the variables, i.e. the operand kept in the execution stack in case it is a variable in a local scope, or the variable register in the symbols table in case the variable is in global or module scope.

- A parameter, even though passed by reference, is considered belonging to the local scope inside the routine receiving it (see figure 4.11 as an example).

- We define the *calling level scope* as the scope of a variable when it is used as a parameter of a call. In figure 4.11, `param` in A is in this *calling level scope*. The *calling level scope* can be any of the three scopes depending on the own scope of the variable `param`.

```
PROCEDURE A () IS        PROCEDURE B (integer &par) IS
   ...                      ...
  B (param);              par = par+1-par*par;    // 'par' is in the
   ...                      ...                   // local scope
ENDPROCEDURE             ENDPROCEDURE             // of B
```

Figure 4.11: Different scopes for a parameter

- We define the *called local scope* as the scope of a variable inside a routine when it has been received as a parameter. In figure 4.11, par inside B is in the *called local scope*. The *called local scope* is always a local scope.

4.3.2 The asynchronous call as a sentence

A routine is called as a sentence, in both synchronous and asynchronous cases, usually when the routine is a procedure and does not have a return result. But in ATL it can be done also with functions having a result and in this case it means the return is not needed.

ATL example: `ASYNC routine (param1, ..., paramN);`

The only special treatment required then for an asynchronous call as a sentence is for those parameters passed by reference. Each variable being used as the actual parameter in a routine in association with a formal parameter that is passed by reference must be marked *dirty* in the *calling level scope* until the asynchronous call is finished. This parameter, however, will not be marked *dirty* in the *called local scope*.

When the asynchronous routine finishes, the variables can be *cleaned* in the *calling level scope* and any execution stopped waiting for these variables to be *clean* becomes ready to continue.

Some information should be kept when a variable becomes *dirty*. This information includes the variable that must be changed by the routine, the identifier of the asynchronous routine responsible of *cleaning* the variable and a list of executions that are stopped waiting for this variable. Therefore we keep a list of objects encapsulating this information.

Information about the asynchronous routines active at any time is also kept; this information is different whether the routine is external (belonging to an external process) or internal (implemented in an ATL module to be executed by the Virtual Machine) –see implementation details below, section 4.3.4–.

The use as a sentence of a function call, which therefore has to discard the return value, is usually done by adding a POP instruction after the call in order to take the result out of the temporary stack where it is pushed by the function. But in case of an asynchronous function call the same treatment wouldn't work, because the routine calling the asynchronous function can continue its execution without waiting for the result of it. In this case then the compiler uses a specific intermediate code instruction which indicates that the result of the function must be discarded.

4.3.3 The asynchronous call as the right side of an assignment

Let us consider the following ATL statement:

```
var = ASYNC function (param1, ..., paramN);
```

Besides the treatment for the parameters passed by reference, which is the same as described in the section above, an asynchronous assignment also requires a special treatment for the return of the function and for the variable to be assigned with this return.

The intermediate code generated by the compiler when it parses an asynchronous call in an assignment can be seen in figure 4.12 (with an internal call -CINT-, it is similar for an external one).

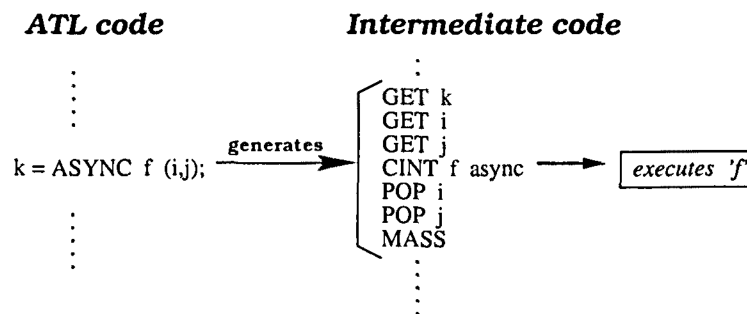


Figure 4.12: Intermediate code generation for an asynchronous internal call

The first four operands shown in figure 4.12 are the same as in the synchronous call case. The three GET instructions are just preparing the l-value of the assignment and the two parameters (i,j) for the function call (CINT). The preparation puts them into the temporary stack, where the CINT or the function called can access them. After the CINT instruction in a synchronous call there would be a MOV instruction because when the call finishes it pushes the result into the stack, then the MOV instruction just have to assign the first value on the top of the stack to the second value from the top, which is the l-value pushed before the call.

But for the asynchronous assignment the treatment must be a little bit different. The compiler thus inserts some additional information for the Virtual Machine:

- Although the Virtual Machine is able to distinguish between an l-value and an r-value (when it accesses to the value or assigns a value to the variable), it does this distinction at a low level of the computation. In order to have this information before, it is given with the access instruction by adding an operand to the GET instruction.
- After the asynchronous call the intermediate instruction to make the assignment (move instruction) must be also special, because it may not yet

have the value available. The compiler then generates a specific intermediate code instruction for this asynchronous move (MASS instruction).

The Virtual Machine is not doing anything special with the GET instruction if the variable was *clean*, but if it was *dirty* it acts differently depending on the flag "l-value/r-value". These effects will be explained below in section 4.3.5.

In the execution of the MASS intermediate code instruction, if the variable to be assigned was *clean* it becomes *dirty* and a *dirty variable* object is created where the result of the asynchronous call will be copied when it finishes. But if the variable was already *dirty* the only work to do is to change the identifier of the routine that is responsible of *cleaning* the variable to the identifier of the new asynchronous function called. The last asynchronous function called thus is now responsible of *cleaning* the variable when the function finishes.

4.3.4 On differences between external and internal asynchronous calls

External call

An external call is produced when a routine belonging to an application process and implemented in the code of the process (not in the ATL module) is called from the ATL code. The execution of this external call is not performed by the Virtual Machine but by the application process itself.

The only work done in this case by the Virtual Machine is the preparation of messages being sent to the process for the call and its parameters and the storing of the returned values in the corresponding places.

The preparation of messages has no difference from the synchronous external call, the difference is only for the WAIT intermediate code instruction which is used in synchronous calls to stop the execution after sending the messages and therefore waiting for the results.

The most important difference in external asynchronous calls is in the mechanism to store the results of the routine into the right place. It is necessary to keep some information about the routine called that must be accessible when the call is finished. This information includes the identifier given to the routine that identifies also the parameters, a list of pointers to variables where the parameters passed by reference should be stored at the return, and also a pointer to the variable where the return value should be stored (if it is the case), and a boolean flag indicating if the call is asynchronous. The variable into which it stores the return value is created from the value of the operand pushed into the temporary stack when the l-value of the assignment is accessed. We will call it the *return site* (place to put the return).

The Virtual Machine keeps a list of objects having this information for each external asynchronous call that is being executed (by the corresponding process) at any time.

When the returning messages arrive at the Virtual Machine communications driver, it looks for the corresponding object at the list of external asynchronous

calls and stores the result (either return or parameter) into the correct place. If there was no place for the return of a function and the Virtual Machine receives this return it will be deleted without using it, because it means the function was used as a sentence (see section 4.3.2).

Checks are also made to see if the variable being stored is a *dirty* variable (it would be at the list of *dirty variable* objects) and in this case it *cleans* the variable if the routine is the one responsible of it. This seems to contradict the "third main idea" listed in subsection 4.3.1). However, if the variable is marked as being owned by another asynchronous call, it means that an assignment to the variable with the result of that call has happened at a later point in time, and in accordance to that "main idea" the second assignment prevails.

Once the variable has been *cleaned* it has to be removed from the list of *dirty variables* and those executions waiting for this variable to be *cleaned* become ready to continue their work. This last part is implemented by appending these executions to a list and starting a loop to execute all of them at the end of this cleanup procedure.

Internal call

An internal call is defined as a call to a routine (procedure or function) implemented in an ATL module. Therefore an internal call will be executed by the Virtual Machine itself. It means the asynchrony in this case is designed as a concurrency of different executions that we will call *ExecSteps* (as the C++ class that stores its state -see section 4.2.3 *The execution*). Although these executions are similar to threads they are not implemented using low level threads and no mechanism is provided to control access to shared memory (variables in the global scope).

The ATLAS Virtual Machine always acts as a concurrent program because any command arriving to it is considered an independent execution (an *ExecStep*). An *ExecStep* is then an execution which is independent from the others except for global variables which are the same for all executions.

When an internal call is asynchronous and called asynchronously it must be considered as a new *ExecStep*, i.e. as an independent concurrent execution.

The general idea for the asynchronous internal call management can be described as follows (figure 4.13 shows the idea by using an example):

- it generates a new *ExecStep* for the execution of the internal call;
- it marks the variables used as actual parameters passed by reference and the one used to assign the return (if it is the case) as *dirty*. They are marked *dirty* in the *calling level scope* but not in the *called local scope* where they (the parameters) must be used;
- it *cleans* this variables when the routine call is finished.

The idea is very similar to the external call but there are a lot of intricacies in this management because the same interpreter code is executing both synchronous and asynchronous calls, therefore some information must be available for each *ExecStep* and also to communicate them in some situations.

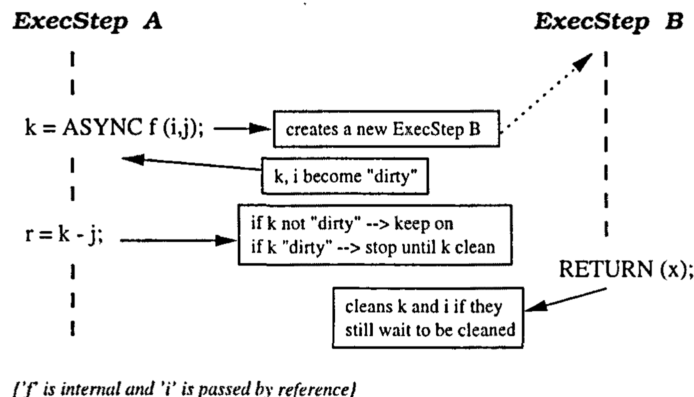


Figure 4.13: An example of asynchronous internal call

The additional information needed in the *ExecStep* to manage asynchronous calls includes: the identifier given to the execution (to the *ExecStep*) in order to be able to recognize it as the execution that should *clean* some *dirty* variables; a boolean indicating if the *ExecStep* belongs to an asynchronous call; and an integer counting the number of nested synchronous calls (a CINT increments it and a RET decrements it).

But besides the *ExecStep* information the Virtual Machine also needs to keep some information about the internal asynchronous call like it did for the external call (see section 4.3.4). We will call this object *OpenedCall* in order to be able to refer to it in the text below. This object is also stored in a list and the information includes the *return site* for the return value if it is required, a pointer to the *ExecStep* that executes this routine, and a boolean flag to coordinate the destruction of this information between the calling and the called routine (which is discussed below).

In order to make this general idea work there are several details to be added to the normal execution of the interpreter of the intermediate code:

- First of all the new *ExecStep* stores a copy of the stack of temporary variables because the actual values for the parameters are in this stack and should be there in order that the code of the routine acts the same way than for synchronous calls. The calling routine eliminates these values from its own stack of temporary variables because it is not going to use them (this is the reason for those POP instructions in figure 4.12).

- The parameters need to know about the variable which has been used as the actual parameter in the calling routine, i.e. they have to know about their origin (see figure 4.14). This is because the management to *clean* and *dirty* this variable acts differently for different scopes (*calling level scopes* and *called local scopes*).

  Upon entering the routine the origin of the parameter becomes *dirty* (in the *calling level scope*), not the parameter itself, because it has to be usable

```
PROCEDURE A (...) IS
  typel varl;
    . . .
  ASYNC B (x, varl, y);  ——►  PROCEDURE B (tl x, typel &v, t2 y);
    . . .
ENDPROCEDURE
```

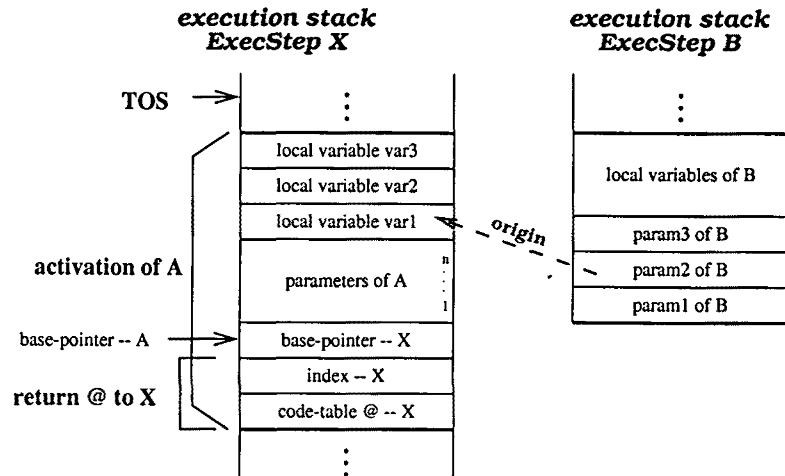


Figure 4.14: Links to the original variables for parameters passed by reference

inside the routine (in the *called local scope*). In the example of figure 4.14, 'varl' is *dirtied* in the *ExecStep* X.

At the end of the routine the parameter (in the *called local scope*) must be deleted, and this is done by interpreting the intermediate code instruction RMV. Since the intermediate code interpreter (in the Virtual Machine) is the same for both synchronous and asynchronous calls, the execution of the RMV instruction will act differently depending on whether the routine is asynchronous or not:

- In case it is an asynchronous call, when the variable to remove is a parameter passed by reference, there are two possible cases that must be treated:
  * the variable in the *called local scope* (inside B in the example of figure 4.14) is *not dirty* ⇒ the origin becomes *clean* if this call is the one responsible of cleaning it (case a in figure 4.15)
  * the variable in the *called local scope* is *dirty* (it has been *dirtied* by another asynchronous call inside this one) ⇒ if the origin is also *dirty* (the most common case because if not it means it has been cleaned by another assignment), the origin variable changes the identifier of the routine responsible of *cleaning* it to the one responsible of *cleaning* the parameter in the *called local scope* (case b in figure 4.15). The *dirty variable* object corresponding to this parameter in the *called local scope* is removed from the list. If the origin is *not dirty* nothing must be done (case c in figure 4.15).

- In case it is a synchronous call and the variable to remove is a parameter passed by reference, if the parameter in the *called local scope* (inside this synchronous call) is *dirty* the origin of this parameter must become *dirty* (case d in figure 4.15), and the *dirty variable* object, created for the parameter marked *dirty* inside the synchronous call, must be changed to point to the origin of the parameter.

If the entity to be removed is *dirty* and it is not a parameter passed by reference but a variable only visible in the local scope (parameters passed by value or local variables), the *dirty variable* object must be removed from the list in both cases, synchronous or asynchronous.

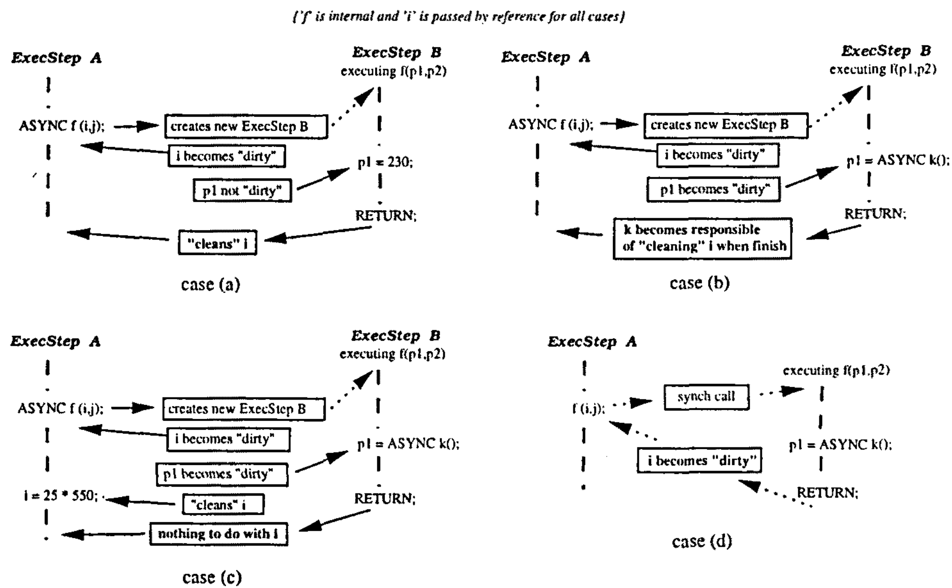


Figure 4.15: Different cases when removing a passed by reference parameter

- Since the instructions order in an assignment of an asynchronous internal call is that the real call is done before the MASS instruction (see figure 4.12), the possibility that the routine called ends before the calling routine has prepared the return place for the result exists. In other words, the *ExecStep* of the internal asynchronous call may end its execution before the calling one executes the MASS instruction. A coordination between the two *ExecSteps* is needed so that the return value is still available until the last one uses the *OpenedCall*, i.e. independently of which one arrives first to use the *OpenedCall* (where the *return site* will be stored), the *OpenedCall* must persist until the second *ExecStep* uses it. The two intermediate code instructions that implement this coordination are: MASS in the calling routine, which decides where the return of the asynchronous function is going to be assigned; END in the called routine (the asynchronous function), which is the last instruction in each *ExecStep* executed by the Virtual Machine (this END thus indicates the end of the

function execution). These two instructions then use a flag kept in the *OpenedCall* object to coordinate with each other:

END: In case of an asynchronous call, if it is a function (it has a return value), it checks if there is a *return site* on the *OpenedCall*: if there is one it means the MASS of the calling routine has been executed before and the return site is the place to put the return value. After this the *OpenedCall* can be removed and the variable waiting for this return must be *cleaned.* If not, it checks if the flag indicating whether the *OpenedCall* can be removed is true and in this case the return value is not needed. It can be deleted and the *OpenedCall* of the asynchronous call can also be removed. If the flag says the *OpenedCall* cannot be removed yet, it means the MASS of the calling routine has not been executed yet, so the routine doesn't know whether the result will be needed or not. In this case the return value is stored in a new variable pointed from the *return site* and the decision is left to the calling routine. In this case it also activates the flag which indicates the *OpenedCall* can be removed in order to communicate the calling routine it has finished, and it doesn't remove the *OpenedCall* object.

MASS: In this case it just checks the flag indicating if the *OpenedCall* can be removed and in case it is true it already has the return value in the *OpenedCall* object. It copies this value to the right place and removes the *OpenedCall.* In case it is false it puts the right place for the return into the *OpenedCall* and activates the flag in order that the END instruction of the asynchronous call does the rest of the work.

- The last aspect related to the asynchronous internal call is for the global treatment to eliminate those active executions that are related to a destroyed process. A process can be destroyed because of an UNUSE or a KILLPROC sentence ordered by the developer. In these cases any execution being done by the Virtual Machine that includes some command defined into the process' ATL module should be aborted. This treatment is done for each intermediate code table that the process defined in its ATL module.

  Of course the elimination of *ExecSteps* in this treatment needs to be also done for synchronous calls. This has been explained in section 4.2.3 –*Other messages from the* ***distr*** *process.*

  Moreover, in the case of asynchronous calls, this elimination has to do the following:

  - For each asynchronous *ExecStep* to be eliminated, if it has some *dirty* variables waiting for it, these variables should be *cleaned.*
  - For each *dirty variable* object it has to check if any *ExecStep* waiting for it to be *cleaned* has to execute an intermediate code table of the process being eliminated, and in this case it must remove this *ExecStep* from the list in the *dirty variable* object.

4.3.5 Accessing to a *dirty* variable

As an l-value

The access to a variable as an l-value is produced when the variable is used as the left side of an assignment. Since the use of assigning to this variable an asynchronous call has been covered in section 4.3.3, now we focus on the assignment of an expression.

As explained in section 4.3.3 the GET intermediate code instruction receives a flag saying whether the variable is being accessed as an l-value or as an r-value. When the flag says it is an l-value and the variable is *dirty*, the GET instruction still does not know if the right side of the assignment will be an expression or an asynchronous call. Therefore it just stores the *dirty variable* object in order that the following move instruction will decide.

In the case of an expression assignment the following move intermediate code instruction will not be a MASS (asynchronous move), therefore the variable will be *cleaned*, the *dirty variable* object removed from the list and those executions waiting for this variable become ready to continue.

This behaviour designed for an expression assignment to a *dirty* variable causes the variable to become *clean* at this point, but it does not assure it will be consistent from this point because the routine having *dirtied* the variable can be still running and it may change the value.

As an r-value

When a variable is accessed as an r-value it means that something should be done with its value. It is used inside an expression.

As has been discussed above, any execution (*ExecStep*) trying to access a *dirty* variable as an r-value will be frozen until the variable is *cleaned*.

In the implementation this is directly done by the GET intermediate code instruction because it is the one actually accessing the variable and knowing also if it is an r-value or not. The action then when this instruction finds out it has to access a *dirty* variable as an r-value is to stop the current execution (the *ExecStep*) and append it to the list of *ExecSteps* waiting for the *cleaning* of this variable into the *dirty variable* object. The execution will be ready to keep on with its work again when the variable is *cleaned*.

In case the access is to a global variable, before stopping the execution, the GET instruction checks if the routine responsible of *cleaning* the variable is itself (see the example in figure 4.16). In this case the execution is not stopped (allowing the access to the variable) in order to avoid this kind of deadlock.

4.3.6 Possible deadlocks dealing with "dirty" variables

It is clear that this mechanism is not fool-proof. In fact it can lead to deadlocks when asynchronous functions or procedures use global variables that happen to

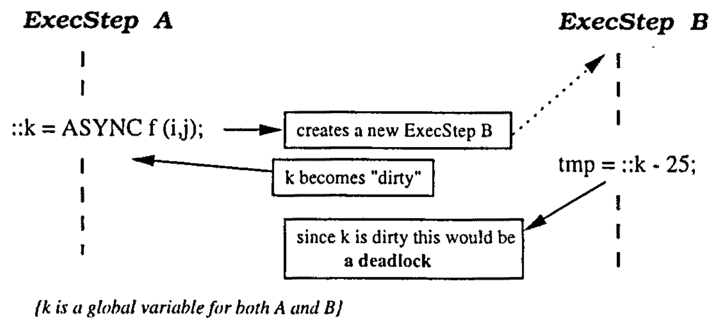


Figure 4.16: Example of an avoided deadlock

become dirty by other asynchronous calls (also using global variables!). Mechanisms to avoid these situations yield costly algorithms, and their usefulness is questionable. At this stage we have thus opted to make ATLAS programmers aware of this and responsible for avoiding this kind of deadlocks, instead of providing automatic watchdogs.

Chapter 5

The Input Subsystem

The ATLAS architecture encourages the developer to think in terms of separate tasks of computing from tasks of data input or user interface. This separation facilitates the reusability of the different modules (a developer may develop his application without taking care of the user interface because he can use another one previously designed, for example).

This separation helps also in the design of the crash recovery system, as ATLAS is (transparently) aware of data pertinent to each process. How `distr` does this is discussed in chapter 8.

Although in ATLAS an input subsystem is a process like the others, and there can be more than one, at least one is needed to give the final user the possibility of entering data or commands to guide the execution of the application. It is the designer's choice also if he wants his application to handle input by itself.

5.1 Generic Input Subsystem

A generic input subsystem is an input subsystem complete enough to be useful for most applications, i.e. an input subsystem allowing to input data of any type in order that the developer does not have to implement another specific one for his application.

The usefulness is clear because a lot of time is spent by application developers designing and implementing a user interface for their applications. Most of the applications nowadays require a non-trivial user interface to work properly.

What would be required then for a generic input subsystem is:

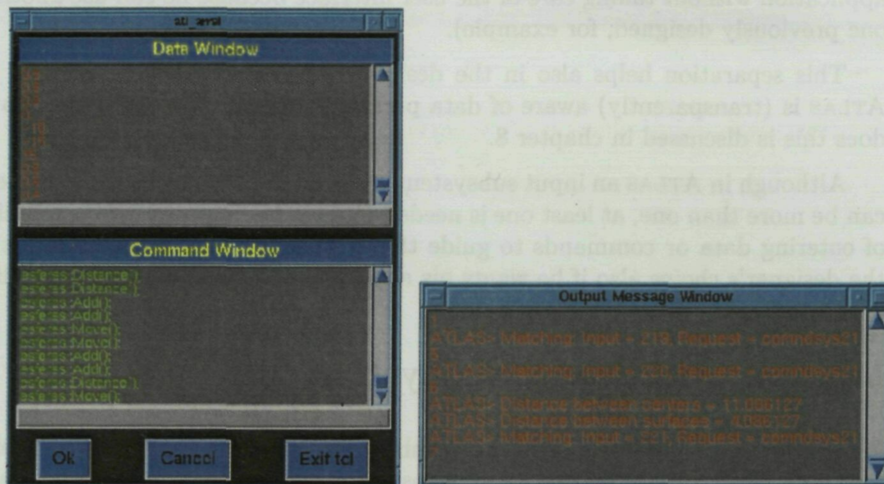
- input data of any type, and transparently if it is possible (without having to say which type it is),
- manage input data from windows created by other processes. In computer graphics applications there usually exist one or more windows where the scene is represented (drawn) and where the final user is allowed to input data by direct manipulation or selection.

But it should also be flexible in order to allow the developer to change some of its behaviours or to extend the interface.

5.2 Design and implementation of the ATLAS Input Subsystem

Presently the Input Subsystem offered with ATLAS provides a window in which all the textual interactions occur (issuing commands or entering numerical data), but can also be instructed to capture events from other windows (owned by the rest of the processes in the application), and it is also an interface between ATLAS and an extended Tcl/Tk [23] engine, so scripts in Tcl can be sent to it to instantiate new interface components.

Furthermore, it provides a small textual window for the output messages of the system. Figure 5.1 shows these two windows that the ATLAS Input Subsystem loads by default (5.1(a) the textual interactions window and 5.1(b) the output messages window).



(a) Input window (b) Output window

Figure 5.1: Snapshot of the Input Subsystem windows

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

5.2.1 Basic input data

The window provided for textual interactions accepts commands and input data of basic types. *Commands will be sent to the Command Subsystem to be*

compiled and executed, and input data of basic types (integer, float or string data) will be sent to the `distr` process to be treated as user input data. Input data of basic types are automatically distinguished from each other because of their format (an integer is a set of digits, a float is a set of digits having also a decimal point and a string is a set of characters).

Moreover the following conventions are adopted:

- If you want to enter a string (chain of characters) containing the representation of an integer or a float, you have to put the special character '#' before the chain you want to enter as a string. Only the first character '#' will be interpreted.
- If you want to enter an integer to be treated as a float, it must have the decimal point.

The textual input window also offers a *small help* if you press the F1 key while the window is active.

5.2.2 X-events control

The ATLAS Input Subsystem can also be instructed to capture events from other windows, owned by other application processes, and consider them input data to be channeled to those processes. This is done by offering a routine that makes possible that given a window identifier and an events mask, the Input Subsystem will receive input events from that window and will produce user input data to the `distr` process with those events.

Since the ATLAS matching mechanism between input data and requests is based on the type name, we need a mechanism to rename the events input data over a window so that these data match only with requests done by the appropriate process. The Input Subsystem keeps information of the processes asking for the events in a window and when an event is produced in that window, it knows which process asked for it.

The Input Subsystem defines special types for the window events and exports them for other processes to use them. These types have the necessary information for each window event type. The rename mechanism uses these type names to prefix them with the name of the process which asked for the corresponding event in that window followed by the character '_'. The process must therefore redefine the type defined by the Input Subsystem prefixing its name with the name of the process followed by '_'. The matching mechanism will then send these events only to that process.

As an example, the Input Subsystem defines the type *Key_pressed_event*. The user process must redefine it in its ATL module:
#deftype ::process-name_*Key_pressed_event se::Key_pressed_event*
(*se* is the name of the ATLAS Input Subsystem module.)

It is also necessary to define this type globally to produce the same type name as that attached by the input subsystem to the event input data messages.

The special types defined by the ATLAS Input Subsystem in its ATL module are:

```
EXPORT #deftype Key_pressed_event
    STRUCT
      window -> integer;
      keycode -> integer;
    ENDSTRUCT

EXPORT #deftype Key_release_event
    STRUCT
      window -> integer;
      keycode -> integer;
    ENDSTRUCT

EXPORT #deftype Button_pressed_event
    STRUCT
      window -> integer;
      pos_x -> integer;
      pos_y -> integer;
      button -> integer;
    ENDSTRUCT

EXPORT #deftype Button_release_event
    STRUCT
      window -> integer;
      pos_x -> integer;
      pos_y -> integer;
      button -> integer;
    ENDSTRUCT

EXPORT #deftype Motion_event
    STRUCT
      window -> integer;
      pos_x -> integer;
      pos_y -> integer;
    ENDSTRUCT
```

and the command offered by the *se* module to give the events control of a window to the Input Subsystem is:

```
FUNCTION X_Control (string opcio, integer window, string mask,
                    string proc)
    RETURNS integer;
```

This command can be called by a process to ask the Input Subsystem to take control over that window of the indicated events (mask). The Input Subsystem will produce input data to ATLAS when these events are produced in that window.

The parameters for this command are:

- string `opcio` ⟶ this string indicates the operation to be done with the specified window control. The possible options are:

- "a" : adds the events control in the mask over the window for this process, if these events control already exists it returns error. *Create the control.*
- "b" : removes the control of all events over the window for this process, the parameter 'mask' is not effective. *Remove the control.*
- "c" : change the events mask, removing the old one and using the new one from this moment, if there was not a control over this window from the process it returns error. *Change the events.*
- "m" : modifies the mask adding to it new events to be controlled over the same window for the same process. *Add new events to control.*

- integer `window` ⟶ this is the identifier of the X window to be controlled.
- string `mask` ⟶ this string contains the name of the events mask to be used (using the X masks), or more than one of these names with the character '|' between them if notification of several kinds of events is desired.
- string `proc` ⟶ this is the name of the process which asks for this control, it is used by the Input Subsystem to construct the type name for the input data.

The function returns an integer indicating if the operation has completed correctly.

These input data generated by the Input Subsystem from X events produced over windows owned by other processes are sent to the `distr` process as *"immediate"* input data, i.e. if a datum of this type is produced but no process has made a request for it (the datum has not been requested) the datum is not considered and it is lost (see section 6.3).

5.2.3 Extended Tcl/Tk

Besides giving the possibility to input basic type data and control X events over windows owned by other processes, the ATLAS Input Subsystem is also extensible. In fact it is also an interface between ATLAS and an extended Tcl/Tk engine, so scripts in Tcl can be sent to it to instantiate new interface components.

The extension made to Tcl/Tk adds some commands to the Tcl/Tk interpreter that produce input data of either a basic type or X events recognized by ATLAS, or commands to be sent to the Command Subsystem.

- Tcl/Tk commands to send basic type data to the system:
  - **SendIntAtl** integer
  - **SendRealAtl** real
  - **SendStringAtl** string
  - **SendCommandAtl** command
  - **SendDataAtl** basic_type_datum
    where 'basic_type_datum' can be any datum of one of the basic types.

- Tcl/Tk commands to send X events of Tcl/Tk windows to the system:
  - **SendXEvents** tcl_window_name mask process_name
    the events produced over the window 'tcl_window_name' will be sent to the process 'process_name' following the X mask 'mask' (only one mask name or more than one with '|' between them). It is also possible to do several calls using different masks.
  - **EndSendXEvents** tcl_window_name process_name
    Since this command is called the process 'process_name' will not receive more events input data from the window 'tcl_window_name'.

  This mechanism with these two commands allows an interface with Tcl/Tk to send X events to a process in the application through the Input Subsystem.

The developer must pay attention to the fact that these Tcl/Tk commands have been created specially for the Tcl/Tk interpreter used by the ATLAS Input Subsystem, therefore they will be only recognized by this interpreter; they will not be valid in another interpreter.

In order to make useful the interface with the Tcl/Tk interpreter offered by our Input Subsystem, it includes in its ATL module some command definitions related to the use of the Tcl/Tk interface. These commands (or procedures) offered are:

- "PROCEDURE **Envia_fitxer** (string file, string dest_file, string proc)"
  Gives the order to execute the Tcl/Tk *script* 'file'. This command also needs another parameter 'dest_file' which is the file name containing the *script* to destroy what was created with 'file' and finally the name of the process ordering this command 'proc', that will own the *script*.
- "PROCEDURE **Envia_ordre_Tk** (string cmd)"
  Allows to execute a Tcl/Tk command interactively, i.e. this command causes the Input Subsystem to send the Tcl/Tk order 'cmd' directly to the Tcl/Tk interpreter.
- "PROCEDURE **Sortida** (string miss, string level)"
  This command uses the Tcl/Tk interpreter to produce error messages. The parameter 'msg' is the message to be shown as an error message and the 'level' parameter indicates if the message must be treated as an **error**, a **warning** or simply as an **output message**.

There is also the possibility to change the textual input window offered by default with another designed by the developer. This window is just a script written in Tcl, so if the developer writes a script defining a window to enter textual data he only has to call the following command also offered by the ATLAS Input Subsystem:

"PROCEDURE **Change_InputWindow** (string s, string d)"

where the two parameters are the names of the files which are scripts written in Tcl for the creation and management of the new desired window (s) and for the destruction of this window (d). This *Change_InputWindow* command destroys the default window and sends the new one to the Tcl interpreter.

5.3 Extension proposal

In order to extend the present Input Subsystem in ATLAS it is necessary to add an engine which allows to input not only basic data but also structured data.

The current Input Subsystem also lacks the possibility of giving *timeout* values to the user input data (see section 6.3). This extension proposes also to include this possibility.

It would be also desirable to offer the possibility of changing each default window (now it is not possible for the output window).

Another important goal would be adding also to the Input Subsystem a menu manager engine to be able to define menus at run time.

Chapter 6

The distr process

The distr process is the ATLAS communications center. Its main role is to control the execution of the application, so it manages the information of each process being executed on the application and the creation and destruction of these processes as well; it takes part on the communications among processes by redirecting incoming messages to the right process; it also manages the input data produced in the application; and is also responsible for the journaling mechanism and its facilities. In this chapter we will explain these tasks in detail except those related to journaling, which is discussed in chapter 8.

6.1 Processes management

The processes management done by distr consists of the following tasks:

- making the decision of where a process must be executed (in which machine on the LAN) and ordering its execution as well;
- controlling the status of each process during the application execution;
- killing a process;
- starting and managing the recovery of a process if it was dead because of a crash on its machine or of its communication with distr.

Each one of these tasks will be described in the following subsections.

6.1.1 Distribution in ATLAS

As ATLAS is a multi-platform environment, supporting features that ease the execution of processes over heterogeneous local area networks, one important feature is transparent execution of processes, which allows a user to run a process in the network by simply knowing its name, regardless of:

– which machine(s) have the desired process available
– which machine is the user connected to

ATLAS selects from the different machines that can execute the program the most suitable one. In fact, although not fully implemented yet, ATLAS does a load balancing of processes among the network. The user's view will be that of a virtual machine he talks to (the ATLAS system) which encapsulates and hides all the complexity, but also offers all the power, of the network.

Global architecture managing the distribution

The ATLAS processes distribution requires a component to be used on a per-host basis, which defines that host as an ATLAS-capable machine. This component is the server (the `server_atlas` process). Servers do not communicate between them, but only with the `distr` processes, which give them orders to execute specific processes for the applications.

Besides the servers and the `distr` process there are also some configuration files which are important to several mechanisms ATLAS uses for the distribution. These are:

AtlasSettings This is a user's configuration file. It must be located in the user's home directory, under the name **$HOME/.AtlasSettings**. This file configures two things, the directories where a process can find shared libraries and the hosts allowed by the user to be used by ATLAS for his applications. We will see its structure in more detail later.

AtlasConfig This is an ATLAS internal configuration file. It should be located in the ATLAS installation directory, under the name **bin/AtlasConfig**. It contains the port numbers used for the communications between `distr` and the servers and optionally the minimum and maximum time to wait for the broadcast responses (see *"The broadcast mechanism"* below). This file must be changed only under special circumstances, and always by the ATLAS system administrator. It is used by both the `server_atlas` and also the `distr` processes.

ATLuserid Each ATLAS user has a unique identifier. This identifier is contained in the file **$HOME/.ATLuserid**, and is a 32-byte hash of random information from the system at the time of creation (essentially 32 random bytes) that ATLAS will use for all user identification. The file is used by both the `server_atlas` and the `distr` processes. It must be generated before the user runs any ATLAS application by using the "`genid`" utility also included in ATLAS. This identifier is only as secure as the file system, but that seems reasonable. Presently ATLAS communications are not encrypted, so this identifier is not protected against network eavesdropping.

The `server_atlas` process is a daemon which should be running in each of the hosts willing to provide ATLAS execution services. Its role is to accept connections from the `distr` process and run the application process requested by it. But it also implements a handshake with `distr` giving it information

about the architecture where it is running and the processes available on this machine. This information will allow the `distr` process to decide dynamically which is the best host to connect to ask for the execution of a certain process.

The `server_atlas` process is connected to the `distr` process by a connectionless protocol (meaning each command is self-contained and there are no command sequences) and it is running with root privileges. The server implements as of today two main functions, a broadcast response and a process execution. We will see them later when discussing the corresponding mechanisms.

The `server_atlas` process is stateless, which means that it does not keep internal data structures permanently. In other words, each command processed by the server is independent, and it implies starting from zero, performing some tasks, and returning to the starting point.

The `distr` work for the distribution task is to manage the broadcast mechanism, to keep the needed information of each process available for the application and to decide the host where a process must be executed depending on availability of the process in the host. How this is done will be seen later.

The ATLAS *"Sysid" to recognize architectures*

To make ATLAS able to find a process across the network and execute it, obviously it is required some setup work and following certain guidelines.

First of all, ATLAS does not scan whole hard disks looking for processes to execute. Whenever a user wants to execute a file with ATLAS, he or she must put it in a special directory. The key items to access this feature are System Identifiers, *Sysid.*

An ATLAS SysId (System Identifier) is an ASCII string used by ATLAS to group together several binary-compatible machines in a local network. This means that all computers sharing a unique SysId must be able to execute the same binary files. For example, two PCs can execute exactly the same files, because they share the same processor architecture. Thus, their SysId should be the same.

The SysId is therefore not a computer identifier, but a "binary architecture" code. It is built automatically by the ATLAS system, and has the following format:

`sysname-release-machine`

where each substring (sysname, release and machine) are derived from the `uname` system call. In fact, writing on a system shell the command `uname -srm`, it will print the three members, and in the desired order. Sysname identifies the family of computers your system belongs to. For example, `"SunOS"` would identify Suns, as `"Linux"` would identify Linux based machines. The second term, release, is used to identify what version of the OS is the system running. The uname call will return a full version number, such as `5.5.1`

As far as ATLAS is concerned, only the major release number is required, so any minor version number or additional information will be stripped away. Finally, the machine identifier is a vendor-dependent code which identifies the

hardware implementation. For example, some Silicon Graphics systems have machine code IP25, and some others have the IP32 code.

There are, however, some observations to be made. First, some machines have sysnames that are mixed-case. "SunOS", for example, would be such case. The ATLAS SysId, following the naming convention of Unix, is case-sensitive.

Second, there are some machines with OS release numbers containing non-numerical information. Some HP machines exhibit this behavior, having release numbers such as B.10.10. In these cases, ATLAS will consider that the major release number is the first number found scanning the release number from left to right. This means that in our example "B" will be dropped and the version number would thus be 10.

Last, but not least, sysnames and machines can contain some special characters, such as slashes, points, or other printable characters. ATLAS strips these characters away from the names, and so machine names such as "9000/735" will be converted to "9000735".

To avoid problems deciding which is the correct ATLAS Sysid for a given architecture, ATLAS comes with a handy utility which will tell you the SysId of any machine. This program is called gensysid, and may be invoked at any time from the shell command line by writing:

```
gensysid <CR>
```

and its output will be a message telling you the desired string identifier.

When the Sysid is used by the developer to identify binary–compatible architecture directories, this ATLAS Sysids can be also wildcarded, thus allowing a unique identifier to be shared by different architectures. This adds flexibility to the system. For example, a Sun machine could have Sysid "SunOS-5-sun4u" which would mean a Sun UltraSparc with Solaris and a different machine could have the Sysid "SunOS-5-sun4m" which would mean an older Sun machine, but also with the Solaris Operating System. Now, there is no need to use two different identifiers, as both machines are binary compatible. Thus, it would be useful to allow Sysids to define which of the three fields (vendor, os, and machine) should be taken into consideration, and which should be simply ignored. The way to achieve this results is by using the wildcard "any" as a substitute of the undesired field. For example, the two Sun machines described earlier in this section had Sysids "SunOS-5-sun4u" and "SunOS-5-sun4m". So, a well-constructed global Sysid for them both would be "SunOS-5-any" as we consider the machine field to be irrelevant.

In case the ATLAS daemon (`server_atlas`) is also working in a not binary compatible machine also working with Solaris (SunOS-5), for example a SunOS-5-X86, the distinction should be done for this machine in order to avoid a process compiled for the others to be sent to this machine.

The broadcast mechanism

Every time a `distr` process is started somewhere in the network, it issues a broadcast message to the network where some servers will be in turn listening

to these kind of messages. This message is thus called Distr Broadcast Message, and has the following structure:

- length of the message (network formatted int)
- user identifier of the user executing ATLAS (net formatted int)
- group identifier of the user executing ATLAS (net formatted int)
- *Sysid* for `distr`. Variable length string
- Execution path for `distr`. Variable length string
- ATLuserid of the user executing ATLAS

This broadcast message tells the servers who (uid and gid) is running the newly-created `distr` process, which machine architecture (Sysid) is the `distr` using, where (absolute pathname) has been the `distr` started, and finally the ATLuserid of the user who started ATLAS.

distr sending the message

When the `distr` process is started up it must send the broadcast message in order to find out the information of the hosts available to execute ATLAS processes. Before sending the message it must do some initializations. These are:

- **Port customization:** ATLAS does not require specific ports to work. In fact, the ATLAS system manager may specify which ports (two are required) will be used throughout the execution by changing the contents of the AtlasConfig file. This file may only be changed by the ATLAS system manager, and specifies vital information for ATLAS. It can be found in the bin directory of your ATLAS distribution. The only lines needed to customize the ports are:

```
$PORTS
stream = 5021
datagram = 5023
```

  Here you can change the values of any two ports (not necessarily contiguous). The only requirement is that both ports must be free.

  This port customization, of course, is also done by each `server_atlas` process when they are started.

- **Process–Host Table initialization:** Before receiving any server response to the broadcast message, the Process–Host Table is created. First the AtlasSettings file is scanned in order to find the allowed host list. This list is a part of the .AtlasSettings file (located in the user's home directory) and is not mandatory. It is useful for those users that want to "ban" hosts from the network. Presently the syntax is:

```
$HOSTS
host-name1:host-name2:...:host-nameN
```

If the ATLAS distr process finds a record like this it will only treat those responses sent by servers running over the hosts in the list. As we said previously, this node of the .AtlasSettings file is optional. So, if not present distr will process all messages from servers that respond to the broadcast message, without further filtering.

After these initializations, the distr process is ready to send the broadcast message to the network. Thus, the message is sent, and also a handler is attached to the "ACE Reactor" object [30] to easily detect the responses coming from the servers. A timer is also added into the Reactor so that we give some minimum and maximum time for the servers to respond. distr then waits until either there are a certain number of hosts offering a Command Subsystem (e.g. 3 –in order to be able to choose a good one) and an Input Subsystem (this process does not require much CPU time) or the minimum time is passed and some hosts have offered a Command Subsystem and an Input Subsystem. In any of these two cases distr will keep on its execution by starting the Command Subsystem process and loading the default ATL module. Otherwise distr keeps waiting until a Command Subsystem and an Input Subsystem are offered or until the maximum time registered to the "ACE Reactor" has passed. In this last case distr will stop its execution notifying the user none of the hosts answering has offered the minimum processes to work.

The minimum and maximum time to wait can be customized by the ATLAS system manager by changing a parameter in the AtlasConfig file. The node to be changed is:

```
$BROADCAST
timeout_min = 2
timeout_max = 5
```

and the substitution must be on the number of seconds of both timeouts. If the default 1-second and 3-second delay are enough, there is no need to have this node on the AtlasConfig file.

To sum up, the distr process generates the broadcast message, sends it, and waits for responses until it can start the ATLAS execution.

Response construction in server_atlas

We can see the broadcast processing divided in four steps:

1. **Message reception.** Servers keep two sockets open throughout their execution cycle. One of them is a datagram socket, and is used only to process broadcasts and send responses to them. The read function (BCastRcv::ReceiveBCast) peeks at the socket when it is notified that there is data available to read, and then consumes it. Then, the different fields of the message are extracted (function BCastRcv::DecodeBCast).

2. **User authentication.** Servers must be available all the time to process requests (either other broadcasts or execution requests). So, it is mandatory that steps two (user authentication) three (response generation) and

four (response emission) do not block the server. Thus, a sub-process is created, and all processing is performed concurrently by the main and secondary threads: one taking care of the sockets and waiting for other requests, and the other performing steps 2, 3 and 4 of the broadcast processing. After message reception, comes the second part, which is the user authentication. ATLAS attempts to prevent situations such as:

- faked identity by the poster of the message
- non-registered user

by performing simple verifications. These verifications consist of:

- First, the broadcast message contains the uid and gid of the user. It also contains an ATLuserid code, which is filled by the `distr` process with the code for that user. An illegal request could probably fake the uid and gid of a certain user, but faking also the ATLuserid would be more difficult, as this information is contained in a file readable only by its owner.
- Second, when the request arrives (with a uid, gid and ATLuserid) the server will access the information on the ATLuserid for the received uid and gid, and check if that ATLuserid is the same we were sent through the socket. The way to do this is to access the `/etc/passwd` file with the command `getpwuid`. This file contains (among other information) the home directory for every user. So, given the uid and gid we can access his home directory, perform a setgid and setuid and read his .ATLuserid file. This way we perform the verification of all data sent being correct.

This provides user authentication as secure as the user's file system and network. The network security could be improved using cryptographic protocols, which is not done at present. It seems however unnecessary to provide stronger security than the user's file system under any circumstances.

Once this step is completed, the server knows the request is valid, and is thus ready for the central step, which is the response generation.

3. **Response generation.** The response of a broadcast message is a list containing the following information:

   - Host name (server which generated the response)
   - Server's Sysid
   - Normalized load coefficient
   - Available process list

   The first two fields are straightforward: the `distr` process needs to know who is answering the broadcast message and what architecture does the responder have.

   The normalized load coefficient is calculated by dividing the maximum of the two more recent load averages (from the responses given by the operating system) over a capacity level given for this host. This capacity level can be configured by the ATLAS system manager at installation time and is an integer higher than 0 (higher number, higher capacity). There is also a default value for it (1).

This normalized load coefficient is, of course, an heuristic solution and must be extensively tested in order to achieve the maximum efficiency.

The fourth element of the response of a broadcast message is a list containing all the files that the responding server found on its directory scan. We will now explain the exact contents of this list, and its creation algorithm.

Whenever a server replies to a broadcast message, it scans certain areas of the file system searching for executable files. Once done, it returns that file list to the `distr` process. This way when the user asks `distr` to execute a file, it is easy to check which hosts have that process available for execution.

This list is priority-ordered. This is useful whenever a process in available in two different places of the disk. For example, with the following structure:

```
$HOME
     Atlas
            bin
                 SunOS-5-any
                      process1
                 IRIX-6-any
                      process2
                 any
                      process1
             (...)
     (...)
```

we have the process "`process1`" available at two different locations: the specific SunOS-5-any and a generic one located at "any". Logically, the first instance should be prioritized, as it refers to a more specific location. Thus, in the executable file list which will be returned to the `distr` process the process `$HOME/Atlas/bin/SunOS-5-any/process1` should be before the same name process located at "any". The way to build such a list is scanning the selected areas of the disk in importance order: most important ones first, secondary after. So, the algorithm for scanning the disk is:

- Scan the area of the disk around the path where `distr` has been started. If that path is inside an architecture directory (such as the path `$HOME/process/bin/SunOS-5-any`), we will first locate the directory corresponding with the server's Sysid, and then scan it. After we have scanned that directory, we will scan the several wildcarded versions (first wildcarding machine, then OS release, and finally sysname). Finally, we will scan the "any" directory and also the base directory. So, the order in which directories are checked is: (assuming the Sysid is in the form S-R-M)

```
i)   ..../S-R-M
ii)  ..../S-R-any
iii) ..../S-any-M
```

```
iv)  ..../S-any-any
v)   ..../any
vi)  ..../
```

- Scan the `$HOME/Atlas/bin` area, including the architecture directories for the server. Again, the order of checking is:

```
i)   $HOME/Atlas/bin/S-R-M
ii)  $HOME/Atlas/bin/S-R-any
iii) $HOME/Atlas/bin/S-any-M
iv)  $HOME/Atlas/bin/S-any-any
v)   $HOME/Atlas/bin/any
vi)  $HOME/Atlas/bin/
```

  Note how we scan this `Atlas/bin` section after we scanned the region around the `distr` boot path. This behavior is intentional. As we said in preceding sections, placing executable files in the same path where we start the `distr` process is a convenient way of working with ATLAS while developing and testing new code. Thus, in case we have the same process in the starting path and in one of the `$HOME/Atlas/bin` sections, we wish to use the first one as the preferred.

- Third and last, we will scan the installation directories of ATLAS. These directories should contain only binaries related to the ATLAS distribution, such as `server_atlas`, `distr`, and several utilities. So, it makes sense to make this the last (less priority) location for executable files. Here again we will scan directories using the Sysid wildcarding feature as described in the two preceding steps.

For every file and directory we encounter during this phase, some tests are made to ensure proper execution. Then, for every FILE, it is considered executable if, and only if:

a) It is a file and has execution permission by its owner, or

b) It is a symlink, and the file linked by it satisfies **a)** step.

And for every DIRECTORY, it will be expanded if, and only if:

a) The process of scanning determines that directory as a possible location of files (a matching architecture directory, "any", etc.), and

b) The user has read and execution rights over that directory.

4. **Response emission.** Once the scanning process is complete, we have a list of files, along with the Sysid and domain name address of the server which performed the scanning. Thus, it is now time to format this list into a network packet, and send it through the network to the `distr` process. So, we build a message with the structure:

address + own Sysid + normalized load coefficient + 1{process name}N

(where i{x}j means repeat {x} from i to j)

> Albeit redundant, the message is preceded by a network–formatted integer value which contains the total length (this is, the four bytes for the integer and all the message data) of the message which is used by the driver to simplify the processing of datagrams.

Whenever the `distr` process receives a server response, it wakes up (specifically, the routine `BCast::handle_input` is woken up by the Reactor). This routine is fairly simple. First, it decodes the broadcast response which is received through the datagram socket and then the message is passed to the PHT (Process–Host Table), where it will be divided into its main elements, and inserted into the structure (if and only if the host that responded is allowed by the AtlasSettings file).

This broadcast messages will be sent periodically by `distr` during the application execution in order to update the PHT information. The user may decide to disallow these periodical broadcast messages. In this case the information obtained with the first broadcast will be fixed for the duration of the execution.

Process–Host Table

During normal `distr` operation, one of the main tasks to be carried out is to decide, each time the user wants to execute a process, which server is best suited (in terms of process availability and also performance) to execute it. The `distr` process has a data structure specially designed to aid in this complex decision, the Process–Host Table (PHT). The PHT is a variation of a multi-list, as can be seen in the following scheme:

Host 1	X	X	X
Host 2		X	X
	Process 1	Process 2	Process 3

The above drawing tells us that Host 1 has Processes 1, 2 and 3 available, and that Host 2 only has processes 2 and 3. The above drawing would depict the classical multi-list as found in countless data structures and algorithms textbooks. Our PHT works in a similar way, but adding priorities. In sections *"The* ATLAS *Sysid to recognize architectures"* and *"Response construction in* ***server_atlas****"* we said users are allowed to store their binary files in different locations, and that ATLAS will scan the disk in order so more priority areas are searched before less relevant ones. So, we will use the following data structure, which is just a variation of the one explained previously:

Host 1	1{path}N	1{path}M	1{path}N
Host 2		1{path}O	1{path}P
	Process 1	Process 2	Process 3

Here, each Host–Process node keeps a list of all instances of the process in the host's file system. The list is priority–sorted, so the first element has the most priority, and so on.

This data structure (HostProc) has two main operations: inserting a new list of processes available in a particular host, and finding the most suitable host for executing a process.

The insertion routine has the disadvantage (due in part to the nature of the broadcast system) that it does not insert host–process pairs, but lists in the form:

hostname sysid normalized-load process0 process1 process2 ...

Process(i) are the processes available at that host. So, our routine reads the first three elements in the list (host name, Sysid and normalized load coefficient), builds the Host Information block (named "InfoHost"), and thus opens a new "row" in our multi-list, which is really a hash table to speed-up access. Now, for every process in the list, it appends it to the corresponding list. As the initial list is priority–sorted, we can ensure that the lists created will also be sorted.

A process execution

Deciding which host

Once the broadcast interval has ended, the PHT now has information about the processes and hosts available. It is time then to start the execution of ATLAS. The `distr` process has to start then the Command Subsystem process and send to it the ATL file having the initialization of the application.

Whenever a certain process must be executed the `distr` process uses the information contained in the PHT to locate the best server to execute this process, and will then try to communicate with it. The core routine here is `QuinHost` which, given a process name, scans the PHT to find the best possible host.

While the *insertion routine* (in the PHT) is row-wise, or horizontal, the *selection routine* (used by `QuinHost`) will be vertical, or column-wise. What we will do is scan from top to bottom each host in the host list, and keep track of the last normalized load coefficient at each one. Then, for each host, we seek the process we want to execute, selecting it from the first position of the corresponding list. Now, as we scan top to bottom, we will select a new host as the executor if and only if it has less current normalized load coefficient, but has the desired process available.

It is important to note some behaviors which may seem strange at first sight, but are fully normal:

For example, two hosts (say H1 and H2) have the desired process, and H1 has less current normalized load coefficient (thus making it a best candidate than H2). Then, H1 will be chosen as the executor, regardless of where the process resides. For example, if H1 is best-suited, but has a process in a low-priority location, but H2 is a bit more occupied, but has a high-priority location process, H1 will be chosen. So, host choice is considered more important than location choice.

The actual execution

To execute a process, `distr`, having decided which is the best host to use, generates a message to be sent to the server on that host, requesting the execution of the process.

This message includes information about the environment that must be set to execute the process. But this information is the same for every process during the application execution, so `distr` stores it during its initialization.

The environment information is a list which includes the following information:

- User identifier (result of a `getuid` call)
- Group identifier (result of a `getgid` call
- ATLuserid value (from the .ATLuserid file)
- DISPLAY environment variable
- HOME environment variable
- PATH environment variable
- PWD environment variable
- ATLAS_ROOT environment variable (only if it is defined)
- LD_LIBRARY_PATH environment variable (only if defined)

All this information is kept throughout the execution cycle.

The message sent to the server then has the fields corresponding to the list above having the ATLAS_ROOT and the LD_LIBRARY_PATH as optional.

We will now focus on the events following the reception of this execution message by a given server, explaining the algorithm and tests it performs. This function can be subdivided in the following steps:

- **Message reception.** First of all, the server's "MessRcv" object is notified (via the `handle_input` method of the ACE library [31]) of the availability of data in the stream socket. Datagram sockets are used for broadcasting purposes only, and stream sockets are used for execution messages.

  Fields in the ATLAS process execution message can be optional and there is no order in the sequencing of the fields. As long as the message contains the required information, this data can be ordered in many different ways.

  Once the message is read into a list of buffers, we reconstruct the several fields and create a list of parameters.

- **User authentication.** Before processing an execution request we must be sure that the sender of the message is really allowed to execute the process, and that no security violations can arise. So, we re-check the ATLuserid using the `/etc/passwd` file exactly the same way we explained in the broadcast section (*"Response construction in* `server_atlas`).

- **Environment setting.** Before executing the process, we must set its environment. The reason is simple. Remember we said the server process starts when the machine boots, and is always under execution as a daemon. So, it has the environment set at boot time. To ensure security, the process spawns a child process whenever operations are needed, thus keeping always an eye on the sockets. This child then performs a setgid and a setuid command, releasing its root privileges. Still, it lacks environment, which should be set in a per-command basis. Every time we want to execute a process, we will set some environment variables (PATH,

LD_LIBRARY_PATH, etc.) accordingly, and then execute the process with the `exec` command. This way we ensure that processes are executed within their right environment and as user processes.

The environment variables that are set are (in the following order):

First, we set the variables received within the execution message.

Second, we set the ATL_LOCAL_EXECUTION_PATH with the path to the executable file. As ATLAS executes processes with the `exec` call and it wants that the PWD environment variable sent by `distr` be inherited, the child process will not have access to the path where its binary file resides. So ATLAS must allow for an alternate mechanism to let child processes access their directory structure. This is particularly useful whenever our binary file must access some data files in its installation tree (see also subsection *"How a process knows its relative path"*).

Third, we set LD_LIBRARY_PATH adding to its value the libraries specified in the ".AtlasSettings" file. This file (located in the home directory of every ATLAS user) contains which library paths should be considered when executing under any of the available platforms. For example, a file containing:

```
$LIBRARIES
SunOS-5-sun4u = /homes/husers10/atlas/danis/Atlas/Atlas/\
   lib/Sun5: \
   /usr/usuaris/sig/mfairen/proves_Atlas/ACE-4.5/\
   ACE_wrappers/build-Sun5/ace/:\
   /usr/local/tcl-tk8.0/Solaris/lib:  \
   /homes/hsoftsol2/Atlas/ACE-4.4/ACE_wrappers/ace/ \
   /usr/usuaris/sig/mdtl/nq/libbonsai/:\
   /usr/local/mesa2.4/SOLARIS/lib:\
   /usr/usuaris/sig/newdmi/nq/oscarsan/newdmi/lib

IRIX-6-IP25=/usr/usuaris/sig/mfairen/vonsai/Atlas/\
   build-IRIX6/lib:/homes/hsoftsol2/Atlas/ACE-4.4  \
   /ACE_wrappers/build-IRIX6/ace/

SunOS-5-sun4m=/usr/usuaris/sig/danis/atlas/Atlas/Atlas/\
   lib/Sun5:  \
   /usr/local/tcl-tk8.0/Solaris/lib:/homes/hsoftsol2/  \
   Atlas/ACE-4.4/ACE_wrappers/ace/

HPUX-10-9000735 = /usr/usuaris/sig/danis/atlas/Atlas/\
   Atlas/lib/HP10:  \
   /usr/local/tcl-8.0/lib:/usr/local/tk-8.0/lib
```

tells ATLAS that hosts with the Sysid `"SunOS-5-sun4m"` should check the following directories for libraries:

```
   /usr/usuaris/sig/danis/atlas/Atlas/Atlas/lib/Sun5
   /usr/local/tcl-tk8.0/Solaris/lib
   /homes/hsoftsol2/Atlas/ACE-4.4/ACE_wrappers/ace
```

So, these three entries will be prepended (added at the beginning) of the LD_LIBRARY_PATH we received through the network. As processes executed by calling `exec` inherit the environment of the caller, we can be sure that the process being executed will be able to access its libraries correctly.

- **Sub-process execution.** Once all three steps have been completed, and no error has occurred, it is finally time to execute the binary file. ATLAS will only perform one final step: it will redirect the process' standard error and output channels to two files. This is only for logging and maintenance purposes. These two files are called (if the process is called "A"): "A.err" and "A.out", and will be in the same path the `distr` process has been started. As interaction with the ATLAS environment should be through the built-in graphical interface, these files are mainly used as log or debug info files of the process "A".

  After these have been opened, ATLAS will only execute the process with a call to `exec`, passing the following command line arguments into argv:

  `argv[0]`: full name (absolute path included) of the process

  `argv[1]`: process name (executable file name)

  `argv[2]`: OOB_Hack info. Used to prevent the Out_Of_Band Data error in Solaris 2.5 [1].

  And the execution will then overwrite the server's child process logical space with that of the desired process, and begin its execution.

Errors, in the `execv` system call or in the communication from `distr` to the chosen server, should be bypassed by `distr` trying to start the process in the same host with a lower priority version or in another host depending on the error. This error management is not included in the current prototype, but it is briefly described in the miscellaneous section of the extensions chapter (10.5).

What the ATLAS user must be aware of

In order to group in this subsection all information the user must know, some aspects we have already explained before are also briefly mentioned here.

Directories structure

As said in section *"The ATLAS Sysid to recognize architectures"*, Sysids are used to refer to compatible machine groups. So, two binaries are compatible if they were generated in machines which share a common Sysid. The idea then is to organize binaries in a directory structure, using Sysids as their names. Sun machines will keep their binaries in a directory named with their Sysid, for example. This will prevent architectures from trying to execute code other than their own.

[1] In Solaris 2.5 the `select` call does not wake up with only a 1 byte OOB data, so we adapt ATLAS communications to send in this case 2 bytes with the OOB data, this problem has been corrected in 2.7 (we have not checked 2.6).

So, it seems now clear that users need to follow some directories structure guidelines in order to use the ATLAS multi-platform feature. To begin with, they should provide a directory called `Atlas` in their account's HOME. This directory can be used to store everything related to the ATLAS system. Then, this directory must have a subdirectory called bin, which will be the storage for ATLAS binaries.

This `Atlas/bin` directory can be organized in turn in a multi-platform structure, to allow the storage of binaries for several architectures. For example, a possible directory layout would be:

```
$HOME
       Atlas
              bin
                   SunOS-5-any
                   IRIX-6-any
                   any
         (...)
```

This structure would provide a place to store Solaris (Sun 5) files, and a different place to keep IRIX 6 files. These two folders do not need to check the machine member of the Sysid, as all possible variations can be considered compatible, so this part can be substituted by the wildcard "any".

Finally, the user has a directory called any, which will be useful to store generic binary files; shell scripts, for example.

The above explanation may suffice for most applications. Some situations, however, may require a higher level of customization. For such uses ATLAS provides an extended mechanism, which allows you to keep your executables in different places of your disk. For example, you may have a "stable" version of an executable file under a certain path, and a different copy (for example, a debug version) somewhere else. The guidelines to follow in this case are:

- Place the stable binary files under `$HOME/Atlas/bin`, as explained above
- Then, place yourself in a directory of your choice, and place binaries there. You may do so directly, or using a directory structure such as the one found around the `Atlas/bin` area. Now, start the `distr` process from this same directory. If you do so, `distr` will scan as the most-priority area the path you placed these new binaries in. For example:

```
$HOME
       Atlas
              bin
                   SunOS-5-any
                        proc1
       debug
              SunOS-5-any
                   proc1
              IRIX-6-any
      (...)
```

With the file structure described above, you can execute `distr` from the path `$HOME/debug` (this pathname is arbitrary). If you do so, the first executables it will scan for are the ones placed in that path in the corresponding architecture directory. So, if you ask ATLAS to execute the file `proc1`, it will always take as a first guess the file `$HOME/debug/SunOS-5-any/proc1`, assuming it has more priority than the version stored at the directories tree `$HOME/Atlas/bin/...` This criterion is intentional: `Atlas/bin` is a place to store stable files, whereas placing them in the `distr` initial path is a useful mechanism for debugging. However, if you still wish to give higher priority to the file stored at `Atlas/bin`, you just have to execute `distr` from a different path than the one containing the debug version.

The ".AtlasSettings" file

As we have already seen in subsection *"The actual execution"*, using only one definition of the environment variable LD_LIBRARY_PATH is not enough because we usualy will need different definitions for different architectures.

The ".AtlasSettings" file should have the directories to be added to the default value of the variable for each architecture. The syntax required has been also described in *"The actual execution"* subsection. The file must be placed in the `$HOME` directory and configured by the ATLAS application developer.

A set of host names that will act as a filter to avoid using in the application other hosts (also able to run ATLAS processes) can also be included in the ".AtlasSettings" file. The syntax for this (non mandatory) section of the file has been described in ***"The broadcast mechanism"*** subsection.

How a process knows its relative path

Also in subsection *"The actual execution"* we have talked about an environment variable giving to the process being executed information about the path where its binary code is.

In order that the developer can use easily this information, ATLAS offers an API call which has the prototype

```
String atl_get_local_path ();
```

that gives a String which contains the absolute path for the directory containing the binary code for the executable process. Since ATLAS is able to decide depending on the architecture which directories look to search executable files, these executable files can be placed on directories different from the *current working directory*, then the process can access to relative paths by prefixing this relative path with the result of the `atl_get_local_path` routine.

Debugging ATLAS *proceses*

One of the mechanisms used by ATLAS to offer *fault-tolerance* requires processes to regularly send messages to `distr` in order to notify their state. A process stopped in a debugger will certainly stop sending this *heartbeat* messages for a while, thus making the ATLAS `distr` process believe it is malfunctioning

(see more about the *heartbeat* message in subsection 6.1.2). In normally executed processes this would force `distr` to kill the user process, but it should not be applied to a process being debugged.

ATLAS introduces a mechanism which allows process debugging in a convenient way. A process being run can be entered in "debug state" at any time with a user command. From that moment on, the ATLAS `distr` process will keep in mind this situation, thus allowing the process to stop sending *heartbeat* messages. This way, the user can then attach a debugger to the involved program, and behave as if ATLAS really was not there, debugging normally. Once done, the process can be put again in "normal running mode". This restores its state, thus re-establishing the *heartbeat* message protocol.

The two commands involved in establishing and ending debugging sessions within ATLAS are:

```
atl_init_debug (string process_name)
atl_end_debug (string process_name)
```

They require the process name to be an already running process. Thus, to debug a process called "TestProc" with ATLAS, first, you would start the process with the command:

```
USE TestProc;
```

Second, you would start the debugging session by entering:

```
atl_init_debug ("TestProc");
```

Since ATLAS is a multiplatform environment, a user process can be executed anywhere in your network, and you will not necessarily know where. The USE command will make ATLAS decide where to execute your process depending on the workload of each machine, and the availability of the given process across the network. So, when you start debugging your process, you will need the information about the host that is actually running your program, in order to attach a debugger to it. This information will be printed out for your convenience by the `atl_init_debug` command. This way you can find where you should place your debugger to work.

Once ATLAS knows you want to debug a certain process (by means of the `init_debug` call), it is time to effectively debug it. Here we will use gdb as the debugger, but you may use the debugger of your choice to accomplish this task. Remember you need to execute gdb at the host currently executing your process. In our example, we would do:

```
gdb
attach [process_identifier]
```

You need to know the pid of the process to debug. To find it, use for example the ps shell command at the machine executing the process (you know which machine it is because `atl_init_debug` gives you this information). Once

you enter the attach command, your process is fully debuggable. You may set breakpoints, examine data, and perform step-by-step execution.

Once you have finished, you should restore the process state back to normal. This means you have to first detach the debugger from the process. Do this by entering:

```
detach
```

After doing this, you just need to issue the atl_end_debug command to end your ATLAS debugging session, and revert the process' state to normal. Do this by simply typing:

```
atl_end_debug([process_name]);
```

as an ATLAS command, and this will end the debugging session.

6.1.2 The *heartbeat* mechanism

The control of the execution status is managed by the *heartbeat* mechanism. This mechanism makes every process being executed in the application send a short message periodically to the `distr` process giving the required information to control the global status of the execution. The *heartbeat* mechanism is very useful to detect if a process or the communication with it fails, therefore it is also the first component of the *fault-tolerance* mechanism which uses the information kept by the journaling mechanism to recover the status of that process when it needs to be recovered.

Although this detection is not completely reliable (we can call it *suspicion*), it has a high level of reliability. The messages are sent through a reliable connection (we use BSD stream sockets), and the periodicity of sending the heartbeat message from a process is driven by the system timer (SIGALRM interruption). So, unless the developer changes the ATLAS generated code that controls this mechanism, if `distr` decides a process is death, it must be that it is either:

a) death

b) unreachable (network failure)

c) locked in an uninterruptable section for too long a period

6.1.3 Killing a process

The killing of a process can be provoked by:

- the lack of communication with it, in this case the `distr` process decides whether it has to start the recovering or not (see chapter 8);
- an explicit order from the user which can be:

- to finish the process execution normally (via "UNUSE"), it says to the process it has to finish its execution and it does everything that it has to before exiting;
- to kill the process at the moment (via "KILLPROC"), it kills the process directly because it is supposed not working well.

6.1.4 Recovering a process

The *heartbeat* mechanism (section 6.1.2) is directly related to the recovering of a process when its communication with `distr` is not working. A process recovering is totally managed by `distr` but is directly related to the journaling mechanism, so it will be extensively explained in chapter 8.

6.2 Communications management

As it can be seen in figure 3.1(b), each process in an ATLAS application is directly connected to the `distr` process. So any (or almost any) communication between two processes passes through `distr`. This is a desirable characteristic in ATLAS because it favours the maintenance of the journal and allows all of its facilities (see chapter 8).

The work of `distr`, being in the middle of the communications path, is just to direct messages to the right destination, so the commands are sent to the `Command Subsystem`, the error or output messages are sent to the `Input Subsystem`, the routine call and parameter messages are sent to the process owning the routine, etc. Therefore the `distr` work (from the communications management point of view) is just to look into each arriving message to know what is its kind and act in consequence.

6.3 Input data management

ATLAS thinks in terms of a separation between computing processes and data input processes (as has been already introduced in chapter 5). This causes that the input of data and the requests for these data are usually produced by different processes being connected to `distr`. This idea of separation is not new. Plenty of references can be found in the literature since the late 70's (see [32] for a survey and references therein). In fact, since the late 80's this separation has been settled and the majority of works in the future trend are more devoted to finding ways or tools to easily construct user interfaces (which are not necessarily depending on the application subject).

Another work also following this idea of separation is the ADV (Abstract Data Views) proposal [2], where an ADV can be seen as an interface associated to an object (also called ADO). An ADO has no interaction with its outside except for invocations coming from an ADV or another ADO. This dissociation,

however, is not compulsory in ATLAS. Some modules require very special interface elements that are usually incorporated into the same process –e.g. virtual reality applications–.

The *asynchronous matching* functionality assigns input data to data requests asynchronously depending on their type. This functionality then is the base of the independence between input subsystems and the other processes and this allows the re-usability of data input subsystems because an input subsystem may be used by different applications, and the developer can implement his application without having to implement also the user interface because he can use another one already implemented.

The *asynchronous matching* consists of keeping the information of both input data and data requests when they arrive to `distr`. Input data will be stored into a FIFO list and data requests into a LIFO list, so in case of equal type name between data and request, the last request is going to be the first to be served with the data coming, but the first input data is the one to serve the request coming. Figure 6.1 shows the scheme of the management that `distr` does. The input subsystems send input data asynchronously to `distr` and processes needing these data send data requests asynchronously as well. The *matching* then acts distributing the data among the requests in the correct way (depending on their type).

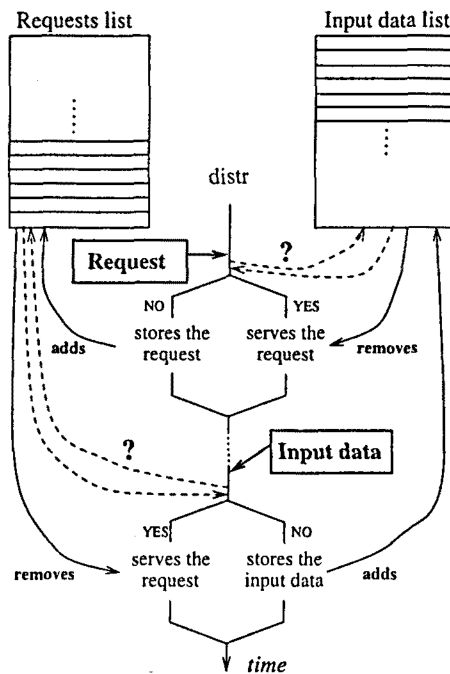


Figure 6.1: Scheme of the *asynchronous matching* functionality

When a data request arrives, `distr` looks at the input data list if there is an input data to serve this request. If there is one it serves the request but if not the request is added into the first position of the list waiting for an input

data for it. When an input data arrives something similar is done, `distr` looks at the requests list if there is a request to be served by these data. If there is one the data serves this request but if not the data is added to the list waiting for a request asking for it.

This functionality is also flexible because assigning *"timeouts"* to input data and requests different interaction modes can be defined. These are:

input data and requests having a timeout greater than 0 ($timeout > 0$)	$\Rightarrow$	it gives expire time for them (the timeout amount of seconds)
input data and requests having a timeout equal to 0 ($timeout = 0$)	$\Rightarrow$	they are immediate, it means if the *matching* cannot be done immediately (when they arrive) they are discarded
input data and requests having a timeout equal to -1 ($timeout = -1$)	$\Rightarrow$	they do not have expire time (they can stay permanently)
requests having a timeout equal to -2 ($timeout = -2$)	$\Rightarrow$	they have an automatic reentrant behaviour, i.e. when they are served they are automatically added again to the list by `distr`.

With these timeout possibilities, the description of how the *asynchronous matching* works explained above must change a little. If the arriving input data or request has a timeout = 0 and there is no partner for the immediate matching it has to be rejected, so it will not be added to the list. Moreover two more treatments have been added to manage these timeouts:

- The timer management started periodically to control the timeout of processes for the *heartbeat* mechanism is also used to control the timeout of input data and requests being stored into the structure. At each timeout, ATLAS will look at the lists (both input data and requests lists) and remove from them any object overtaking its timeout.
- When a request is served and it had a timeout = −2 it is removed from the list but added again automatically, because it is a reentrant request.

Null response

The timeout feature makes it possible for a request to be rejected without doing matching with any input data. Thus no input data is given to this request. But the process that produced the request may be waiting for the response, so a response must be given to the process. This response will be a null response which is a response message (see chapter 7) but without containing input data. The process waiting for this response is then notified and is responsible of dealing with this null response.

Reordering possibility

The order given to the lists (both the input data and the request) is giving priority to the first input data and to the last request received. These priorities introduce some rigidity for the usefulness of the system for a final user who may be interested in giving the data in a different order.

To be more flexible in this sense, ATLAS offers the possibility of changing the order of these lists by selecting one of the objects on a list to be the first of this list from this moment on, i.e. giving the identifier of a request, for example, and calling a specific routine offered by ATLAS this request is changed in the list order to be the first in its list.

To facilitate the use of this order changing routines and to give also flexibility to this functionality in other aspects (adding the possibility of eliminating an input data from the list, for example), ATLAS offers two utility processes that facilitate these tasks to the final user. These processes are `demandes` and `entrades` and will be explained in chapter 9.

6.4 The *events* mechanism

At run time, the ATLAS kernel keeps some structures containing information about the status of the application (which processes are in execution, if there is some request waiting for an input data, etc). To give the opportunity that an application process be informed about changes in this internal structures, ATLAS offers the ATLAS *events mechanism* that allows the process to ask for a subscription to a particular ATLAS event (a list of ATLAS events below). Whenever an ATLAS event is produced the `distr` process sends the corresponding event message to every process subscribed to that event, and the driver of the process, when it receives this event message, calls the callback routine attached to this event at subscription time.

Here is the enumeration of these ATLAS events and their meaning:

ADD_PROCESS: produced when a process is started, its message contains the name of the process;

SUB_PROCESS: produced when a process has been killed and it is not going to be recovered, its message contains the name of the process;

ADD_INPUT: produced when an input data arrives to `distr`, its message contains the input data identifier and its type name;

SUB_INPUT: produced when an input data has been removed from the `distr` structures, its message contains the input data identifier;

MOVE_INPUT: produced when an input data has been moved to the top of the list, its message contains the input data identifier;

ADD_DEMAND: produced when a data request arrives to `distr`, its message contains the request object;

SUB_DEMAND: produced when a data request has been removed from the `distr` structures, its message contains the data request identifier;

MOVE_DEMAND: produced when a data request has been moved to the top of the list, its message contains the data request identifier;

IMALIVE: produced when a *heartbeat* message arrives to `distr`, its message contains the name of the process sending the message to `distr`.

6.5 Journaling management

Almost all information passing through the `distr` process must be stored into the journal in order to be able to recover processes, re-execute an application, offer UNDO and REDO, and perform other journaling functions. Therefore `distr` is the process charged on manage the whole journaling mechanism. This mechanism will be the subject of chapter 8.

Chapter 7

Communications and drivers

A communication can be described as the passing of messages through a communications channel from one process to another and backwards. A communication then implies two end points, both of them sending and receiving messages.

In the ATLAS architecture almost all of these communications have at one end point the `distr` process and at the other another process which can be the `Command Subsystem`, an input subsystem or any other application process. All of these processes then need a communications driver to be able to manage their communications with `distr`.

This chapter describes everything in ATLAS related to communications: the set of messages used, the most relevant communications protocols, the management to pass data through the network and also the way in which the communications driver for an application process is generated automatically by ATLAS.

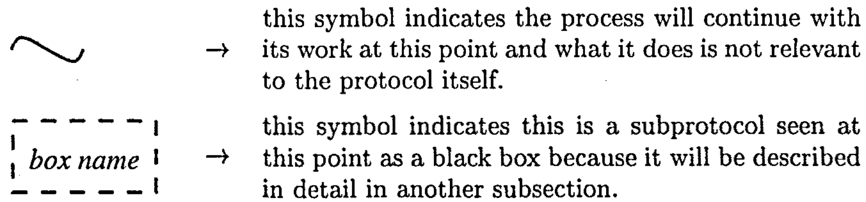
7.1 The ATLAS communications protocols

The specification of a protocol [33] consists of different parts: the *service* to be provided by the protocol, the *assumptions* about the environment in which the protocol is executed, the *vocabulary* of the messages used to implement the protocol, the *encoding* (format) of each message in the vocabulary and the *procedure rules* guarding the consistency of message exchanges.

In ATLAS the different services can determine different protocols between processes interchanging messages. For all of them, except for the "information broadcast" protocol the channels used are BSD stream sockets [34], so ATLAS assumes that the communication at a low level is totally reliable in keeping the order of messages and the assurance that none of them will be lost. For the "information broadcast" protocol the channels used are datagram sockets, which assure the correctness of messages if they arrive but do not guarantee that they arrive.

The vocabulary and encoding for messages involved in protocols are described in the next section (7.2). The service and the procedure rules (described with flow charts) for each protocol are going to be explained together in order to facilitate the understanding of the protocol itself.

Before going into the protocols description we define two symbols used in the flow charts which don't belong to the flow charts language [33].



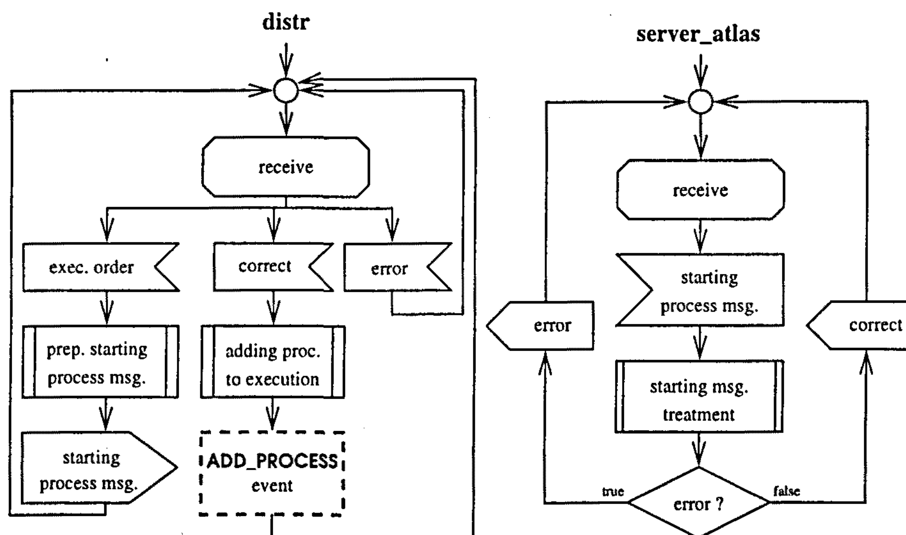
→ this symbol indicates the process will continue with its work at this point and what it does is not relevant to the protocol itself.

→ this symbol indicates this is a subprotocol seen at this point as a black box because it will be described in detail in another subsection.

The most relevant protocols in the ATLAS system can be described as follows:

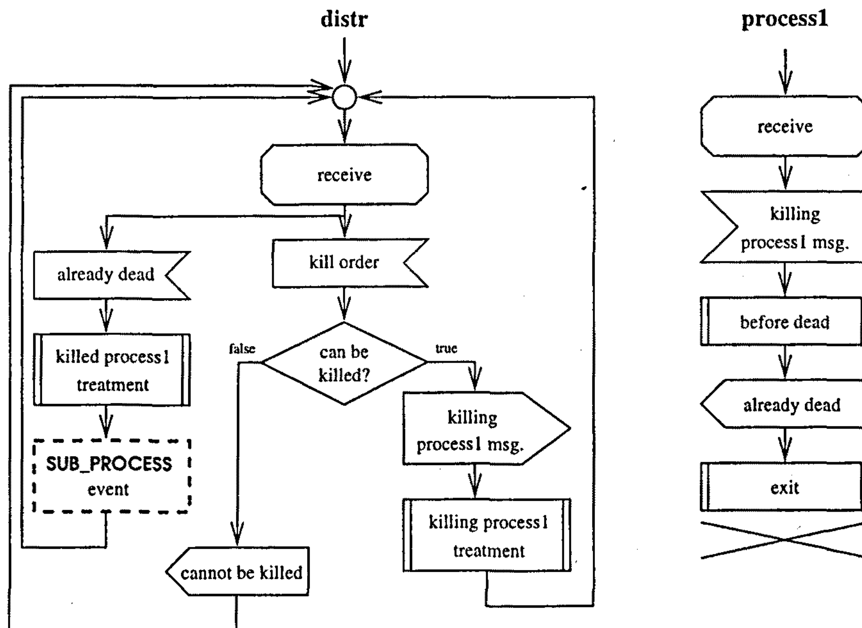
Starting a process

When the `distr` process receives an order to start a process or decides by itself that a process must be started it chooses the *best* server to execute this process and sends the starting message for that process to the chosen server. The server receives the message and tries to execute that process, answering to `distr` whether it succeeded or not.



Killing a process

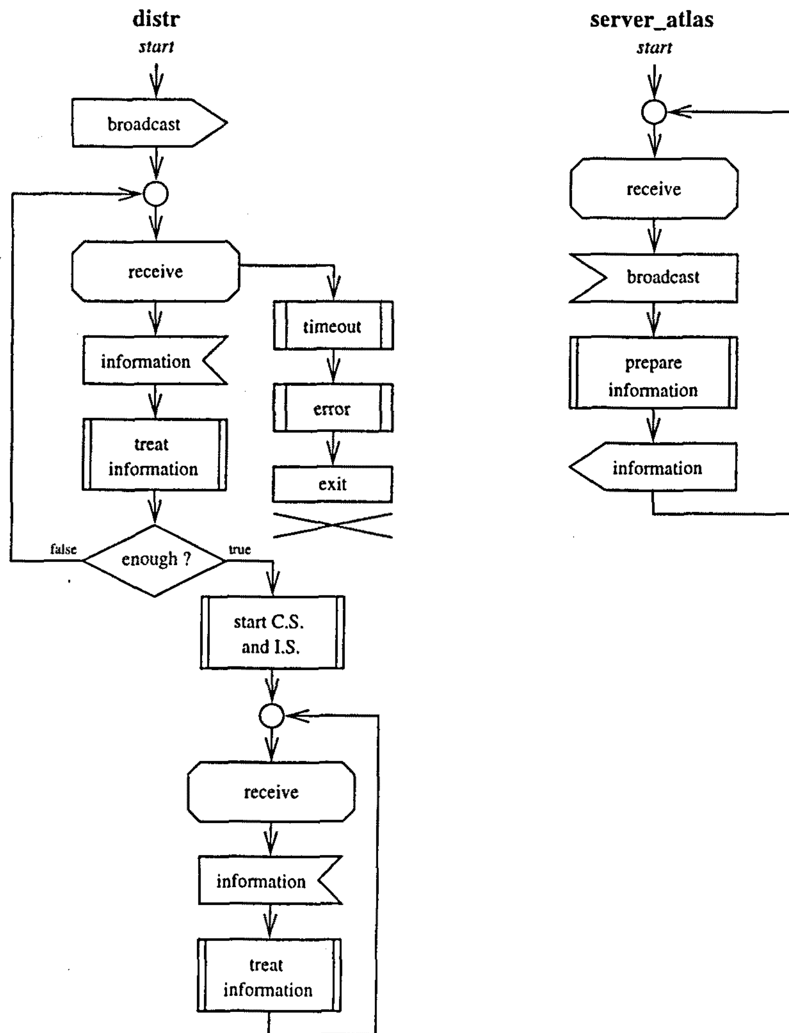
A process must be killed when the `distr` process noticed the communication with the process is lost or when there is an explicit order to finish it. If the communication is lost the *heartbeat* mechanism must know what to do, but if the process must be finished by an explicit order, a protocol is required for `distr` to negotiate with the process its close-down.



`distr` checks that the process is not essential and sends to it a destruction message, tagging this process as not accessible from this moment. The process finishes anything it was doing, receives the message and sends to `distr` the confirmation of its death just before it exits. When `distr` receives the death confirmation it can then clean up its data structures.

Information broadcast

The `distr` process sends a broadcast message and the ATLAS servers listening at the same port answer giving information about their host name, the machine type and the processes of the application that they can execute on that machine. All servers receiving the broadcast message send back the corresponding information. This protocol allows the `distr` process to have enough information to know where a process can be started at any moment.

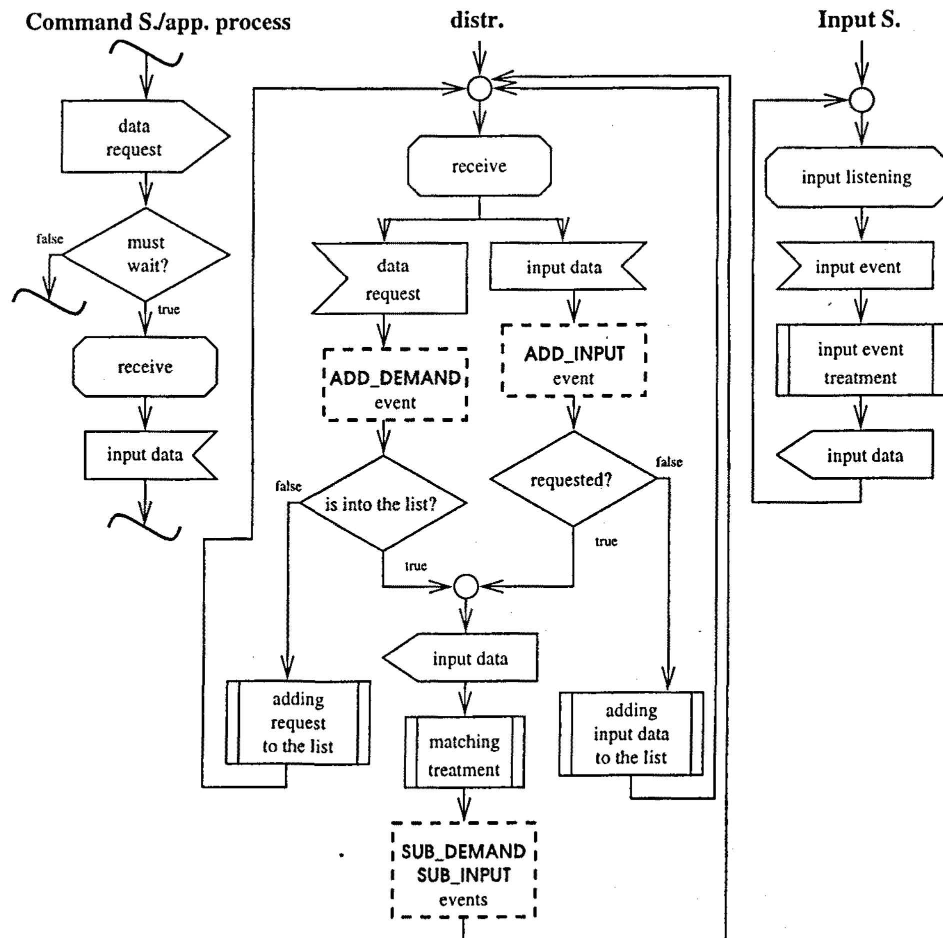


Data input and request

There are two protocols involved in the data input and request, but they are quite related to each other, so I present them as if they were only one protocol.

For an input datum being sent to `distr` by an `Input Subsystem`, `distr` checks if there are any requests waiting for this datum, and if it is the case, sends this datum to the requesting process. Otherwise it stores this datum in the input data list.

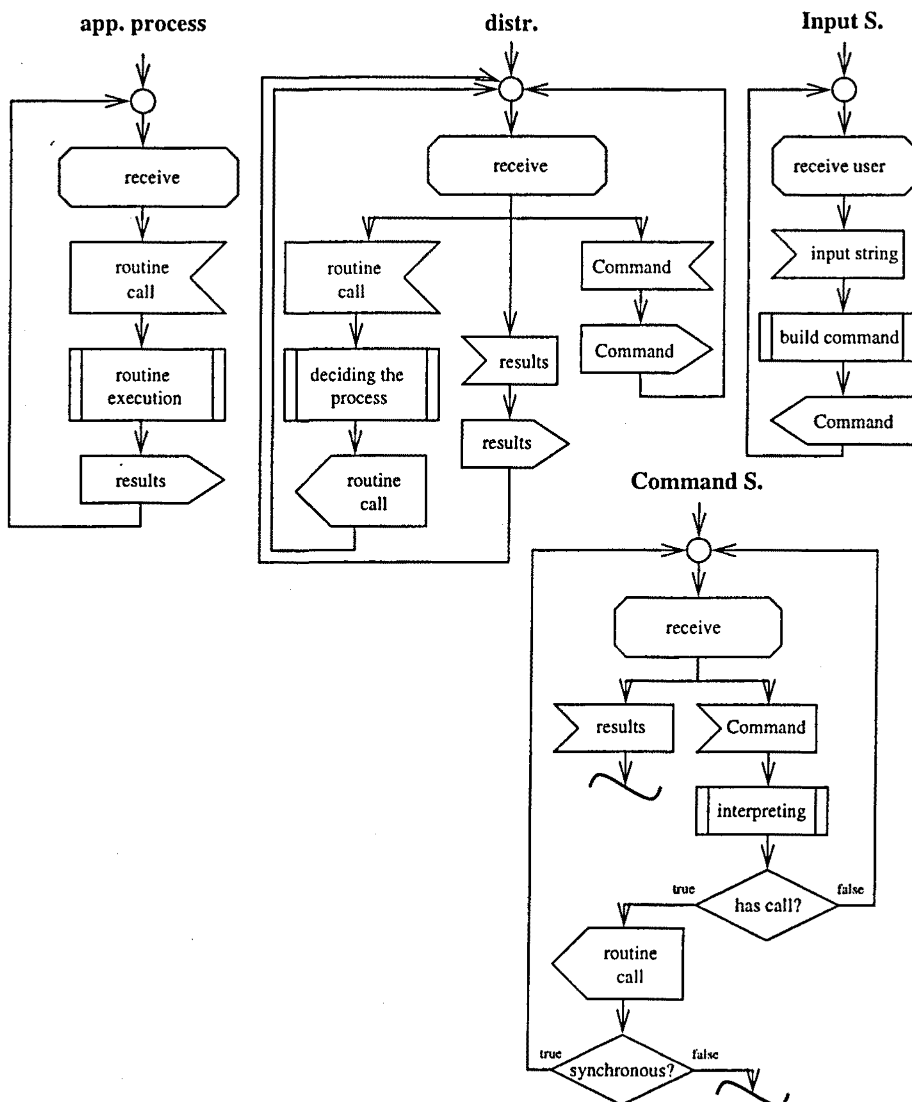
For a request being sent by any process, `distr` checks if there is any input data into the corresponding list to serve this request. In the affirmative case it sends the data to the requesting and in the negative it adds the request into the data requests list waiting until the adequate input data arrives.



User command and external routine call

When an input subsystem produces a command and sends it to `distr`, `distr` just redirects it to the `Command Subsystem` which is the ATLAS component in charge of interpreting this command.

Since an external routine call is always produced in the `Command Subsystem` and most of them because of a user command (otherwise the producer of the external routine call is the `main` procedure of a module –which can be seen also as a user command), both protocols are being presented together in this subsection.



When the `Command Subsystem` sends an external routine call to `distr`, it has to redirect the message to the right process. The reverse happens when the

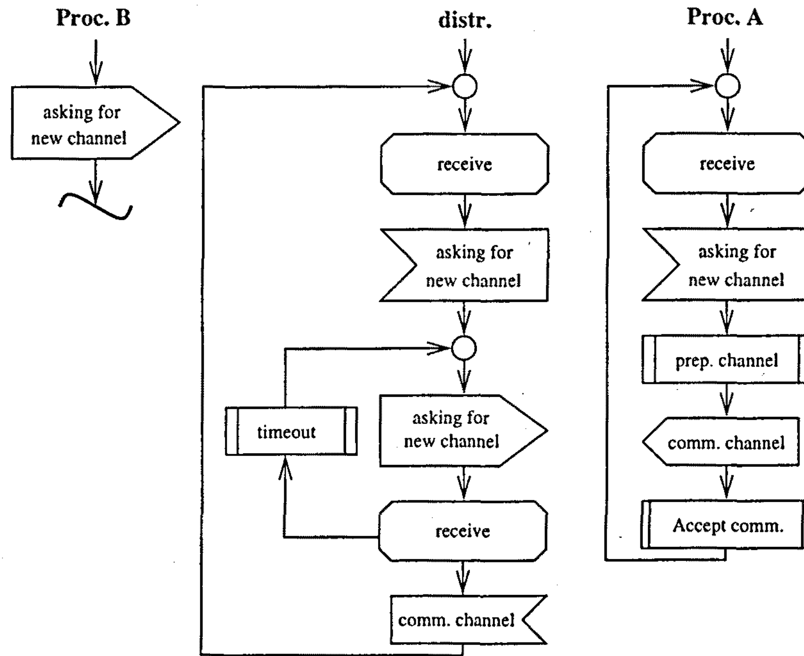
process ends the execution of the external routine and sends the results back to `distr`, which has to send them back to the `Command Subsystem`.

The external routine call can be synchronous or asynchronous. If it is synchronous the execution of this command must be stopped until the results arrive, and if it is asynchronous the execution can continue until the results of the external routine are needed (see the asynchronous management in chapter 4).

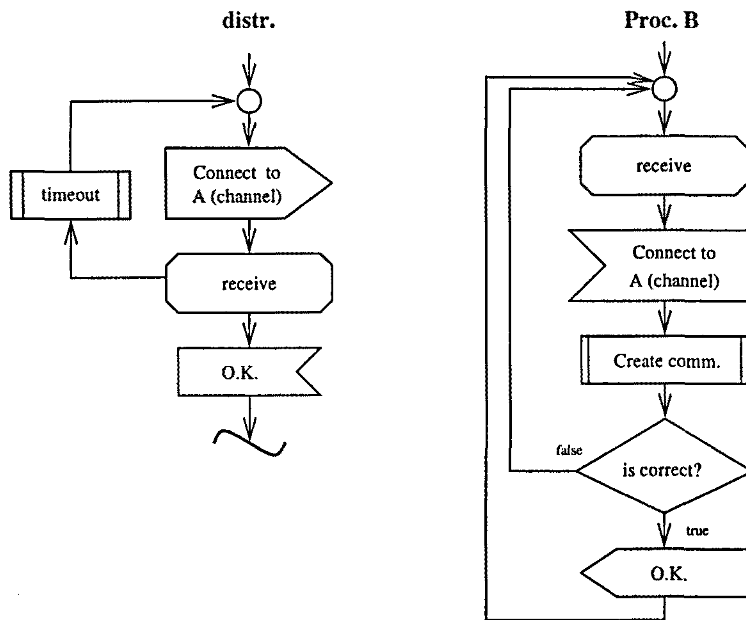
Direct communication between processes

The needed protocol to allow direct communication between processes can be divided into three parts:

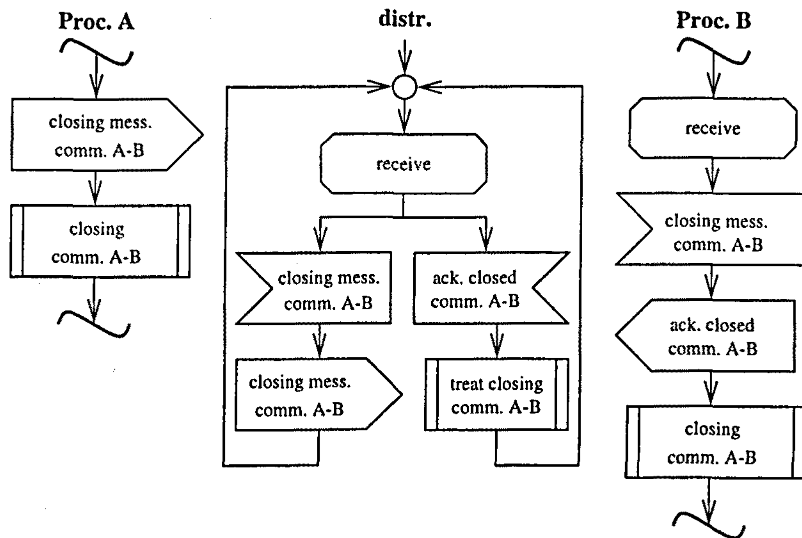
Asking for channel: Process B asks to `distr` for a communication channel with A, so `distr` sends a channel request to A which creates the channel and answers to `distr` sending the channel created. Then A waits until the other process B makes the connection to this channel.



Communication: `distr` sends to the process B the communication channel received from A. B then creates the communication and connects to this channel. Once done it sends an acknowledgement to `distr` and the channel is opened between the two processes A and B.

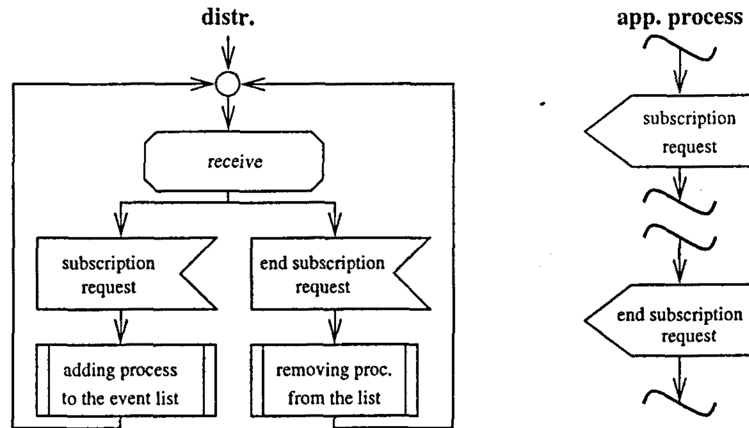


Communication closing: The request to close the channel can be produced by either one of the processes having the communication opened. When the process A decides to close its communication with the process B it sends a message to B saying it is going to close the channel. This message must go through `distr` because it has to know the channel is going to be closed. When B receives this notification it sends an acknowledgement to `distr`, and at this moment `distr` knows this channel is going to be closed without any other action over it.



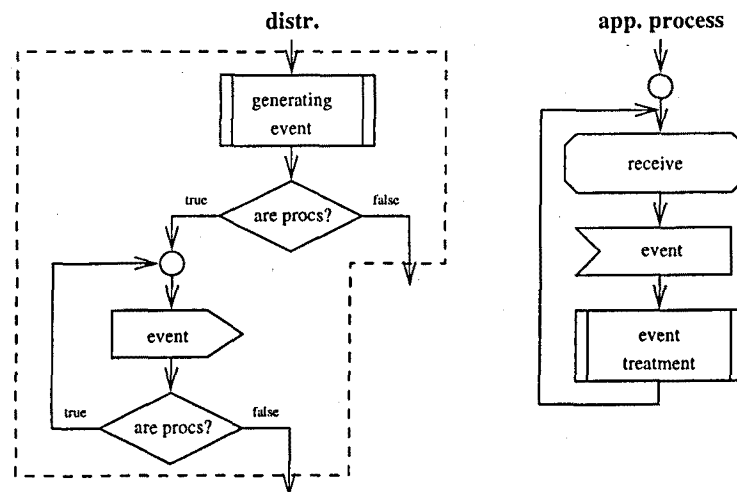
Events subscription

In the *events* subscription protocol the messages involved are: the *subscription request* and the *end of subscription request.* When *distr* receives a *subscription request* message from a process, this process is added to the list of processes subscribed to this kind of events, so that from now on this process will be notified when the event is produced. This subscription will be active until the process sends to `distr` the *end of subscription request* message or until the process ends.



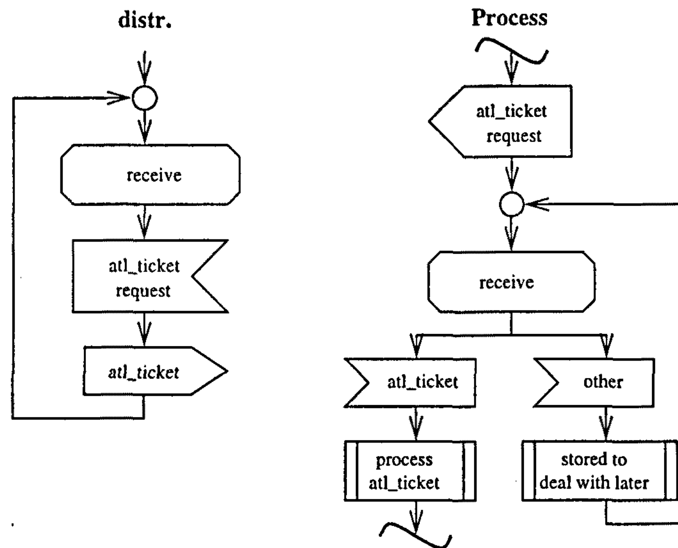
Event sending subprotocol

In this subprotocol the message involved is the message notifying this event. When *distr* detects an event happens, it sends the notifying message of this event to every process which has been subscribed to this event. These processes then will receive this event notification message and do whatever use they see fit.



Atl_ticket request

The protocol that allows a process to ask and get an atl_ticket from `distr` is a synchronous protocol. The process sends a message to `distr` asking for a new atl_ticket and waits until `distr` sends back this atl_ticket. If the process receives other message from `distr` while it is waiting for an atl_ticket, it just stores it in order to deal with it when the routine waiting for the `distr` answer returns to the driver loop. (The driver management is explained in section 7.5, and the global input identification –atl_ticket management– is explained in section 8.1.3).



7.2 The set of messages used

The messages used in ATLAS communications can be grouped depending on the sort of tasks where they are involved.

Apart from the division into these groups, which are control messages, execution messages, messages of ATLAS events and data management messages, there is another important difference between messages that must contain data of any type and messages just containing strings and integers. Messages containing data of any type have to be specially managed in order that these data be interpreted with the same meaning by different architectures in an heterogeneous network. The ATLAS `"Variables"`, fully explained in section 7.4, are used to wrap data in order to hide to the developer the internals of the heterogeneous communications.

Control messages: These messages are involved in the ATLAS internal managements. The ATLAS control messages include those involved in the broadcast mechanism, the *heartbeat* message, those necessary to handle the subscription and unsubscription of processes to ATLAS events, those involved in the journaling functionalities, the starting and finishing of a

debug management for a process and some control answers produced after a process execution request (acknowledgement or error).

The concrete control messages and their contents are the following:

`Bcast`	→	`distr` sysid, user uid, gid and ATLuserid, environment variables.
`AnswerBcast`	→	`server_atlas` host name, sysid, normalized load coefficient, list of processes available.
`ImAlive`	→	status of the process.
`Subscription`	→	event to subscribe, flag asking for update.
`EndSubscription`	→	event to finish the subscription.
`AdvisedCheckpoint`	→	—
`CheckpointDone`	→	name of file.
`RecoverCheckpoint`	→	name of file.
`DontRecover`	→	—
`AskTicket`	→	—
`AnswerTicket`	→	atl_ticket identifier.
`SubstituteTicket`	→	atl_ticket to be substituted and atl_ticket to substitute the first.
`InitDebug`	→	name of process.
`EndDebug`	→	name of process.
`Acknowledgement`	→	—
`Error`	→	error message to print.
`Dummy`	→	— (needed at certain points to wakeup a communications driver).

Execution messages: These messages produce the actual execution of the application. The ATLAS execution messages include those creating and destroying processes, the user commands which will be executed by the `Command Subsystem`, and those related to the routine calls (routine call, parameters and different results).

The concrete execution messages and their contents are the following:

`ExecProc`	→	process name.
`KillProc`	→	process name, flag of normal exit or direct kill.
`ExitProc`	→	exit status.
`ExitExec`	→	—
`Command`	→	the command string.
`CallRoutine`	→	process name, routine name, call identifier, number of params.
`Parameter`	→	process name, call identifier, data `Variable`.
`ReturnValue`	→	call identifier, return value `Variable`.
`ReturnParam`	→	call identifier, return parameter `Variable`.
`ReturnVoid`	→	call identifier.

ATLAS **events messages:** These messages are generated by `distr` when some change in its internal structures occur and has to be notified to some processes who asked for it. The ATLAS events messages are always produced because something happened. The situations to be notified are the creation and destruction of a process, the receiving of a *heartbeat* message,

the receiving of an input data or a data request message, and the elimination from the internal structure or a reordering in it of an input data or a data request.

The concrete ATLAS events messages and their contents are the following:

EvAddProcess	→	process name.
EvSubProcess	→	process name.
EvImAlive	→	process name.
EvAddDemand	→	data request object.
EvSubDemand	→	request identifier.
EvMoveDemand	→	request identifier.
EvAddInput	→	input data identifier, type name.
EvSubInput	→	input data identifier.
EvMoveInput	→	input data identifier.

Data management messages: These messages are involved in the ATLAS asynchronous matching management, so they include the data request and input data messages, the matching message (answering a request), those messages asking for reordering of input data and request and for removing a request, and also those messages used to ask for the data value into an input data object and answer to it.

The concrete data management messages and their contents are the following:

RequestData	→	data request identifier, type name, string to be printed to the user, timeout value.
InputData	→	input data `Variable`, timeout value.
AnswerData	→	data request identifier, input data `Variable`.
ModifDemand	→	data request identifier.
ModifInput	→	input data identifier.
DelDemand	→	data request identifier.
AskInput	→	input data identifier.
AnswerInput	→	input data identifier, input data `Variable`.

All these messages have a common structure which includes an identifier that allows the communications management to know which message has been received. This structure encapsulates the message so that the sending and receiving drivers do not have to know about the contents of the specific message until it has to be treated (see next section for more details on implementation). This makes the sending and receiving code be the same for all sort of messages and it facilitates also the extensibility of the ATLAS set of messages because a new kind of message just needs to add a new identifier and a new specialized management.

7.3 Some implementation details

Each ATLAS message has different contents so it must have a structure different from each other. On the other hand all of them must be treated by the communications management in the same way (being sent and received). The ATLAS

implementation for these messages encapsulates the common information into a base class and derives from it a class for each kind of message adding the specific contents.

The common information being stored into the base class, Message, is the total length of the message, the message identifier and the execution thread identifier (see section 8.1.1). The length of a message sometimes is not known a priori and in these cases the length value is set to -1 to be recognized as a message without known length. This class hierarchy also exploits polymorphism. The base class declares virtual methods to send a message through a channel and to destroy the message. These methods will be implemented by each derived class.

The ATLAS communications mechanism is implemented using sockets, and its implementation uses the wrapper classes for sockets offered by the public domain package ACE_Wrappers (see [31] and [30]).

Since ATLAS processes may be running in different architectures, data must be sent through the network using a standard representation in order that they have the same meaning to the different processes. ATLAS just relies on XDR [35] to transfer the data robustly between different architectures.

Since the translation of data to XDR is written directly to the communications channel where it has to be sent, the length of the message containing these data cannot be previously known and the communication end point receiving this message cannot detect when the XDR data is finished. To solve this problem, the ATLAS communications management adds an Out-Of-Band byte after each XDR message for the end of this XDR stream to be detected. Another Out-Of-Band byte is sent back to clear a semaphore, to avoid that a second OOB be sent through the same channel before the first one has been received.

In order to not have delays because of these semaphore stops the receiving management does not interpret immediately the messages arriving but just receives bytes looking for OOB bytes and sends back the other OOB to unlock the sending process on the other channel end point. This receiving management stores the bytes stream into a dynamical length table of char buffers and when the channel is empty a message interpreting procedure is started over this buffers table in order to actually get the messages received.

7.4 Transparent data transfer: Variables

Besides the Virtual Machine, the ATLAS Variable structure is also used to wrap user data in each process to go through the network. It includes methods to encode and decode XDR streams making these translations transparent to the developer. Following subsections explain this structure and the techniques used to implement data transfer with the least hassle to the developer.

7.4.1 Wrapper structure for data and types

The wrapper structure designed in ATLAS for data and types addresses two important requirements: (1) making the command subsystem able to access

component values of these variables (introduced in chapter 4) and (2) encapsulating these data robustly for it to travel from one process to another.

The ATLAS variables are represented internally by a tree structure guided by a compact type definition. This compact definition comes from the ATL type definition and is initialized by the ATL compiler (see section 4.2.2), which is aware of the complete type definition. As an example, figure 7.1 shows how the ATL definition of a structured type, pyramid (left), is compacted to its type representation string (right). Notice that this compacted type is, in fact, the representation of the *effective type* we talked about in section 4.1.3.

ATL type definition:

```
#deftype point STRUCT
                  x -> real;
                  y -> real;
                  z -> real;
               ENDSTRUCT
#deftype face VECTOR [3]
                  OF STRUCT
                        p1 -> point;
                        p2 -> point;
                        ident -> integer;
                     ENDSTRUCT
#deftype pyramid STRUCT
                    name -> string;
                    base -> face;
                    sides -> VECTOR [3] OF face;
                  ENDSTRUCT
```

Compacted type:

```
"S(name string,
   base V[3]S(p1 S(x real,y real,z real),
              p2 S(x real,y real,z real),
              ident integer),
   sides V[3]V[3]S(p1 S(x real,y real,z real),
                   p2 S(x real,y real,z real),
                   ident integer))"
```

Figure 7.1: Example of a type representation.

The C++ class that stores this compacted type has the interface shown in figure 7.2. The attributes of this class are the name of the type (used by the ATL compiler) and its compact definition, which describes all components of the type. The methods shown in the figure are those needed to manage the access to the different components of the type.

```
class Type {
    String tipus,deftipus;       // deftipus contains the compacted type
  public:
    ...
    int Components ();           // Returns the number of components.
    int Accedir (char *cami);    // Returns the component index from its name (only
                                 // for structures).
    char Codi ();                // Returns a code showing the node type. It is
                                 // useful when we need to make explicit castings.
    Type TipComp (int index);    // Returns the component type from its index.
    String NomCamp (int index);  // Returns the component name from its index (only
                                 // for structures).
      ...
};
```

Figure 7.2: Interface for class Type.

The variables itself are represented by the C++ class Variable, shown in figure 7.3. This representation has been designed to address three different issues:

- it allows ATLAS to be undisturbed by the data type (ATLAS sees them as Variables and does not care about their internal type);

- it contains both the variable tree representation and the standard XDR representation of the variable (where one is computed from the other only when necessary);
- it offers methods to translate automatically from one to the other representation. These translations are possible because the variable is always aware of its own type definition.

```
class Variable {
    String nom;            // name of the variable
    Type tipus;
    node *arbre;           // root node of the tree representation
    XDR reprxdr;           // XDR representation and
    char *mem;             // the position where data
    int posdada,xdrlong;   // is on the XDR stream
    atl_tkt ticket;        // global data identifier
    int posticket;
  public:
    Variable () {}
    Variable (String t, String n);
    Variable (Type t, String n);
    Variable (const Variable & v);
    Variable (char *m, int lng);
    void crea_arbre ();
    int arbre_to_xdr (FILE *f);   // translation to XDR, directly to the channel
    int xdr_to_arbre ();          // translation from XDR
      ...
};
```

Figure 7.3: Interface for `Variable` class.

The tree representation in the `Variable` class is a pointer to the root node of the tree. The different possible nodes can be divided in two sets: one for the *intermediate nodes* in the tree and the other for the *leaf nodes*. The *intermediate nodes* contain references to other nodes in a forward step of the tree. The *leaf nodes* contain the corresponding value in this field of the variable. The *leaf nodes* values are always of basic types whereas the *intermediate nodes* correspond to type constructors. Figure 7.4 shows the node tree representation for a `pyramid` type variable.

The C++ representation used to implement this node tree is based on polymorphism. There is an abstract base class `node` containing a `Type` which is the type of the node and a number of references to control how many copies of this node are in use (see figure 7.5). The rest of node classes derive from this one and represent the different possible kinds of nodes so far: `nodestruct` and `nodevector` represent the two possible *intermediate nodes*, and `nodeenter`, `nodereal`, `nodeboolea` and `nodestring` represent the possible *leaf nodes* (for integer, real, boolean and string basic types respectively).

The use of the compact type definition facilitates the construction of the variable tree because it can be done recursively having only basic types in the leaves and building the intermediate nodes as records or arrays of other nodes or leaves.

Although the set of types usable in the ATL language is restricted to records and arrays, the design for the ATLAS variable representations makes it possible to add extensions easily. It would be quite easy, for example, to extend the

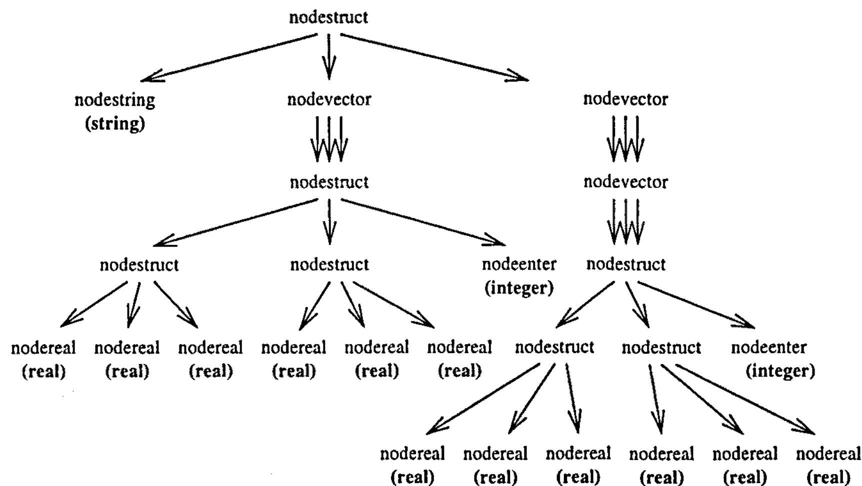


Figure 7.4: Node tree representation for a pyramid type variable.

```
class node {
  protected:
    Type tip;
    int referencies;
  public:
    node () {}
    node (Type tipus) : tip(tipus) { referencies = 1; }
    virtual node *accedir (int index) {}
    virtual node & operator = (const node & n) = 0;
    virtual int fromto_xdr (XDR *xdrs) = 0;       // translation from/to XDR
      ...
};
```

Figure 7.5: Interface for the abstract class node.

accepted types in ATL to lists or hash tables by including the description of these types in the language and extending the node types (classes deriving from node) with nodelist and nodehash. Because of the modularity in the Variable mechanism design the only effort needed to do would be inside these two new node derived classes in order to achieve the methods implementation for the translation to XDR and the access to their components.

7.4.2 Making this design transparent to the developer

Not only does ATLAS offer the automatic translation between the XDR representation and the ATLAS Variable representation (included in the Variable methods), but it also isolates the developer from these ATLAS Variables' representation.

In order to achieve the desired transparency for the developer, ATLAS provides code stubs to automatically transfer the user's data into ATLAS Variables, and backwards, through *bridge types* used to isolate the user from the details of the ATLAS Variables (which an advanced user can use directly if he wishes to).

These code stubs are automatically constructed by ATLAS from the interface declaration of the process, which contains the type definitions used for variables to be exported and the prototype definitions of the process external routines, which describe the parameters and result types for them. All these definitions give ATLAS enough information to generate code stubs to prepare the arguments for user functions or collect results and encode them for being transported over the network, and dispatch calls to user functions.

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

Examples of *bridge types* and also the other code automatically generated by ATLAS can be seen in section 7.5.1.

7.5 Process driver

Each process in an ATLAS application is connected to `distr` by a communications channel, therefore each process needs a communications driver to manage the interchange of messages with the rest of the application.

The default for an ATLAS process driver is to manage the channel connecting with `distr` and serving the interruption of SIGALRM, which is used in the *heartbeat* mechanism (see section 6.1.2). The most common requests or messages received from `distr` are routine calls, data answering a request, or an ATLAS event notification. Figure 7.6 shows schematically the process driver role.

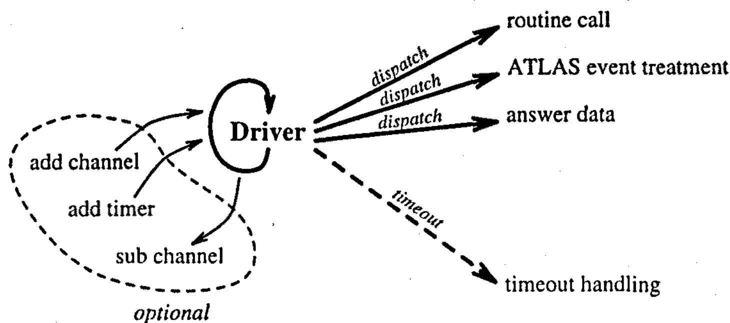


Figure 7.6: Driver role scheme.

Users also may: add other channels for special purposes (for example to listen to X-windows events); add a timer treatment; or remove channels added

before. This flexibility only requires to have implemented the processing of messages being received through these new channels.

Aside from routine calls, parameters and answers to requests –see subsection *"Automatic code generator"* below–, a process may receive other kinds of messages from `distr`, including:

- the contents of some input data item that the user introduced and is still in the system's internal structures. This message is produced in response to an explicit request of this information made by the process,
- an order to finish the process execution. This order causes the call to a finishing routine that the developer can define specially for each process,
- a system event notification. The ATLAS events mechanism gives the opportunity that an application process be informed about changes in ATLAS internal structures (what processes are in execution, if there is some request waiting for an input data, etc –see section 6.4).

There are also other special messages a driver may receive, but they are only used by the Command Subsystem (a command, a return value or a return parameter). Although it is not advisable to change the Command Subsystem process (because it is a heavy-weight component of ATLAS), if some application intended to replace the Command Subsystem, it would need to use these special messages for its communications.

7.5.1 Automatic code generation

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

The ATLAS process communications require quite a bit of code in each process devoted to handshaking with `distr`, generating the *heartbeat* messages at the adequate rate, preparing the arguments for process routines or collecting results and encoding them for being transported over the network, and dispatching calls to process routines. But the programmer should be relieved from these tasks.

To handle this, ATLAS automatically generates code stubs that the developer must link with his program.

The code generator and the compiler grammar of the Command Subsystem

The code stubs to be automatically generated are constructed from the interface declaration of the process (like in figure 7.7), which contains the type definitions used for variables to be exported and the prototype definitions of the process external routines.

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

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

Figure 7.7: Portion of the interface definition in ATLAS for the volume modeling process ("volum").

Since the same language (ATL) is used to define the process interface (more about the process interface in chapter 9) and also to describe the application behaviour (interpreted at run time by the Command Subsystem –explained in chapter 4), we use the same parser in both cases, when the generator is going to generate the code stubs for the communications mechanism (explained in next subsection) and when the interpreter of the Command Subsystem is interpreting the ATL code at run time. The actions the parser does are determined by a flag indicating whether it has to generate code or it is directly interpreting code (run time).

When the parser is doing code generation, it is mainly interested in extracting information about the exported type definitions and about the prototypes of external functions or procedures, to build the code stubs for the communications driver.

Automatic code generator

The automatically generated code must implement the default for an ATLAS process driver (managing the channel with `distr` and serving the SIGALRM interruption *–heartbeat*). The generator creates the driver's code in several files (see *"Automatically generated files"* below) that are eventually linked with the developer's code.

In this subsection we will talk in depth about the design decisions these requirements lead to, and we will show how the automatically generated code works.

Calls, parameters and results

In the '.atl' file defining its interface, each process offers (makes public) some types and routines. These routines are either routines implemented in the ATL module itself, written in ATL language or external routines, implemented in C++, by the process developer. They are called external routines, because they are executed by the ATLAS process and not by the ATLAS Virtual Machine. Once running, each process will be waiting for requests coming from the system to execute its external routines. So, the need for communication between the system and the process arises: calls and parameters should be passed from `distr` to the process and results should go the other way (see figure 3.1(b)).

For each external routine call, `distr` will send the involved process a message communicating the request, and one additional message for each parameter of the routine. After executing that routine, the process will send back a message with the return value (that may be void) and an additional message for each parameter passed by reference.

Bridge types

In order to make type checking, types of the parameters of the external routines declared in the '.atl' file must be defined (or imported) there. As these are ATL type definitions and not C++ types, conversions must be made somewhere from the internal ATLAS storage of variables (of type `Variable` –see section 7.4) to C++ variables.

The automatic code generator produces code that carries out the conversion from the XDR message to a *bridge* C++ type (see also section 7.4.2). The communications driver is able to recover the `Variable` from the XDR message. On the other hand, the bridge class definition and conversions from and to it are provided in auxiliary routines (see below).

```
ATL type definition:

    EXPORT #deftype point STRUCT
                            x -> real;
                            y -> real;
                            z -> real;
                          ENDSTRUCT

Bridge type:

    struct atl_point {float x; float y; float z;
    atl_point() {}
    atl_point(Variable &v) {
       if (v.Arbre()==NULL) atl_exit(-1); // Invalid variable
       x = ((nodereal *)(*(v.Arbre())).accedir(0))->Getvalor();
       y = ((nodereal *)(*(v.Arbre())).accedir(1))->Getvalor();
       z = ((nodereal *)(*(v.Arbre())).accedir(2))->Getvalor();
       }
    operator Variable() {
       Type t("volum::point","S(x real,y real,z real)");
       Variable v(t,"");
       v.crea_arbre();
       *((*(v.Arbre())).accedir(0)) = x;
       *((*(v.Arbre())).accedir(1)) = y;
       *((*(v.Arbre())).accedir(2)) = z;
       return (v);
       }
       };
```

Figure 7.8: Conversion from an ATL type to the corresponding bridge type.

In figure 7.8 we can see an example of an ATL type and its corresponding C++ *bridge type*. All the information needed to construct this C++ class is provided by the ATL definition of the type, and therefore all the code for the bridge type shown in figure 7.8 has been automatically generated. As can be seen, the conversion from and to an ATLAS `Variable` has been automatically generated. The first direction is provided with a constructor of the bridge class. The other one is made using a conversion operator.

With this mechanism, the conversion from the received XDR message to the *bridge type* is made transparently to the developer, as well as the conversion from a *bridge type* to a `Variable`, a step needed in order to send back results and parameters of the call passed by reference.

So, the only step where the developer must act is in the last conversion: from the bridge type to the actual parameter type in the C++ side. This last step is kept manual in order to offer more flexibility: ATL types need not exactly fit the definitions of the application's C++ classes. Furthermore, incompatibilities between C++ and ATL types can be circumvented.

In counterpart, the developer must write the routines to convert an object from and to the *bridge type*. The automatically generated code will use this conversions to translate parameters and results. It should be noted that this conversion is usually very simple and easy to write, as the bridge class members usually match the actual class members. This two-step design has been adopted so that developers need not handle `Variables` directly in most situations. A scheme of the different representations a parameter has in its "trip" through the network can be seen in figure 7.9.

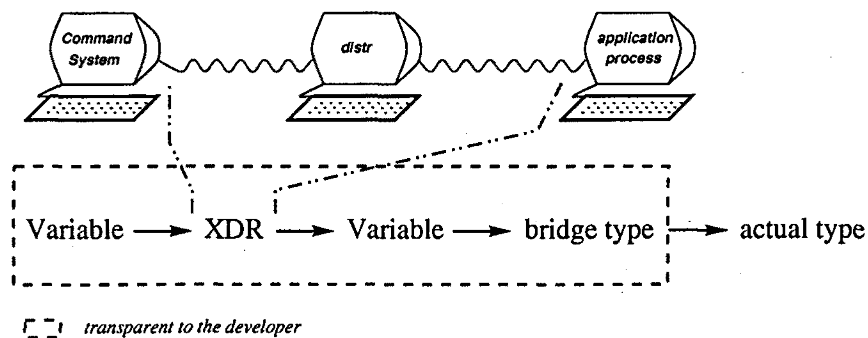


Figure 7.9: Scheme of the different parameter representations along its trip through the network.

A better solution for these last conversions (*bridge type* ↔ actual type) would be the derivation of the *user type* (actual type) from the *bridge type*. This solution has not been adopted yet in the present prototype because of backward compatibility of user applications code, so the examples used in the thesis still use the first adopted solution. The derivation solution is better because it avoids unnecessary copies of the object contents and is more intuitive and easier for the developer. When the derivation is made public the only required in the derived class is a constructor from a `Variable` object like the following:

```
user_type (Variable &v): bridge_type(v) {
   // the addings required by the user type
       ...
}
```

If the derivation is private, besides the constructor, a redefinition of the `Variable()` operator should be added:

```
operator Variable() { bridge_type::operator Variable(); }
```

Auxiliary routines

In order to isolate application developers from communications topics, an auxiliary routine is generated for each external routine. The auxiliary routine carries out all the type conversions, from and to the XDR representations that travel through the network and to the actual C++ classes of the parameters and return value of the external routine. This is done using the *bridge types* already explained. Figure 7.9 shows one way of this conversions. The other way is necessary in order to send return values and pass-by-reference parameters back to the external routine requester.

From the ATL declaration of an external routine, the automatic code generator extracts the information on the type and number of parameters and the type of the return value (see subsection *"Calls, parameters and results"*).

When an ATLAS process requests the execution of an external routine from another ATLAS process, `distr` receives this request. Many requests for the same routine may arrive at a time. An identifier is assigned to each of them in order to be able to send the return value and the existing pass-by-reference parameters back to the appropriate process. So, `distr` assigns each call a different identifier code.

An auxiliary routine receives two parameters. The first one is the code identifying the call. This code will be included in the return messages the auxiliary routine will send to `distr`. The second parameter consists of the list of actual parameters of the external call. These parameters will be translated to the appropriate C++ types through the *bridge types* and passed to the process routine for which the auxiliary routine acts as an interface.

When the actual execution of the process routine finishes, the auxiliary routine sends a message for each pass-by-reference parameter back to `distr`. An additional message is sent for the return value of the routine. If this return value is of type void, a special message is sent to indicate the end of the execution. In this last situation, if there exists any pass-by-reference parameters, the return-void message is not necessary (as the parameter itself is enough to indicate the end of the execution), and thus it is not sent.

An example is shown in figure 7.10. In this example the process routine receives a parameter passed by reference and has no return value, so only the parameter is sent back to `distr` at the end of the process routine.

Some of the complexity of the code in this example arises from the combinatorial complexity originated by the *"Global data identification"* functionality

(see section 8.1.3). The conditionals "`if`" and "`switch`" thus, are covering all possibilities for a parameter or a return value containing data or being only a global identifier (for re-executions when data have been substituted –again explained in *Global data identification*). When only an identifier must be treated the *Variable* only contains this identifier (an `atl_tkt`), so no translation through the *bridge type* is needed ("`else`" code blocks).

```
1     void aux_segmentation(String codi,DLList<Variable *> &parametres) {
2        Pix p=parametres.first();
3        char indexpar=0;
4        atl_tkt ticketaux;
5        ticketaux=parametres(p)->Ticket();
6        io_abstract *iopar0;
7        if (parametres(p)->Gettype().Gettip()!="atl_ticket") { // parameter contains data
8           atl_scene ptp0(*(parametres(p)));
9           scene par0(ptp0);
10          iopar0=new io<scene>(ptp0,ticketaux);
11          ((io<scene> *)iopar0)->ChangeMakecopy();
12          indexpar=(indexpar<<1); indexpar+=1;
13          }
14       else { // parameter does not contain data, it is only a global identifier
15          iopar0=new io_base(ticketaux);
16          indexpar=(indexpar<<1);
17          }
18       parametres.next(p);
19       ticketaux=parametres(p)->Ticket();
20       io_abstract *iopar1;
21       if (parametres(p)->Gettype().Gettip()!="atl_ticket") {
22          property ptp1(((nodeenter *)parametres(p)->Arbre())->Getvalor());
23          iopar1=new io<property>(ptp1,ticketaux);
24          indexpar=(indexpar<<1); indexpar+=1;
25          }
26       else {
27          iopar1=new io_base(ticketaux);
28          indexpar=(indexpar<<1);
29          }
30       parametres.next(p);
31       switch (indexpar) {
32          case 0: segmentation(*((io_base *)iopar0),*((io_base *)iopar1));
33                  break;
34          case 1: segmentation(*((io_base *)iopar0),*((io<property> *)iopar1));
35                  break;
36          case 2: segmentation(*((io<scene> *)iopar0),*((io_base *)iopar1));
37                  break;
38          case 3: segmentation(*((io<scene> *)iopar0),*((io<property> *)iopar1));
39                  break;
40          }
41       ticketaux=iopar0->Ticket();
42       if (iopar0->Containsdata()) {
43          ptp0=((io<scene> *)iopar0)->Dades().conversio_a_tipus_pont();
44          Variable *rp0=new Variable(ptp0);
45          rp0->AddTicket(ticketaux);
46          ReturnParam *retpar0=new ReturnParam(codi,rp0);
47          distrib.envia(retpar0);
48          }
49       else {
50          Variable *rp0=new Variable(ticketaux);
51          ReturnParam *retpar0=new ReturnParam(codi,rp0);
52          distrib.envia(retpar0);
53          }
54       delete iopar0; delete iopar1;
55       }
```

Figure 7.10: Example of an auxiliary routine.

Line 8 contains the conversion from the `Variable` representation to the bridge class. The conversion from the bridge class to the actual parameter class happens in the next line. The user's routine is called in one of those cases

treated in the `switch` conditional (lines 32, 34, 36 or 38) depending on whether parameters contain data or not. The pass-by-reference parameter is translated, in case it contains data, to the bridge type in line 43 and to `Variable` in the next one. Lines 46 and 47 construct the message and send it back to `distr` (when the parameter going back does not contain data it is done by lines 51 and 52). The message contains the identifier of the call and the parameter itself. A more complete example can be seen in chapter 9.

Management of remote procedure calls in an ATLAS *process*

In order to carry out the dispatch of calls to the routines offered by an ATLAS process (its external routines), some more code is needed in the automatically generated code. The missing piece is a structure bridging between messages and the auxiliary routines introduced in the previous section. This structure is provided by the class `gestio_crida_a_rutina` (that is, routine call management). The class definition is shown in figure 7.11. One instance of this class is created in the automatically generated code for each process. It is in charge of receiving and storing routine calls and parameters and of calling auxiliary routines.

```
typedef void (*rut_tract_crida)(String,DLList<Variable *> &);

class gestio_crida_a_rutina {
    VHMap<String,rut_tract_crida> crida_a;    // table to store the routines to be
                                              // called for each function
    VHMap<String,dades_crida *> crides_pendents;  // table to store the routines and
                                                  // parameters while they are arriving
  public:
    gestio_crida_a_rutina() : crida_a((rut_tract_crida)NULL,20),
                              crides_pendents((dades_crida *)NULL,20) { }
    void lligar_nom_crida(String nom,rut_tract_crida rut) {crida_a[nom]=rut;}
    void nou_missatge_crida(String codi,String nom,int npars) {
       if (npars==0) crida_a[nom](codi,DLList<Variable *>());
       else {
          dades_crida *aux = new dades_crida(nom,npars);
          crides_pendents[codi]=aux;
          }.
       }
    void nou_missatge_param(String codi,Variable *v) {
       crides_pendents[codi]->afegir_parametre(v);
       if (crides_pendents[codi]->ja_tots_els_parametres()) {
          crida_a[crides_pendents[codi]->nom()] (codi,
                                     crides_pendents[codi]->obtenir_parametres());
          crides_pendents[codi]->alliberar_parametres();
          delete crides_pendents[codi];
          crides_pendents.del(codi);
          }
       }
  };
```

Figure 7.11: Class *gestio_crida_a_rutina*

Let's take a closer look at this structure. Basically, it consists of two maps: each external routine is linked with its auxiliary routine (through the class member `crida_a`) and each external call is linked with its parameter list (through the class member `crides_pendents`). The first link is established in the initialization step of the process, inside the automatically generated procedure `ini_per_crides`. The second link is created each time that a request for a routine reaches the process. Notice that an external routine cannot be linked with a unique parameter list, because many calls to that routine may be pending at a given time.

When a routine call message sent by distr arrives to the process driver, the method nou_missatge_crida (that is, new call message) is invoked. The parameters of the method are the code identifying the call (see subsection *"Auxiliary routines"*), the name of the routine and the number of parameters of the routine (this information is extracted from the routine call message). If the number of parameters is zero, the corresponding auxiliary routine is instantaneously invoked. Otherwise, the call is stored and the auxiliary routine will be effectively invoked when all the parameters of the call have arrived.

When a parameter message sent by distr arrives to the process driver, the method nou_missatge_param (that is, new parameter message) is invoked. It adds the parameter to the call's parameter list. If all the parameters of the involved call have arrived, the call to the auxiliary routine is dispatched. Notice that the parameters of an external call arrive in order, because they are sent in order by distr.

```
class dades_crida {
      String funcio;                  // function name
      DLList<Variable *> parametres;  // pointers to the received parameters
      int quants_parametres_falten;   // how many parameters are missing
   public:
      dades_crida() : funcio(""),quants_parametres_falten(0) {}
      dades_crida(String n,int np) : funcio(n),quants_parametres_falten(np) {}
      String nom() {return funcio;}
      DLList<Variable *> &obtenir_parametres() {return parametres;}
      bool ja_tots_els_parametres() {return (quants_parametres_falten==0);}
      void afegir_parametre(Variable *v) {
         parametres.append(v);
         quants_parametres_falten--;
         }
      void alliberar_parametres() {
         for (Pix p=parametres.first();p!=0;parametres.next(p))
            delete parametres(p);
         parametres.clear();
         }
   };
```

Figure 7.12: Class *dades_crida*

An auxiliary class is used to temporarily store the parameters of the calls. This structure, dades_crida (that is, call data), is shown in figure 7.12. For each call, it stores the name of the routine, its parameter list and how many parameters are still missing.

main() part of processes

The code generator also constructs the main() module of the ATLAS process. It consists of the code needed to control the network communications plus code for the auxiliary routine introduced in subsection *"Auxiliary routines"*.

Part of this code is the same for every ATLAS process. So, the generator simply appends the process-dependent code with the process-independent part.

Figure 7.13 shows the piece of code that remain the same for every process. As can be seen, it defines and uses global variables that are in charge of communications and external routine call management:

```
#pragma implementation "taula.h"
#pragma implementation "Map.h"
#pragma implementation "VHMap.h"
#include "globals.H"
#include "ComunicDistr.H"
#include "Driver.H"
#include "String.h"
#include "gestio_crides.H"
#include "Variable.H"
#include "DLList.h"
#include "iodades.H"

#ifndef COMUNIC_DISTR
#define COMUNIC_DISTR Comunic_Distr
#endif
#include "inc_atlas.H"

String nomprogram;
gestio_crida_a_rutina gestor_crides_ext;
COMUNIC_DISTR distrib(CANAL_COMUNIC_DISTR);
Driver driv(distrib);

void main(int argc,char **argv) {
   nomprogram=argv[0];
   ini_per_crides();
   driv.set_name_program(nomprogram);
   ini_process();
   driv.Dispatch();
   close(CANAL_COMUNIC_DISTR);
   exit(0);
   }
```

Figure 7.13: Process-independent code

- The `distrib` object encapsulates the communication channel with `distr` and is responsible of the dispatching of messages depending on its contents. The description for this `Comunic_Distr` class is:

```
class Comunic_Distr : public MyEventHandler
{
    ACE_SOCK_Stream canal_com;
    Receiver_socket rebuts;
    FILE *fd_stream;
    typfunc Events[NEVENTS+1];  // to keep the callbacks for ATLAS events
    bool waitsemph, newimalive;
    ImAlive im;                 // This message cannot be created dynamically because
                                // it is used in the interruption call.
    bool oob_dos_bytes;         // flag to bypass the Out_Of_Band Data error
                                // detected in Solaris 2.5

    void soc_viu ();
    void Handle_Message (Message *miss);

  public:
    Comunic_Distr (ACE_HANDLE fd);
    void ini_event_function (int ev, typfunc f);
    void initialize (char *c) { oob_dos_bytes = (c[0]=='T'); }
    bool is_oob_dos_bytes () { return oob_dos_bytes; }
    int envia_oob ();
    int envia (Message *miss);     // method to send a message to distr

    ACE_HANDLE get_handle () const { return canal_com.get_handle(); }
    int handle_input (ACE_HANDLE fd);    // method called when there is input
                                         // through this channel
    int handle_signal (int signum, siginfo_t * = 0, ucontext_t * = 0);
};
```

- The `driv` object encapsulates the process driver itself, it manages the loop listening on the process' channels (at least the channel communicating it

with distr). It also offers the possibility to add and remove other channels to be listened to. The description for this Driver class is shown below.

```
class Driver
{
    ACE_Reactor reactor;
    String nomprogram;

  public:
    Driver (Comunic_Distr &d);
    Driver (Comunic_Comp &d, String nom)
      {
        nomprogram = nom;
        reactor.register_handler (&d, ACE_Event_Handler::RWE_MASK);
      }
    void set_name_program (String nom) { nomprogram = nom; }
    void Add_handler (ACE_Event_Handler &e,
                     ACE_Reactor_Mask mask=ACE_Event_Handler::RWE_MASK)
      { reactor.register_handler (&e, mask); }
    void Add_handler (int signum, ACE_Event_Handler *new_sh,
                     ACE_Sig_Action *new_disp=0, ACE_Event_Handler **old_sh=0,
                     ACE_Sig_Action *old_disp=0)
      { reactor.register_handler (signum, new_sh, new_disp, old_sh, old_disp); }
    void Remove_handler (ACE_Event_Handler &e,
                        ACE_Reactor_Mask mask=ACE_Event_Handler::RWE_MASK)
      { reactor.remove_handler (&e, mask); }
    void Add_timer (ACE_Event_Handler &e, const ACE_Time_Value &delta,
                   const ACE_Time_Value &interval, const void *a=NULL)
      { reactor.schedule_timer (&e,a,delta,interval); }
    void Dispatch () { for (;;) reactor.handle_events (); }
};
```

- The gestor_crides_ext object has been explained in subsection *"Management of remote procedure calls in an* ATLAS *process"*.

It is also worth mentioning that the ini_process routine called in the main function (figure 7.13) is intended to allow the developer to make some initializations of the process before entering the dispatching loop. This routine does nothing by default, but the developer can redefine it to include the process initializations.

The process-dependent code simply consists of the auxiliary routines and the procedure ini_per_crides. The task of this procedure is to initialize the structure that links external routines with their corresponding auxiliary routines.

Automatically generated files

There are two sort of files containing the automatically generated code: the permanent files and the temporary files.

The names for the permanent files consist of the name of the process (procname) prefixed by *atl_* or *stub_*, and they are:

***atl_*procname.*H*:** This file has the *bridge types* implementation for those types used by the external routines of the process (exported types in the ATL module). It includes the Variable.H file needed for the conversion of *bridge types* and those generated ".H" files corresponding to the modules used by it (in the example of section *The code generator and the compiler grammar of the Command Subsystem* the volum.atl module uses the se module so the atl_volum.H generated file includes also the atl_se.H file.

***atl*_procname.*C*:** This file implements the main code for the communications driver and also the auxiliary routines introduced in subsection *"Auxiliary routines"*.

***stub*_procname.*H*:** This file contains the stub prototypes necessary to cover all possibilities of calling the actual routine used by the "`switch`" conditional in the auxiliary routines (see *Auxiliary routines* above).

The temporary files use unique temporary names given by the system in order to not disturb any other existing file. There are several temporary C++ code files and a script file.

The C++ code files (named *_stubXXX.C*, where *XXX* is the unique part given by the system) implement the different actual routine stubs (those whose prototypes are in the *stub*_procname.*H* file). Since these routines should not be executed in a normal correct execution, they just produce an error notifying the developer something is missing in his code (an example is shown in figure 7.14). The use of a nopar parameter in this routine is a trade off to solve an implementation problem explained in section 8.1.3 *-Global data identification.*

```
#inlcude "iodades.H"
#include "inc_atlas.H"
#include "volum.h"

io<simplex> contained_in (const nopar &) {
   atl_send_error ("A stub routine has been called",'e');
   io<simplex> ret;
   return ret;
   }
```

Figure 7.14: Stub routine for the `contained_in` routine.

The script file (named *_scptXXX*) executes a Bourne shell and compiles all these temporary C++ files, creating a dynamic library named *libstub*procname.*so* that must be linked with the rest of the components of the process.

A complete example of this code generation for a toy process is presented in section 9.4.

7.5.2 Handling of ATLAS events

When a process wants to be subscribed to an ATLAS event it only has to call the API routine *"atl_subscribe (event, function)"*, giving to it the event identifier and the routine to be called when an event of this kind arrives (see chapter 9 for the complete description of this API routine and its parameters).

In order to be able to treat the ATLAS events messages arriving to the process, the driver keeps the information of the callback functions passed as parameters to those *atl_subscribe* calls, so when an ATLAS event message arrives the driver only has to start the corresponding routine.

7.5.3 Requesting input data

One of the messages arriving to a communications driver can be the data fulfilling a previous request. Any application process can cause a data input request

by using the API routine call *"atl_send_request (typename, message, timeout)"* (described in detail in chapter 9), so the `distr` asynchronous matching mechanism will send back the input data being matched to this request.

The implementation for the management of these requests in the current prototype only permits these requests to be totally asynchronous, so the process sends the request and goes on with its work and when the answering data arrives a user routine is called which should be implemented by the developer to treat this message. This routine must conform to the following prototype (described in more detail in chapter 9):

```
void treat_data (AnswerData miss);
```

Since this management is a little bit restrictive –because it doesn't allow the process, for example, to wait for the answering data stopping until it arrives–, a new design has been thought which is more complete and more flexible. This new design has not been added to the prototype yet, but is explained in detail in chapter 10 (section 10.1).

Chapter 8

Journaling

The ATLAS journal records all the relevant actions that occur in a working session in order to be able to repeat this work session at any other time.

By "relevant actions" we mean every message being received or sent by `distr` which has effect on the execution of a process, i.e. every message coming from or going to each process except the *heartbeat* messages, which have no effect on the processes execution.

The re-execution of a working session is only one of the functionalities of the ATLAS journaling. The information recorded in the journal is also useful for other things like the fault-tolerance (a process' state can be recovered by replaying portions of the journal), the ability to make undo's and redo's or the possibility of recovering the last state of the whole application stored in the journal, to keep on with the same work session left and closed at any other time in the past.

From an abstract point of view the ATLAS journaling design consists of two phases:

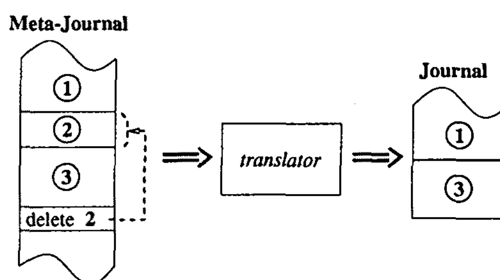


Figure 8.1: Scheme of the two phases of the ATLAS journaling

- the creation of a Meta-journal which also includes orders to modify the journal caused by user editions of it,

- the translation of this Meta-journal that executes these modification orders. This phase needs to check that the resultant journal is consistent. As an example if block 2 in figure 8.1 defines a new data tag referenced in block 3, the `delete` order will not be executed.

ATLAS considers the Meta-journal just as a journal which has also some registers containing instructions that affect other journal registers (Meta-instructions). These Meta-instructions are executed over the Meta-journal to translate it into a journal (without Meta-instructions).

8.1 The journal and its functionalities

8.1.1 Design

Before discussing the design decisions for the ATLAS journaling, we need to review quickly the requirements for this journal, i.e. what are the functionalities that the ATLAS journaling must include.

From an abstract point of view we have seen a design distinguishing between the Meta-journal and the journal because the first one is also a journal but modified by user edition, and this modification must be checked by the system. But the journal itself must support:

- the fault-tolerance to crashes of hosts or communications, being able to recover a process from the start to the status it had when the communication was lost;
- the re-execution of a working session, either to offer a demonstration previously saved to a journal file or to keep on with the working session after an accidental interruption;
- the global identification of input data introduced in chapter 2;
- the definition of process checkpoints, being able to know when it has a checkpoint for a process in order to avoid a total re-execution of the process when it has to be recovered;
- the possibility of doing undo's and redo's of the last user commands without having to edit the journal to do it.

Four levels of information

The design for the ATLAS journal distinguishes four different levels of information to be kept as registers in the journal:

level 0: the commands or instructions introduced by the user. From the user point of view these can be considered "the working session" and most of them will trigger some execution;

level 1: the ATLAS actions or annotations. This information is directly dependent of the previous one because it is triggered by the executions of commands;

level 2: the Meta-instructions, caused by editions of the journal and only relevant to the Meta-journal (see section 8.2);

level 3: the messages produced asynchronously by the system, which are not caused by any command or instruction introduced by the user (for example an execution triggered by a *heartbeat* message through the ATLAS events functionality –see section 6.4).

The third level will be discussed in the Meta-journal section. The other three involve messages arriving to and leaving from the `distr` process. All this information is needed to be able to offer the desired functionalities. The details of how these functionalities work will be explained in section 8.1.3.

Sane points

The ATLAS journal has to keep also information about the state of the input data and requests lists. An ATLAS *sane point* indicates that at this moment of the execution the information of input data and requests that the `distr` process has in its internal structures is not relevant.

The *sane points* are used to avoid problems because of the asynchrony of the arriving of input data and requests when a re-execution of the journal is required to continue its work. This asynchrony can provoke that a re-execution has, at the end, a different order in the internal structures of `distr`. Since the order is important to decide the matching (see section 6.3), this situation can lead to an undesired execution because in fact it is not the same execution as the first one.

This problem can be present in two different situations: when a working session is stopped by the user in order to be continued at some other time, or when the machine where `distr` is being executed crashes. In the second case the situation of the input data and requests structures in the `distr` process is unpredictable.

A *sane point* will be produced each time `distr` notices the structures for input data and requests have no relevant information (they are empty or have data or requests with timeout). There will be a lot of these *sane points* in a journal so they must occupy the minimum possible space.

The use that the journaling does of these *sane points* is on a re-execution. The re-execution goes on until the last *sane point*, where it can assure consistency with the last execution of this journal. The rest of the journal registers after this *sane point* can be reexecuted or not, depending on the decision of the user.

Different linkages for a register

The journal is directly managed by `distr`, so that the algorithms handling it should be as efficient as possible to avoid an excessive amount of working time for `distr`, and consequent delays in the execution.

In order to make these more agile, ATLAS keeps several different linkages among the registers in the journal:

Timing order linkage: the order of registers in the journal is the order in which they arrived. Each register has a linkage with the previous one registered in the journal and will have another with the following one.
This timing order linkage is the most important because it gives the order of the exchange of messages with the processes, and is also the most easy to maintain.

Command level linkage: this is the linkage among the registers of level 0. Each register of level 0 has a linkage with the following register which is also of level 0.
The command level linkage gives the order of the commands input by the user. In some way it is the working session viewed by the user. This linkage is important for the re-execution functionality in a demonstration because these registers are those to be reexecuted. It will be also useful to allow an edition of the Meta-journal because these registers are those that can be modified by a user edition (see section 8.2.)

Execution dependent linkage: each register of level 1 depends on a register of level 0 which is the one provoking an execution that can involve many other registers of level 1. This linkage between a register of level 0 and the registers of level 1 caused by the execution of the first one is the execution dependent linkage. The time order among them is always maintained.
The execution dependent linkage gives the order of all actions caused by a given command execution.

Process linkage: for each process there is a linkage joining those registers associated to this process (representing messages being sent and received through the communications channel with this process). The process linkage gives the order of the registers related to a given process. This linkage is very useful for the process recovering functionality because it groups all the registers that are needed to recover the process.

The maintenance of these linkages must be done each time a new register is going to be stored into the journal. To do this efficiently, `distr` must keep some extra information to be able to know quickly which are the registers on the journal that have to link with this new register (this will be explained at length in section 8.1.2).

ATLAS messages need to be enriched with adequate information to allow `distr` to properly construct all these linkages. First of all the execution flow of a command in the system must be recognized. The execution of a command can cause many interchanges of messages among processes because of different external routine executions. Since ATLAS is totally asynchronous in message receiving (`distr` never waits for a specific message, it treats them when they arrive), we need some kind of identification of these messages to know which command execution they are depending on.

In order to establish this relation among messages and the command that caused them, ATLAS uses the *execution thread identifier* which is part of each

ATLAS message. This identifier is unique for each command execution and will be the same for all messages related to this execution thread [1]. The management of this identifier by the different ATLAS components is as follows:

- *In the `distr` process:* The messages arriving to `distr` with a thread identifier have to propagate this identifier to everything they produce. In cases that the message arriving did not have a thread identifier yet (when it is a command of level 0), `distr` gives it a new thread identifier which will identify everything related to this command.
- *In the process driver:* Each time a new thread identifier arrives with a message it changes the present thread in the driver and the new thread becomes active. This active thread causes each message produced by the process to have this thread as its thread identifier, so everything caused by this execution thread will be marked as belonging to this thread by the communications driver.
- *In the `Command Subsystem`:* Since the `Command Subsystem` is formed by three different processes with communications between them (see chapter 4), the treatment it has to do with the execution thread identifier is quite different. The execution of an ATLAS command is managed by the Virtual Machine but is guided by the compiler, so any new execution arrives to the Virtual Machine from the compiler.

  When a message arrives to the `Command Subsystem` driver it already has its own execution thread identifier. In case this message is an ATL command introduced by the user, it can produce an execution (it is an ATL sentence out of any function or procedure definition). The driver then passes this thread identifier to the compiler in order to pass it to the Virtual Machine when the compiler sends to the Virtual Machine a new execution request. In the compiler the thread will continue to be valid:

  - until the execution order is given to the Virtual Machine, in case the command was a direct execution order;
  - or until the file compilation finishes, in case the command was a "`USE namefile`" command, because in this case there can be more than one executions related to this thread identifier.

  In the Virtual Machine the thread identifier received from the compiler for an execution is associated to the `ExecStep` of this execution and inherited when an asynchronous call creates another `ExecStep` (see sections 4.2.2 and 4.3). Each message generated by the Virtual Machine to go to `distr` will use the thread identifier associated to its own `ExecStep`.

There are two special cases in the management of the thread identifier: the ATLAS events messages and the input data messages.

[1] Although it is not exactly a low level thread, it means the same in an abstract view. That is why we called it execution thread.

- The ATLAS events messages must contain the thread identifier of the execution that caused this event. The only event messages relevant to the journal are those that are sent to a process, i.e. at least one process is subscribed to the event. The execution triggered by the event message sent will also belong to the same execution thread.

  In case the message causing the event is sent directly by the user, it belongs to level 0 so a new thread identifier will be assigned to the execution this event triggers.

  When the message is caused asynchronously by the system, it belongs to level 3 and a new thread identifier is also assigned to the triggered execution.

- An input data is not really effective until it is matched to a data request (see section 6.3). The thread identifier for this datum then must be the same as the one identifying the request, because the data is going to become part of that execution thread.

Deciding when a message is a direct consequence of a user action (and therefore belongs in level 0) or not is not always simple.

In a window-driven user interface and specially in computer graphics applications, the user input data is produced in a window (X-window, for example) and managed as window events. This events are usually managed by the process that creates the window where they are produced. An ATLAS application process can also manage these events by adding a connection to the X server to the process driver as a new channel to be listened to (an example for X-windows events management is presented in the "`demandes`" utility process – section 9.3.2). In this case `distr` is not aware of which are the messages directly input by the user because the interface with the user is managed by an external process.

The only difference the process' communications driver can use to decide something is produced at level 0 (by the user) is the channel through which the message is received. Usually an external routine call (in the process that implements it) is synchronous, therefore everything produced by it will belong to the corresponding execution thread, it will not be at level 0. But something arriving to the process driver through a channel not being the connection with `distr` can be considered independent of an execution order coming from `distr`.

The process driver then will assign level 0 to any message coming through a channel not being the connection with `distr`.

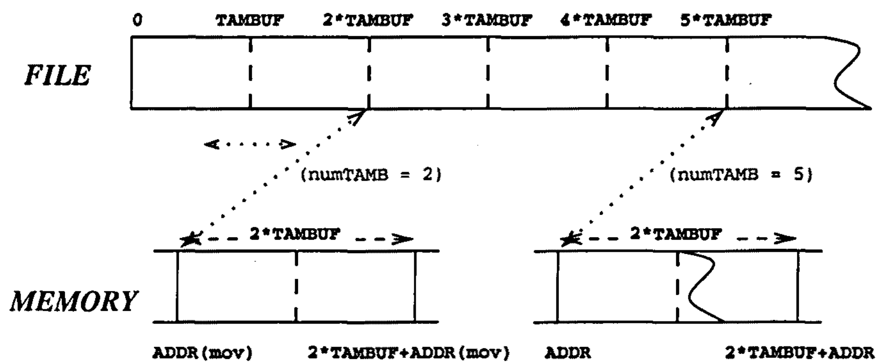
It is not true that everything a process driver decides is of level 0 can only be produced by the user directly. As an example we can think on a process that forks when a certain routine is called and maintains a communication with its son until it finishes its work. The messages coming through this channel to the driver could be considered as a production of the routine call which causes the process to fork. However we consider that in such cases the asynchrony produced by the sub-process is desired by the developer, so those messages going to `distr` which are produced during the execution of that channel's `handle_input()` method are also considered of level 0.

8.1.2 Some implementation details

As we have seen in chapter 6, `distr` is the most relevant component of the system. It is also responsible for the journaling facilities, so the internal management of the journal must be as efficient as possible in order to not disturb the execution of the rest of the application. Moreover the journal structure is required to be persistent in order to survive to possible crashes of the machine where `distr` is running. These two requirements make it necessary to devise a combination of persistent physical structure and quick updating management.

To solve these problems, the ATLAS approach uses a structure based on a file where the addresses are always referring to the file addresses and the linkages are also absolute addresses of the file. Each register knows its own length. The last part of this file is mapped in memory in order to have a much faster access to it (specially for the addition of registers which is the most common activity), and the operating system handles the updating to the file automatically. When the memory mapping is full, it is unmapped and mapped again from a later position of the journal.

But besides the recording of registers, the journaling management many times requires an access to positions other than the last one, and sometimes these access are to positions far from the last one, so some of these positions asked to be accessed may not be in the mapped area. To solve these cases, ATLAS also maps another area of the file in order to have access to positions far from the last one; this second mapping (the mobile mapping) will be changed from one area to another each time the access required cannot be solved in the first mapping area (of course these cases will not be numerous). Figure 8.2 describes this management graphically and also includes the formulas to obtain a file address from a memory address and backwards.



@file = @mem - ADDR + numTAMB * TAMBUF

@mem = ADDR + @file - numTAMB * TAMBUF

A register is in memory when: @file div TAMBUF >= numTAMB

Figure 8.2: Internal structure to manage the journal

The internal management maps the n*TAMBUF file address in memory taking 2*TAMBUF space. When the second TAMBUF block in memory is almost full, the memory is unmapped and mapped again from the next block ((n+1)*TAMBUF), and so on.

The TAMBUF length must be adequate to avoid an excessive number of accesses to the mobile mapping. This length is configurable in the system in order to allow fine tuning.

The register structure

According to the journal structure already described, a register must be a part of the journal file. As we already said the linkages among registers will be implemented also as absolute addresses to the file. A register is implemented internally as a stream of bytes and its contents are:

- the length of the register;
- pointers to other registers to maintain the 4 linkages (see section 8.1.1);
- the thread identifier of the register;
- the process identifier of the register;
- a byte containing several different information like:
  - level (requires 2 bits),
  - in/out (from/to the process),
  - *sane point*/no,
  - active/no.
- and finally the specific content of the register, i.e. the message sent or received by `distr`.

The flag saying if the register is active or not is used by the editions of the journal (see section 8.2).

The flag indicating if it is a *sane point* in fact indicates whether after this register `distr` has marked the situation as a *sane point* or not. This implementation of the *sane point* (see section 8.1.1) is adequate to waste the least possible space on it.

The Journal class

The Journal class is the wrapper of the structure described to manage the journal. It also offers some important methods to be mentioned:

- to add a register, it has to maintain all linkages (see "Maintenance of linkages" below);
- to access to elements of a list (first, next, last) which are able to act over all linkages (which one depends on their parameters);

- to un-do and re-do one or more user actions (see subsection 8.1.3);
- to ask if there is a checkpoint for a given process name;
- to substitute a global input identifier for another (see subsection 8.1.3);
- or to add a Meta-instruction (see section 8.2).

Maintenance of linkages

The addition of registers to the journal is obviously the most common work that *distr* has to do related to the journal. This work is not easy because the linkages among registers described in section 8.1.1 must be also maintained at this time.

In order to facilitate this maintenance when a register has to be added, ATLAS keeps some extra information in some of its structures:

- The *timing order linkage* and the *command level linkage* are unique in the journal. ATLAS keeps the `first` and `last` pointers to these linkages into the Journal class, so when a new register is going to be the next on these linkages, the management accesses directly the `last` one to modify its link to next and updates the `last`.
- There is a *process linkage* for each process taking part on the application. In order to maintain these linkages ATLAS keeps the `first` and `last` pointers to these linkages with the corresponding process information, so for each active process in the application execution a register adding to the linkage does the same as the other two linkages explained above but for the corresponding process.
- For the *execution dependent linkage* the information of `first` and `last` of all these linkages cannot be kept. Each user command triggering an ATLAS execution thread creates a new execution dependent linkage and the number cannot be limited. ATLAS is not able to determine when a thread has ended and will produce no more registers.

  Since all this information cannot be kept, but is needed in order to have some efficiency for the maintenance of these linkages, we decided to keep only a certain number of these linkages. In fact the `N` last referenced threads (where `N` is configurable) will be kept in a special structure that allows a direct access to a given thread linkage information. The management is:

  - the structure keeps `N` thread linkage informations;
  - these `N` are the last referenced in the execution;
  - when a new register is created with its corresponding thread identifier, ATLAS tries to access directly to the linkage information by accessing the structure;
  - if the thread found in the position accessed is not the one being looked for, it means this thread must be searched in the journal and the structure must be updated leaving out the one which spent more time without being referenced.

The fourth step is, of course, the most costly, so the `N` configurable parameter must be big enough to cover almost all threads active at any time but not spending an excessive amount of memory.

8.1.3 The journal functionalities

Re-execution for a demonstration

The relevant information in the journal for the re-execution functionality is the registers of level 0 (commands input by the user) and level 3 (asynchronous system messages). The input data matchings will be used to repeat the same input for the same request.

The `distr` process is executed in a special mode "REEXEC" that causes it to act differently in the following cases:

- it will execute only those registers of levels 0 or 3 in the journal being re-executed;
- it won't take into account anything coming at level 0 nor input data from the user;
- it will manage the *heartbeat* messages from the processes because it has to control the execution of them, but this management will not produce new event messages because they are already registered in the journal at level 3 so they will be also re-executed;
- it will check if the messages coming from the processes are the same that those registered in the journal in order to control that processes are deterministic;
- finally, it will not generate another journal.

The only treatment `distr` will do the same way as a normal execution is the broadcast done by `distr` to know which server daemons are able to execute ATLAS processes (see chapter 6).

The determinism of a process is checked only with respect to ATLAS (i.e. for the same messages being sent to a process in two different executions, they produce the same messages back to `distr` and in the same order). When a process is indeterministic with respect to ATLAS, `distr` may detect it and give a warning message to the user in order to let him know of this situation.

Checkpoints

The recovering of a process with state is much more efficient and quick if the process maintains checkpoints periodically. A checkpoint in a process keeps the important information for the process state at the checkpoint time, so it facilitates the recovery of this state of the process at any other time.

The ATLAS checkpoints functionality offers to the developer several possibilities in order to be more flexible in its use. These possibilities for a process are:

- *never do any checkpoint:* this is the case for a process without state or for a process which wants to be totally re-executed when it must be recovered.
- *do a checkpoint each time* ATLAS *advises to do it:* in this case the process does not need a special treatment to provoke the checkpoint. There are also two possible treatments for this case:
  - to implement the desired checkpoint for this process,
  - to take the ATLAS default checkpoint as the process checkpoint.
- *do a checkpoint when the process decides:* this possibility requires a specific treatment to control when and how the checkpoint must be provoked.

Since ATLAS does not know anything about the internals of processes, the only way it may offer a default checkpoint is by dumping the memory of the process in a file in order to undump it later to recover the state.

The advise `distr` sends to processes to provoke a checkpoint is generated when `distr` is in a *sane point* (see section 8.1.1) and the journal has grown over a fixed threshold (the larger number of registers in the journal, the more expensive re-execution of it will be).

When a process does a checkpoint it must notify it to `distr` in order that the checkpoint be saved in the journal to be used in case of recovery of the process. This notification is done by using the routine "`atl_checkpoint`" of the API library (see section 9.2) which sends to `distr` the name of the file saving the checkpoint and also the name of the routine to be started to recover the checkpoint. This routine must have always the same signature. Its parameter will contain the name of the file where the checkpoint was saved.

Processes recovery

One process recovery

In a process recovery ATLAS acts differently depending on the process recoverability:

- a **process without state** just must be re-started without recording in the journal its death nor its re-starting.
- a **process with state** must be recovered and/or re-executed.

  First of all `distr` must be put in a special mode, RECOVER-PROC, which causes it to act differently with respect to the process being recovered than with respect to the rest of the application.

  For the process being recovered, `distr` re-starts it and finds the last checkpoint of this process in the journal. In case a checkpoint is found it instructs the process to recover this checkpoint, passing the file name where the checkpoint is saved to it, and puts the re-execution pointer to the next register in the journal for this process (linked by the process linkage – see section 8.1.1).

If there was not a checkpoint the re-execution pointer in the journal will be the first register for this process.

In both cases the recovering follows by re-executing the rest of the process, i.e. following the process linkage in the journal registers, re-sending to the process all messages sent before and for each message being received from the process, checking if they are the same than the process sent before (in order to check the determinism of the process).

For the rest of the application processes, `distr` does not send any message to them because the execution is supposed to be stopped, but the processes are not really stopped, so they can keep on sending messages to `distr`, in fact they also send the *heartbeat* messages as usually. `distr` does not have to treat other messages coming from processes while it is in recovering mode, but it has to treat the *heartbeat* messages in order to control other possible failures. The other messages, then, are stored in memory in order to be treated when the recovering finishes.

Once the recovering is finished `distr` has to treat all messages stored in memory before continuing the normal execution. In this case it also has to treat the *heartbeat* messages coming asynchronously but now they can be treated normally (triggering an execution in case they have to –see ATLAS events in section 6.4).

Before the decision of recovering a process or not, `distr` needs to have information about the process saying if it has state or not. Since this information is only known by the developer, in case the process does not need to be recovered (it has not state), the process implementation has to notify it to `distr` by using a routine "`atl_dontrecover()`" offered in the API library (see section 9.2) which indicates this process does not need to be recovered.

`distr` controls that each process is running normally with the *heartbeat* mechanism (see section 6.1.2). When `distr` detects a process is not sending *heartbeat* messages it starts the recovery of the process by checking if the process must be recovered or not. In case the process must be recovered (because it has not send the "dontrecover" message), `distr` will ask the user if the recovering must be started. This allows the user to decide if the recovering is necessary or not and can avoid the recovering of processes having errors at developing time. If the user says the process does not need to be recovered, `distr` stores an exit message for this process in the journal in order that a total re-execution of this journal kills the process at this moment in the application execution.

Total recovery

A total recovery implies a re-execution of the journal not for a demonstration but only to recover the state of the system in the journal, and to continue with its execution. The quickest way to do this is by recovering all processes, including `distr`, from the information kept in the journal.

First of all `distr` has to start the system and do the broadcast as usual in order to have the information needed to determine where each process can be executed.

In a total recovery `distr` acts in a special mode, RECOVER, which causes the following:

- `distr` considers itself in the last *sane point* registered in the journal;
- for each process active in the journal at this *sane point* `distr` applies the recovery of a process at this point, i.e. goes to the last checkpoint before the *sane point* (if it exists) and re-sends all messages until the execution of this process reaches the *sane point* in the journal. In this case `distr` has to update the process information as needed in order to be ready to continue;
- once all processes have reached the *sane point*, `distr` will ask the user if he/she wants to continue the execution until the end of the journal or he/she wants to discard the registers in the journal from the *sane point* on. If the user wants to continue, `distr` will re-execute all registers of level 0 and 3 from the *sane point* to the end of the journal;
- finally `distr` will change its mode to a normal execution and will listen for new commands.

Global data identification

Giving a global identification to input data favours a consistent management of graphical input data and a generic use of these data for constraint solving processes (both of them special requirements for computer graphics applications –see chapter 2).

As introduced in chapter 2, this functionality is based on a very simple idea. Input data are attached a unique tag used to identify that datum globally, and both tag and datum are recorded in the *journal* together. (A similar mechanism but with a different meaning and motivation is proposed in [36]. Tags are used there to define topologic relationships in order to be able to compute the results in a model where its parameters have been changed).

The ATLAS API library (described in section 9.2) offers some routines which help the developer to use this functionality in the easiest way. For example, an application process can ask for some of these tags (new tags) to be attached to its own data (`atl_get_ticket`), or the process receiving this input datum can request that an annotation be made in the *journal* that his interpretation of that datum corresponds to some other (previously obtained) datum (`atl_substitute_ticket(t1,t2)`). This new tag will be used in replays of the *journal* instead of the datum, speeding up replays and making them more robust. The effort the process must do to tell the system (`distr`) this association of tags is minimum, and also minimum is the information that goes through the network.

This simple global identification achieves the required robustness in the re-execution of the *journal*, and is also useful internally to control the consistency of the *journal* after being edited and modified.

The conceptual idea for the wrapper classes interface to manage this tag association to input data can be seen in figure 8.3. The actual implementation in fact, is a little bit different in order to deal with a trade-off between the desired transparency to the developer and the conceptual idea (see below for a more detailed explanation of this).

```
class io_base {
  protected:
    atl_tkt ticket;
    String typename;
    int referencies;
    bool containsdata;
    void canviticket (atl_tkt tck) { ticket = tck; }
  public:
    io_base (atl_tkt tck, String n): ticket(tck), nom(n)
      { referencies = 1; containsdata = false; }
    io_base& operator = (const io_base &iob);
    void afegir_referencia ();
    virtual void eliminar_referencia ();
    atl_tkt Ticket const () { return ticket; }
    String Typename const () { return typename; }
    bool Containsdata () const { return containsdata; }
    virtual ~io_base ();
};

template <class T>
class io : public io_base {
    T dades;
  protected:
    String atl_get_type_name ();
  public:
    io (T d) : dades(d)
      { ticket = atl_get_ticket(); typename = atl_get_type_name();
        containsdata = true; }
    io (T d, atl_tkt tck) : dades(d)
      { ticket = tck; typename = atl_get_type_name(); containsdata = true; }
    void eliminar_referencia ();
    operator T&() { return dades; }
    ~io ()
};
```

Figure 8.3: Wrapper classes interface for ATLAS tag–datum association

Although this tag is included in the ATLAS `Variable` structure to go through the network (see section 7.4), the *io* design is needed by the driver of each application process in order to offer this "tag–datum" association to the developer in an easy way of use. This design is also used by the Virtual Machine component of the Command Subsystem (described in section 4.2). It makes possible this tag association goes through a data requested in the ATL code being sent to an external routine as a parameter.

From the developer point of view, a process can use this association if it is worth for it, or can elude its use by doing nothing on it. The simplicity of using this and also the achieved level of transparency for the developer can be shown through a simple example:

Suppose we want to implement a routine to select an edge from a 3D point (if this point is near the edge). If we design the classes *"Point3D"* and *"Edge"* as in figure 8.4 and considering we have an array of edges as:

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

the routine `do_anything` (figure 8.5) can be implemented taking a *"Point3D"* as a parameter and searching for the corresponding edge in the array. If the corresponding edge is found, the routine asks the system to make an annotation in the *journal* to replace the *"Point3D"* tag for the selected *"Edge"* tag for future re-executions.

Since the `do_anything` routine asks for a substitution of the tag passed to

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

Figure 8.4: Interfaces of classes "Point3D" and "Edge"

```
1       void do_anything (io<Point3D> &input)
2       {
3         int i=0;
4         while ((i<N) && (!near_edge (input, Table[i])))
5           { i++; }
6         if (i<N)
7           { atl_substitute_ticket (input.Ticket(), Table[i].Ticket());
8             Compute_with_edge (Table[i]);
9         else error ("there aren't edges near that point");
10      }
```

Figure 8.5: Routine searching the edge to compute with it

it (line 7 in figure 8.5), in a re-execution of this journal the routine will receive just an *"Edge"* tag as a parameter. The routine `do_anything` then must be overloaded to accept also this tag as a parameter:

```
void do_anything (io_base &id1)
  { Compute_with_edge (search_for_edge(id1)); }
```

Moreover, this management can be even more flexible if we provide a routine with a generic parameter, which can be used also passing an edge if the application allowed the user to input edges. (figure 8.6)

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

Figure 8.6: Routine with a generic parameter

The actual usefulness of this `generic` routine requires the possibility of declaring external routines in the ATL module with overloading in its parameters (accepting a *"Point3D"* or an *"Edge"*). This is an easy extension to the language which is planned to be done as soon as possible in the current prototype (see chapter 10), but is not yet available.

The actual classes implementation

As we already introduced the wrapper classes in figure 8.3 are not the structures actually used in the implementation. The actual structure design is shown in figure 8.7 where the io_base class is not the base class for the io class, even though it has the same information and the same role than explained before.

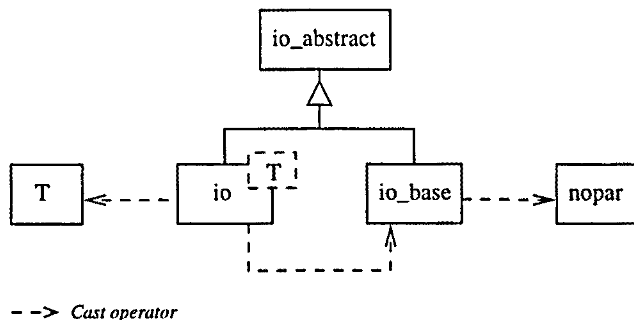


Figure 8.7: Actual design for the "tag-datum" implementation.

In the actual design used in the implementation there is an added abstract class io_abstract in order that an io is not an io_base directly. The conceptual idea, however, is the same and the user can see this in the same way.

The reason for this change in the implementation comes from the automatic code generation (see generated code in section 7.5.1). On one hand the developer is not requested to use this tags (desired transparency for the developer), so he can use directly the process' types for his routines. This is possible and transparent by having the operator T() defined as a method in io<T>. On the other hand the developer is also allowed to substitute tags for any parameter of a routine, causing a re-execution use only an io_base as a value for that parameter. The generated code, thus, must cover all combinations of io<T> and io_base for the parameters of a routine. Some added stubs must be generated also to allow the compilation of the generated code covering all these combinations (see section 7.5.1 and also a complete example in section 9.4). But, if the io<T> class derives from io_base, the matching from io<T> to io_base has higher priority than the one from io<T> to T, so the code would execute the stub routine instead of the user's one even the value passed for the parameter is an io<T>. The dissociation of classes io<T> and io_base added to the fact that the routines in stubs ask for a nopar parameter type which is only reachable from io_base and not from io<T> solves the problem and does not interfere in the conceptual idea of the management.

UNDO and REDO possibilities

The idea of having the UNDO and REDO possibilities is directly related to the execution because they must have an immediate effect. Since the journal is editable and changeable, the user can also remove or add blocks of registers by using the Meta-journal facilities (see section 8.2), but these are not interactive,

so the UNDO and REDO are the interactive commands to do the same at the end of the journal (acting on the last user actions).

From an abstract point of view the UNDO and REDO actions just have to cause a movement, back and forward, of the execution pointer in the journal. They only act over the user actions. In fact a REDO can only be done if at least one UNDO have been done before. In order to make more flexible the use of UNDO and also more efficient in cases that the user wants to un-do more than one action, the UNDO command can have a parameter indicating the number of actions the user wants to un-do, so the journal goes directly to the last point where the UNDO guides it to.

Since the UNDO action can be very costly in time, ATLAS has different possibilities to do this trying to reach the cheapest one. These are:

- first it checks if there is an inverse function for the action to un-do; this would be positive because it avoids the recovery of the process;
- if there is no inverse function, it must recover those processes taking part on the action to un-do (looking for their checkpoints and recovering the processes from them, or from their starting point). This recovery goes until the action before the one to un-do.

An inverse function is a function or procedure, implemented by the developer, which does just the contrary in execution than the other function ('a' is inverse of 'b') achieving the same state in the process as if it had not executed 'b' in the first place.

An inverse function can only be defined by the developer because he/she is the only one knowing the semantics of the function. As it has been explained in the ATL language definition (section 4.1.7), an inverse function (which is also a command) must be declared in the ATL module saying that it can be used to invert a command. `distr` keeps this information when the module is compiled and when the command having inverse must be un-done, `distr` knows the inverse function and can call it instead of recovering the process (a more expensive operation).

Internally, the work to be done by `distr` in case the user requests an UNDO action is:

- If there is an inverse function for the command, `distr` orders the execution of this inverse function;
- If there is no inverse function, `distr` has to find out the processes taking part on this command in order to know which processes must un-do their last actions; after this it has to recover (with or without checkpoints) the state of these processes before the last user action.

A REDO action only makes sense after an UNDO action is done, and it will re-execute the last user action being un-done.

8.2 Meta-journal

As we already said in the introduction of this chapter, ATLAS considers the Meta-journal as a journal having also Meta-instructions which are orders to modify the journal.

The only modification orders allowed in a Meta-instruction are: insertion, remove or move, and can be applied to a register or an interval of registers in the journal. The registers that can be directly affected by a Meta-instruction are those belonging to level 0. This is because a Meta-instruction is produced by an edition of the Meta-journal done by the user, so he is only allowed to modify the user commands (level 0).

A *Meta-instruction* then consists of a modification instruction (insertion, remove or move) and the reference to a register or an interval of registers of level 0.

Since the Meta-instructions must be introduced by the user and this requires the existence of a user interface, ATLAS offers a specific process to manage this user interface in order to relieve *distr* of this work. This specific process is an ATLAS process that the user can start by introducing the command: "`USE MJEditor`" as done with any other ATLAS module.

Although the `MJEditor` process is the one that implements the user interface and listens to the user requests to insert Meta-instructions, only `distr` is responsible of the journal and the Meta-journal. In order that `MJEditor` can have some effect over the Meta-journal, there are several routines offered as a journal specific API that an ATLAS process (like `MJEditor`) can use to ask `distr` to do a certain Meta-journal operation. These routines allow:

- registering a Meta-instruction in the Meta-journal;
- translating the Meta-journal to the journal;
- checking the consistency of the journal after modifications.

Meta-instructions have to be produced in an edition of the Meta-journal, but the user can do more than one modification in the same edition. We define a *Meta-edition* as a set of Meta-instructions which is required to preserve the first level of consistency of the journal. This first level of consistency is checked interactively when an edition is finished (see section 8.2.2 for more details on this).

The Meta-editions (and also the Meta-instructions) are created and registered in the Meta-journal on the first phase of the ATLAS journaling design (explained in the introduction of this chapter), therefore they have no effect on the application execution until the second phase, the translation, is done. In other words, when the user edits the Meta-journal and changes it, these changes will not produce any change in the execution until the user explicitly requests the translation of the Meta-journal to the journal.

8.2.1 Translation Meta-journal → journal

The translation of the Meta-journal to build a journal which has the changes ordered by the Meta-instructions must be requested explicitly by the user. This translation means the application execution has to go back and forward again in order to take away the effect of commands that the user wants to remove (Meta-instructions ordering remove), or to execute commands that the user wants to insert (Meta-instructions ordering insertion or move).

The whole translation consists of two phases:

- The first one is a checking phase. In this phase the final consistency of the journal must be checked. The consistency checking (explained in the next section) is done before the real translation to avoid, if possible, processing erroneous editions.
- The second phase is only executed if the first one has had a positive result. In this case this phase is the actual translation or evaluation of the Meta-journal, i.e. the inclusion of the Meta-instruction orders in the current application execution.

  To do this translation, `distr` uses the journal functionalities explained in section 8.1.3 combining them as follows (shown also in figure 8.8):

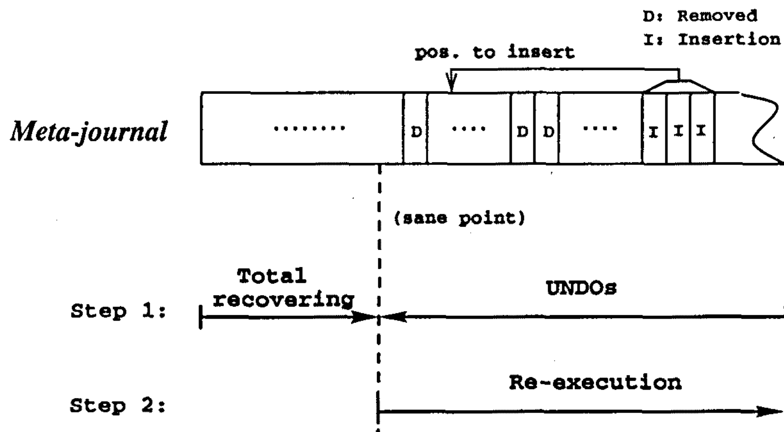


Figure 8.8: Translation process from Meta-journal to journal.

  - The first step is to go back in the execution to the last *sane point* before the first register modified by the Meta-instructions. To do this the UNDO functionality can be used, so in some cases the execution can go back through the inverse functions and in other cases it must recover the status of the processes at the decided point. This must be done also for `distr`, so the recovering will be a *total recovering*.
  - The second step starts at that *sane point* and must *re-execute* the rest of the journal considering those changes made by the Meta-instructions (inserting registers in the right position and not considering registers marked to be removed).

Those registers marked to be removed will not be executed in the translation, but they will not be removed from the journal file either. This is to facilitate the user to un-do a removing done before even though the translation has been done.

8.2.2 Consistency checking

The consistency of the journal can be altered at two different levels: one is the *identifiers* level and the other is the *ATL definitions* level.

The ***identifiers*** level is affected by changes on the registers containing some atl_ticket (identifier given to input data –see also subsection 8.1.3) or containing a request identifier (given by *distr* to any data input request produced in the system). The first case can create inconsistencies if the declaration point of the atl_ticket is removed and there are other registers in the journal making some reference to this atl_ticket. The use of the atl_tickets is usually managed by the application process, so cases where an atl_ticket is used more than once in an execution can be very common. The second case is easier because a request identifier is created in the same execution thread that uses it (matching it with an input datum). Here the only possible inconsistency is produced when before the matching happens this request is re-ordered in the list of requests, because this re-ordering command is also referring to its identifier.

To solve this level of consistency checking, in the first case, `distr` looks at the Meta-instruction being evaluated and if it affects a register declaring an atl_ticket and the Meta-instruction order is a remove or move order, `distr` checks the rest of the journal looking whether any of the registers recorded have some reference to this atl_ticket that must be removed or moved in the journal. If this case is found, `distr` has to notify to the user there is an inconsistency and not take into account the Meta-edition causing it. In the second case, if the register causing a data input request is going to be removed or moved, `distr` just have to look in the middle of the two references to this request if there is a re-ordering command for this request. This inconsistency can be solved by just removing also the re-ordering command if the Meta-instruction is to remove, or moving the re-ordering command together with the registers in the Meta-instruction if this is to move.

This *identifiers* level of consistency checking can be done interactively because `distr` can manage it just by looking in the journal (it does not need any other ATLAS component to check it). This consistency checking is done then each time the user wants to record a Meta-edition in the journal even though the Meta-journal is not going to be translated yet. This is called then the first level of consistency.

The ***ATL definitions*** level is concerned with the ATL declarations or their order, because these changes can cause a re-execution of these ATL commands, which when sent to the `Command Subsystem` may have compilation errors because of undefinitions. A clear example can be the removing of a `USE` sentence of a module when there are still commands using exported entities of this module (see figure 8.9).

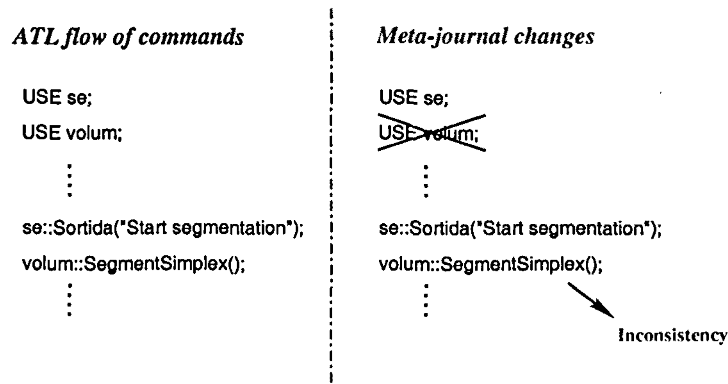


Figure 8.9: Example of an inconsistency at ATL definitions level.

This kind of inconsistencies cannot be noticed by `distr` itself, because they need the symbols table of the `Command Subsystem`. To do this level of consistency checking, `distr` requires some help from the ATL compiler.

The consistency at this level is only checked by `distr` when the translation of the Meta-journal is required (at the first phase of this translation –section 8.2.1). To do this checking, `distr` starts an auxiliary process, `AtlasCompAux`, which has the ATL compiler. Next, `distr` sends to it all the commands to be parsed by the ATL compiler, and the `AtlasCompAux` process compiles them and notifies to `distr` if there is any error in compilation. If there is a compiling error the consistency is not preserved and the user must be notified of it.

The `AtlasCompAux` process is an ATLAS process that executes the ATL compiler without producing intermediate code. It has as input those commands sent by `distr` and causes the error output to be sent also to `distr`.

This process, like the `Command Subsystem`, does not have an associated ATL module. It is started by `distr` only to check the ATL definitions consistency and finishes when it is done. Figure 8.10 shows the scheme of the `AtlasCompAux` process and its communication channels.

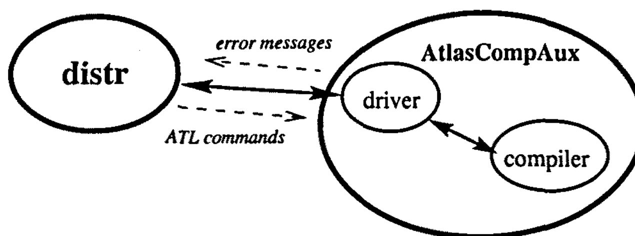


Figure 8.10: The `AtlasCompAux` process and its communications.

8.2.3 The journal API offered by `distr`

The small API offered by `distr` to be used by the editor of the journal consists of three routines:

- "void **atl_jour_addMetaInst** (MetaInstruction &mi, Comunic_Distr &d);"

  This routine asks `distr` to insert the Meta-instruction received as a parameter in the Meta-journal. The MetaInstruction `'mi'` is the first parameter of the routine.

  The second parameter is 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).

- "bool **atl_jour_translation** (Comunic_Distr &d);"

  This routine orders the translation of the Meta-journal to a journal. It does not need any parameter, but it can have the communication channel which is optional like in the routine `atl_jour_addMetaInst`.

  At any time `distr` keeps information about the current level of consistency of the Meta-journal. When a translation order arrives, if the current level of consistency indicates that some consistency level has not been checked after a change in the Meta-journal, `distr` automatically checks the consistency at that level before translating the Meta-journal.

  The return value indicates whether the translation succeeded or not, i.e. if the Meta-journal has been translated or not.

- "bool **atl_jour_consistcheck** (int level, Comunic_Distr &d);"

  This routine orders a consistency checking of the Meta-journal.

  The first parameter is optional and indicates the level of the consistency to be checked. The correspondent values are:

  - level = 0: both levels identifiers and ATL definitions must be checked. This is the default value.
  - level = 1: only the first level, identifiers, must be checked.
  - level = 2: only the second level, ATL definitions, must be checked.

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

  The return value is a boolean indicating whether the checking succeeded or not.

8.2.4 The `MJEditor`

The `MJEditor` is the editor process offered by ATLAS as a utility process to do the edition of the Meta-journal.

The current version of this process in the ATLAS prototype is not yet what we require for this process to be actually useful. It has been implemented just to test the Meta-journal facilities in ATLAS, so it is able to call the journal API routines but without having a friendly user interface that facilitates this work to the user. Nonetheless, we want to enumerate here the requirements this process should satisfy to be useful.

First of all, it requires a user interface able to show a compact vision of the journal so the user can easily identify in this vision the commands (level 0 registers) that he can deal with. It should also allow the user to select one or more of these commands and choose one of the permitted Meta-instruction orders (insertion, remove or move). A textual input to permit the insertion of new commands by the user must be also allowed.

The editor is also responsible of dealing with Meta-editions. It has to send the Meta-instructions taking part on the Meta-edition to `distr` and it also should ask for the consistency checking at the identifiers level when all the Meta-instructions of the Meta-edition have been sent.

It also should offer to the user the possibility of requesting the translation of the Meta-journal when he wants to do it.