The Observer: A tool for observing Object-Oriented Distributed Applications

C. Roisin, M. Santana

Unité Mixte Bull-IMAG, 2 avenue de Vignate, 38610 Gières, France

e-mail: santana@bounty.imag.fr

1 Introduction

An important challenge in the area of system development is to give the users a maximum of facilities for programming, testing and running their applications. Above the kernel of a system, many system services, such as editors, compilers, browsers, debuggers, graphical tools, etc, may be developed [7], [17]. With the advent of low-priced high resolution bit-mapped displays, it is possible to have more natural and visual user interfaces (with not only text but also graphics) for all these environment tools [1].

This issue is more important in a distributed environment because parallelism, remote execution and object sharing are more complex to manage. The user has to understand the basic functionalities of the system in order to be able to use them correctly and efficiently, as well as to fully master the behavior of his applications: distribution management, computational model, resource sharing, concurrency control, etc.

On the other hand, it has always been difficult to present a convincing demonstration of the inner working of an operating system, a system service, or a run-time environment, since the main quality of such software is to be as invisible as possible. This situation is even worse in a distributed object-oriented systems, because:

- objects provide a high level of abstraction, which tends to hide the details of the operations;
- location transparency tends to hide the distributed nature of the computations.

In order to enhance the demonstration of applications in an object-oriented distributed environment, we have designed and implemented the Observer, a graphical tool for the observation of the execution of application programs. The main goal of this tool is to demonstrate and to explain the inner working of an application, i.e. to show how the different entities which compose an application evolve and interact during the execution. The Observer runs on top of the Guide system [2], an object-oriented distributed system developed in our laboratory; it is therefore targeted towards the visualization of object-oriented distributed applications.

This paper is organized as follows. Section 2 precisely defines the objectives of the Observer and describes the context in which it has been developed. Section 3 describes the operation of the Observer: the execution of several different applications is used as the basis for an illustration of the visualization facilities. Section 4 is a discussion about some implementation issues. Perspectives and evolution of the tool are presented in Section 5.

2 An observation tool in a distributed environment

The observation of an application consists essentially in the visualization of its component processes and data and their evolution during its execution. Three kinds of motivations for such a visualization tool can be identified:

- As a teaching tool to facilitate the understanding of algorithms or basic mechanisms: animation algorithms as in Balsa [4], method invocations in an object-oriented language as in GraphTrace [10], concurrency and synchronization constraints in parallel languages such as Ada [6], etc.
- As a debugging tool in order to allow graphical debugging and process execution analysis, such as the tracer tool Trick [3], the Ada debugger described in [6], Pie [13], a Programming and Instrumentation Environment supporting performance visualization, Provide [16], a Process Visualization and Debugging Environment allowing deferred-binding pictures for program animation, or Voyeur [19], which provides graphical views of parallel programs.

- As a demonstration tool allowing the application designers to easily show how their applications run and how they use the mechanisms of the supporting system.

This last motivation was our main goal in the design of the Observer. Other objectives were also achieved however, as will be explained later on.

The Observer is different from a simulation tool in the sense that it is the effective code of the application which is running on the real distributed system. However, running an application under the Observer tool is not a "normal" execution, because the intrusive character of the Observer slows down the application.

2.1 Goals of the Observer

Our first objective for the Observer is to build a demonstration tool. Demonstrations have become an important matter in the software application area, in order to convince the clients that an application matches its specifications or that the ideas and concepts it shows are worthwhile. The obvious way to proceed is to run the application and to show its functionalities. But this is generally not enough for a well-informed audience who wants to know more about the internal working of the application: design, implementation, basic choices, etc. This is a more elaborate level in the demonstration and its difficulty depends on the kind of programming tools used:

- If the application was developed in a traditional sequential language, the description of the program is usually sufficient because the execution is completely linear (there is an unique flow of control).
- When using an object-oriented language, it can be interesting to show what the objects involved in the application are and particularly how they interact with each other; [9] and [10] have shown the benefits that can be expected of graphical tools.
- Parallelism can be another critical aspect to illustrate with adequate tools. For example, one may want to show how the parallel branches evolve during an execution, how they interact and synchronize (rendez-vous mechanism such as in Ada [6], asynchronous or

synchronous communication).

- If the application is distributed on several machines, the demonstration of its execution is still more complex. For example, one may want to see the location of the objects and their migrations in real time, even if the system is designed to hide these details from the user.

This discussion shows that the more complex the application is, the more essential it is to provide a high-level demonstration tool.

A second objective of the Observer is to show how the underlying system operates. This goal was particularly important in our context because we are developing an experimental distributed system. We will see also in sections 3.3 and 5.1 that such a tool can be easily improved in order to provide some graphical debugging facilities.

The specification and the design of an application observer closely depend on the basic concepts of the underlying system (process or object-oriented, active or passive objects, etc). So, before going on with the description of the Observer, we briefly present in the following section the main features of the Guide system. A detailed description may be found in [2], [11].

2.2 The Guide distributed system

The aim of the Guide project [2] is to explore distributed computing, structured in terms of objects and based on a set of heterogeneous workstations interconnected via a local area network. The system is targeted towards cooperative applications such as document processing and program development. Therefore, *object sharing* is an important feature of our model: objects are *persistent* and the principal way of communication between concurrent activities. The project aims at defining an architecture which hides most of the problems inherent to distribution for storage, execution and resource allocation, thus providing location, object and execution transparency.

The basic concepts of the Guide object model are [11]:

- *typed objects* for separating interface from implementation and allowing static control; a subtyping relation is also defined for interface specialization;
- *class mechanism* for object abstraction and instantiation; a subclassing relation

(inheritance) allows code specialization and code sharing.

The main abstraction provided by the computational model is called a *job*. A job is defined as a multi-threaded, transparently distributed virtual machine, which groups in a common address space a set of objects and a set of threads of control, also called *activities,* operating upon these objects. The execution of an activity within a job consists of a sequence of method invocations on objects; the creation of parallel activities is defined by a language construct. Guide objects are *passive*, i.e. there is no process attached to an object; so, object methods are executed by activities.

An object is mapped in the address space of a job at the first reference. The set of objects mapped in a job at a given time is called the *context* of the job. This context is implicitly shared between all the activities of the job; in addition, objects may be shared between the contexts of two jobs. An object can be loaded on any site, depending on the site of the caller, the current use of the object (e.g. whether it is shared by other activities) and the load-balancing policy.

A class is handled as a particular kind of object. It is actually a read-only object that can be automatically mapped in the address space of a job when one of its instances is used; furthermore, the class can be duplicated on several sites to allow a local execution for its instances. A class object is composed of a data template and a set of method templates (in fact the code of these methods) that will be used to instantiate objects of the class. Several methods are provided by a class object to perform this instantiation: *New*, *NewOnSite*, etc.

2.3 An Observer for Guide

Starting an application in the Guide system consists in creating a job in which an initial object is mapped, and invoking a first method of this object by an initial activity. The rest of the execution is defined by a sequence of method calls with the possible creation of new parallel and synchronous activities or the creation of asynchronous jobs. As some objects may require to be executed on a different site than the site of the caller, jobs can dynamically expand on several sites.

With this description of the execution of an application, we can identify the different entities involved in the observation of an application:

- Objects with their access methods: the execution flow can be shown by a graphical trace of the sequence of methods calls. Classes are also displayed as ordinary objects.
- Activities: animating icons associated with activities can give a visual image of concurrent execution.
- Jobs with their contexts: the evolution of job contexts can be dynamically illustrated by opening and closing windows associated with jobs. These windows display the objects mapped in job contexts, allowing to show object sharing.
- Sites: in order to show the distribution of the application and its dynamic evolution when jobs expand, a window on the observation display can be associated with each site. This window will contain the representation of the entities (jobs, activities, objects), currently located on the site.

3 The Observer at work

The best way to get to know the Observer is by seeing it at work. We have chosen four simple programs to illustrate the main Observer features. These programs do not necessarily represent applications targeted for the Guide system but allow us to introduce the Observer in a progressive fashion, as well as to illustrate some features of the Guide model.

3.1 Observing simple programs

Our first example is the well-known Hello program whose goal is to print the message Hello, world on the standard output. Figure 1 shows a snapshot of this execution under the Observer; we can see in this snapshot that:

- A job has been created to execute the program (*Job 1*).
- The job contains a single activity (*Act 1.1*).
- Six objects have been mapped in the job context: four classes (*Class, Top, Hello, Chan*) and two instances (*Hello 1, Chan 1*). Each object is represented by a box containing its name and the list of its methods.

- The activity is currently executing the method *WriteString* of the object *Chan 1*(*Chan* is the class which provides input-output methods).

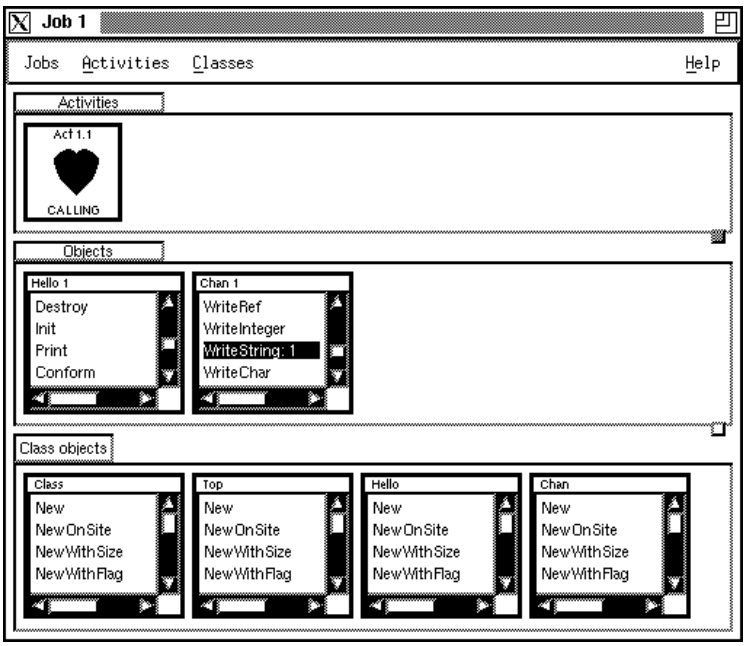

Figure 1. Observing the Hello Program.

We can see in the snapshot that this application is implemented by a single class (*Hello*) which uses the predefined classes *Class*, *Top* and *Chan*. An animation allows the user to follow the execution flow: each method call causes a heart beat in the activity doing the call and the highlighting of the method called. When a method is called by an activity, an identification of this activity (its number) is displayed beside the method name (e.g. *WriteString: 1*). Moreover, the user can ask to see the activity stack at any moment; figure 2 illustrates this feature at the time of our snapshot. We can see in particular that *Init* method is the entry point of the application.

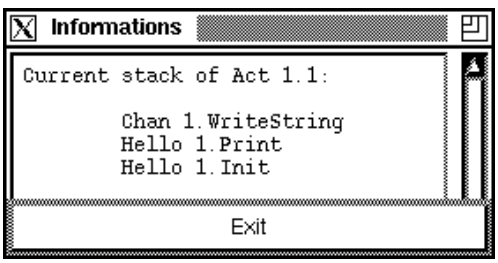

Figure 2. Activity stack for the Hello program.

Many features of the Observer are hidden by the simplicity of the Hello program. A second

application will be useful to show them, e.g. a classic of its kind: the Towers of Hanoi. This program is more difficult to understand because of the number of objects used and methods called. Two features could be interesting to observe Hanoi:

- Filtering out objects of a given class. No graphical trace will be displayed for filtered objects. All operations performed in these objects will be ignored by the Observer.
- Grouping objects of a given class. A window will be associated to each such class and reserved to the observation of its instances.

Disks and pickets are the most important objects in Hanoi. So, we can choose to see only these two classes and filter out all others. Moreover, we can group together objects of each of these classes to better visualize the execution. Figure 3 illustrates these choices.

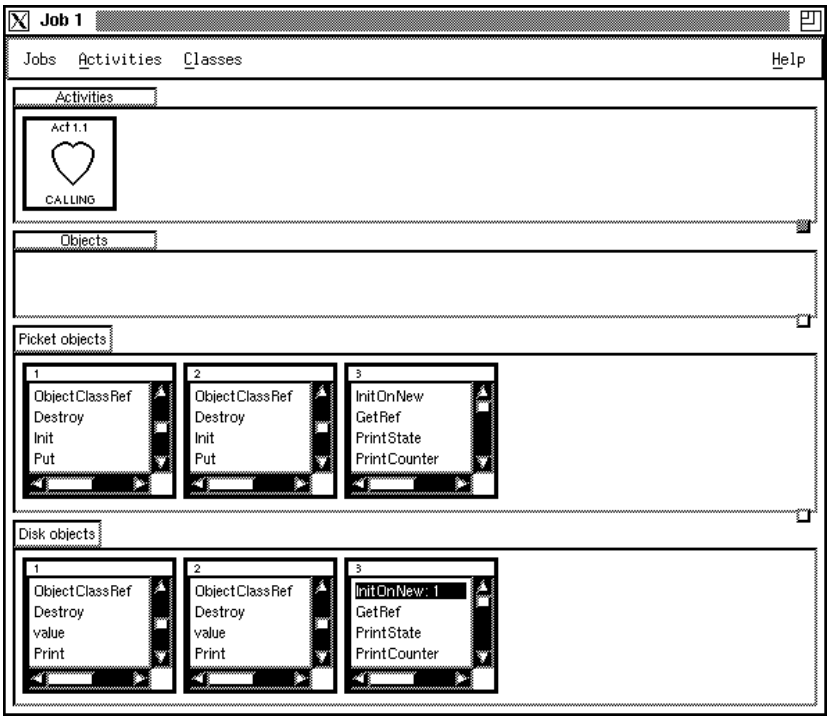

Figure 3. Another snapshot during the execution of Towers of Hanoi.

3.2 Observing more complex programs

Parallel programs are more difficult to observe because of the simultaneous execution flows composing these programs; communication and synchronization operations increase this difficulty.

The Observer has been carefully designed to deal with this kind of programs and to simplify this observation. Another classical program, Dining Philosophers, will illustrate this aspect.

The execution of Dining Philosophers is illustrated by figure 4. This snapshot shows six activities: the first one (*Act 1.1*) is the root activity for *Job 1,* which has created the other activities (*Act 1.2* to *Act 1.6*) corresponding to the five philosophers; the first activity is waiting at this time for termination of the others. We can also see three classes of objects (the others have been filtered out):

- A *Table* which is an abstraction of the table where the philosophers are dining. Five plates are set on the table with one fork between two adjacent plates.
- An *IoMutex* which represents an IO stream used in mutual exclusion.
- Five *Philosophers* seated around a dish of spaghetti.

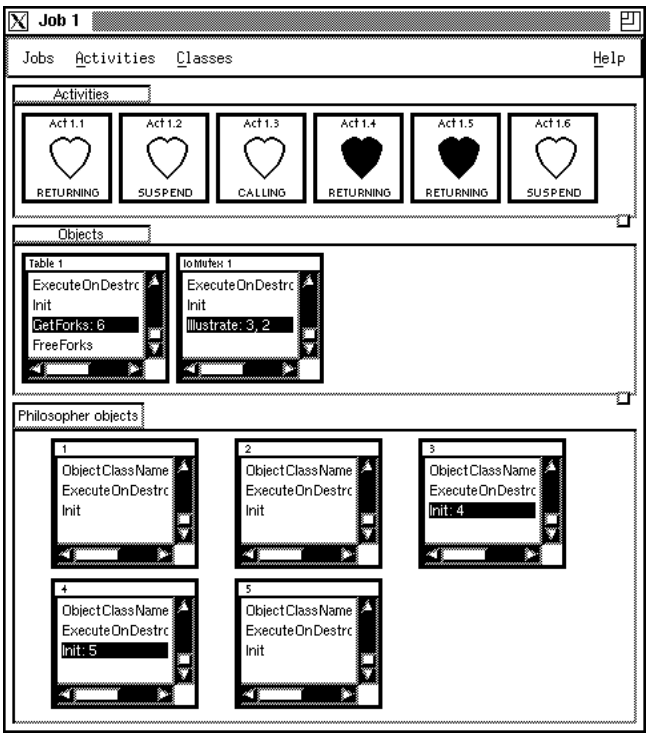

Figure 4. A snapshot during the execution of the Dining Philosophers.

Each philosopher is implemented by a different activity executing method *Init* of an instance of class *Philosopher*. The life of a philosopher consists of alternate periods of eating and thinking.

When a philosopher is hungry, he tries to get the couple of forks close to its plate (*GetForks* method in *Table* object): if the forks are not free, the philosopher is suspended until their release (e.g. *Act 1.6*). Each time a philosopher attempts to change its state, he calls the *Illustrate* method (*IoMutex* object) in order to print a message. Figure 4 shows two activities that have called this method: only one of them is currently executing it (*Act 1.3*); the other one has been suspended because of the mutual exclusion associated to the *IoMutex* object.

Each activity is animated in a similar fashion (heart beat and highlighting of methods called). On the other hand, each object is displayed only once, even if it is used by several activities, in order to represent object sharing (*Table* and *IoMutex* objects).

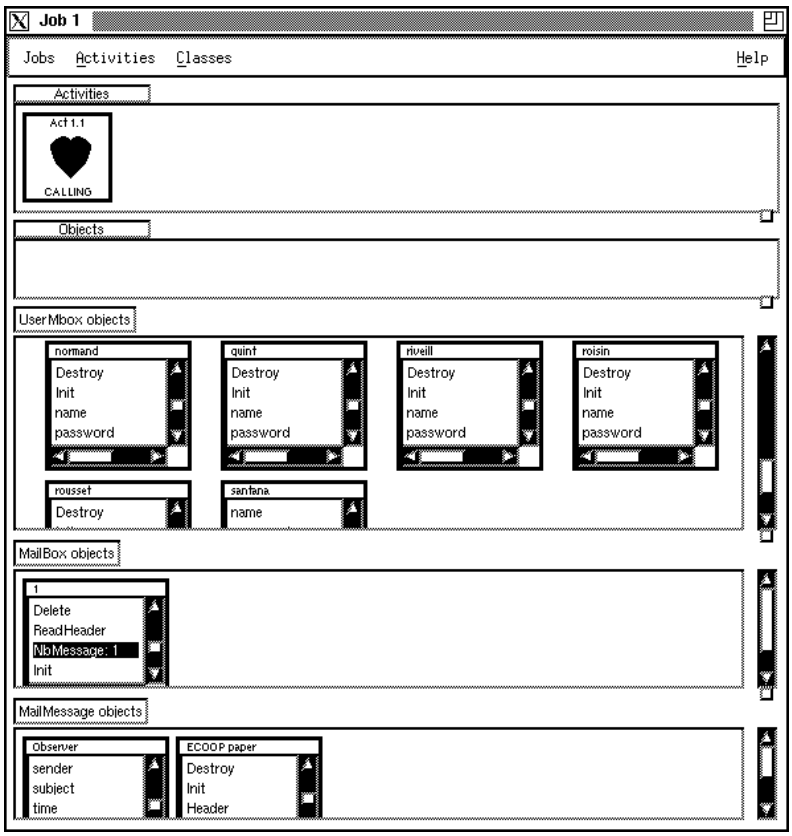

Figure 5. Transparency view for the Distributed Mailer.

Another area where the Observer can be helpful is distributed computing. Figures 5 and 6 show different views of the execution of a distributed mailer written in Guide, observed while user *santana* was reading his mail. The first view (job view) is similar to snapshots shown for previous

applications; distribution is not visible in this view: the execution is shown as if it was done in a virtual machine, where object location is completely transparent. On the other hand, views (site views) in figure 6 are directly connected to the sites visited by this application. Several remarks can be made about these views:

- We have chosen to filter out several classes that are irrelevant for our purpose.
- The user is consulting his mailbox (*UserMbox santana* and *MailBox 1*) which contains two messages (*MailMessage Observer* and *ECOOP paper*).
- The activity (*Act 1.1*) and its related job (*Job 1*) are represented on two sites (*bounty* and *guadalupe)*. They were created on the first of these sites but were diffused to the second one in order to use the *MailMessage Observer* object.

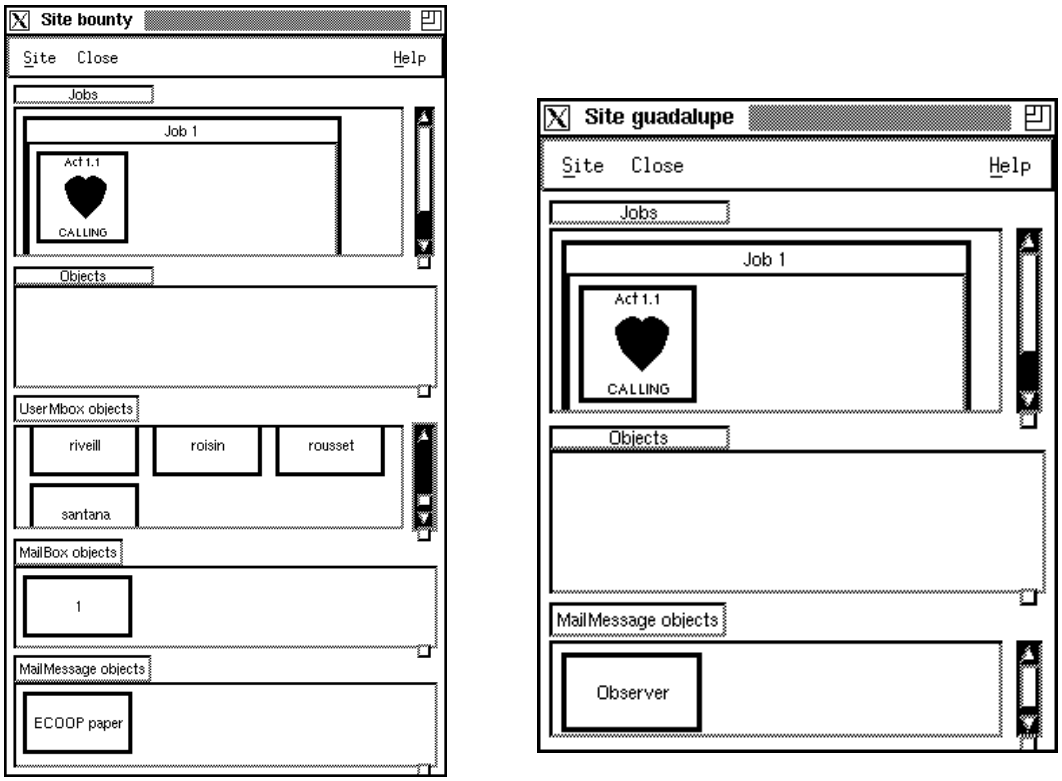

Figure 6. Distribution views for the Distributed Mailer.

- Most objects are mapped on the site where they have been called; it applies for all *UserMbox* objects, even if some users have their mailboxes located on another site. But some objects are constrained to be mapped in their location site (e.g. *MailMessage Observer*) or

are already mapped on another site when they are called.

These two kinds of views are complementary for the observation of distributed applications. However, both are not strictly necessary; actually, a user can choose to see only one of them, according to the type of execution wanted (to show or to hide distribution). The site views are specially interesting to observe remote method invocations as well as object migration between two sites.

3.3 Summary

Observer features

The previous examples have illustrated the main features of the Observer. We summarize them as follows:

- Visualizing application structure and execution flow.
- Filtering and grouping objects of a given class.
- Providing multiple views of the same application to show or to hide distribution.

The Observer also provides several mechanisms to control the execution speed:

- Stopping the execution to see the progress of the application and to analyze the current status of activities and objects. The execution will be killed or resumed at this point depending on user request.
- Tuning the execution speed to better follow the progress of the application.
- Step by step execution. All operations and events are considered as single steps (see 4.2), as long as they are not related to a filtered object.

These mechanisms allow the user to refine and better control the observation of its application. They could be very useful in particular to explain its behaviour and operation in real-time.

Doing other tasks

The Observer has been designed to facilitate the demonstration of applications; many features

reflect this choice: speed tuning, animation, configuration, object naming, etc. However, after some early tests, we realized that this tool would be helpful in many other areas: demonstrating or learning the Guide system itself, understanding a program, debugging a program, etc. We then decided to explore some of these possibilities, adding new features to the Observer in order to deal with them:

- Several mechanisms to give meaningful names to objects: enumerating instances of a class, value of a variable of the object, object identifier (system reference), etc.
- Printing informations about an object or an activity: location, mapping site, state, activity stack, object variables and methods, etc.
- Printing the contents of an object (value of its variables).
- Execution control: stopping/resuming the execution, starting an application on a given site, killing the application, step by step execution, breakpoints, etc.
- Miscellanous: animation control, object iconifying, etc.

Preliminary tests seem to show that the Observer gives very illustrative results in many cases. However, some problems still remain, specially in debugging, due essentially to the intrusive character of the tool (see 4.4).

4 Implementation issues

In this section, we describe our implementation of the Observer, with an emphasis on the main problems that were found and the solutions that were applied. The current implementation consists of about 12500 lines of C code and 2000 lines of uil (Motif language), and runs on top of the Guide system.

4.1 The Observer Architecture

The Observer is made up of four components, working in a close interaction: the tracer, the kernel, the interface and the mediator (Figure 7). The first component catches events signalled by the Guide system to indicate the progression of the execution, and forwards them to the kernel.

These events are generated only for applications executed in a special mode and correspond to elementary actions performed by the application (see 4.2).

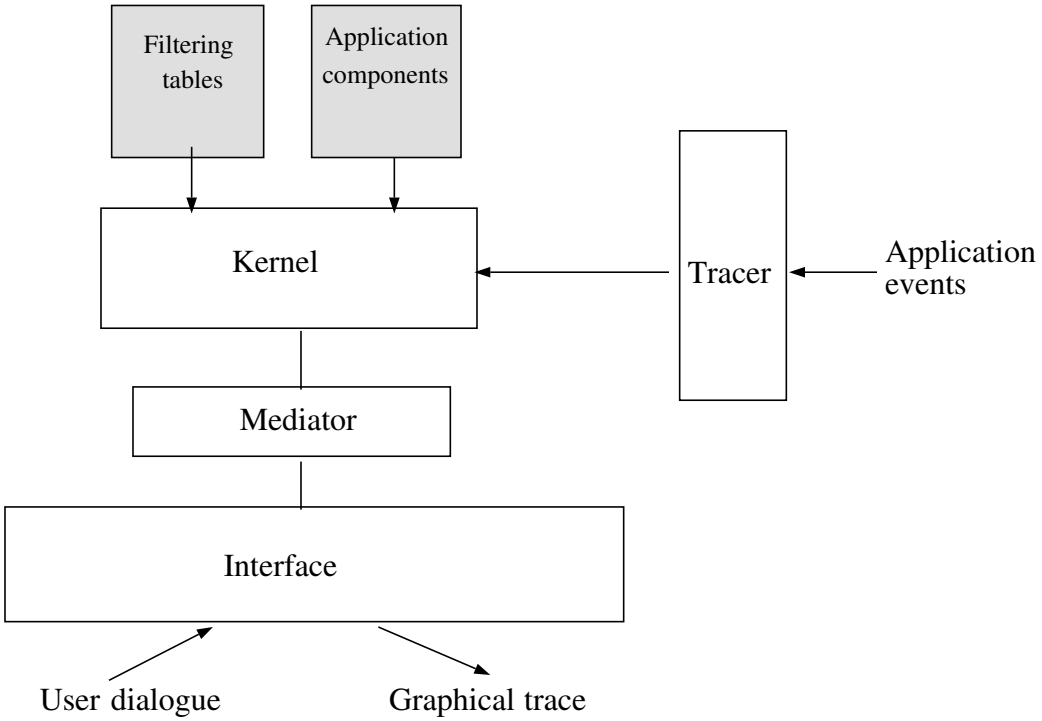

Figure 7. Global architecture of the Observer.

Event handling is done by the kernel, in several steps. First, it attempts to identify the components concerned by the event: job, activity, method and site. Then, it filters out relevant events, discarding those that are related to a filtered job, activity or object. And finally, it processes the event, updating its internal data structures and printing a graphical trace of the event.

The interface component plays a leading role in the Observer structure. It is in charge of displaying the graphical trace and managing the dialogue with the user. The communication with the user involves different kinds of graphical elements (widgets): menus, dialogue windows, icons, etc; figures 1 to 6 give a general idea of this interface.

Communication between the kernel and the interface components is done through the mediator. The kernel uses a simplified model of interface, which is translated by the mediator in terms of procedures and data structures used by the interface; in this way, the kernel does not depend on the interface implementation. A similar approach is used for the calls done by the interface to the

kernel, in response to operations requested by the user through the dialogue.

4.2 Communicating with the system layer

The observation of an application is achieved through a close cooperation between the Observer and the Guide system. Event signalling is the main feature of this interaction: an application execution is followed by the Observer through the elementary actions that it performs; these actions are called *events* and can be of three kinds:

- Object events: method call, method return, object creation or deletion.
- Virtual memory events: object/class mapping, object migration.
- Virtual processor events: job/activity start or termination, job/activity diffusion towards another site, activity suspension or resumption depending on a synchronization condition.

In order to insure homogeneity with the Guide model, we have chosen to use objects and standard system mechanisms to implement this interaction. Therefore, the Observer associates a communication object with any application it launches; this communication object is a bounded buffer managed according to the producer/consumer model. The messages which describe the events are inserted in the buffer by the Guide system, without any intervention of the application. The Observer withdraws the messages from the buffer and processes them sequentially. If the buffer is full, the application is stopped because of the synchronization conditions associated with this communication object.

Moreover, the Guide compiler creates a symbol table object for each class compiled and links it to the class. These objects are used by the Observer during the computation to find the data it needs: variables and methods of an object, class name, etc. They are also used to keep other data, which are specific to the Observer, such as links toward class source, class icons, default filters, etc.

Finally, the data associated with jobs, activities and objects such as system references, loading node, activity status, etc, are continuously collected during the time the application is in progress. These informations are used internally by the Observer but can be displayed at explicit user request.

4.3 Visualizing data and execution flow

The observation of an application is essentially an interactive task. In order to understand the execution of the program, the user tries to follow it through the pictures displayed by the Observer tool. Thus it is important to give to the user some means for interacting both with the display and with the execution of his application.

The display of the graphical trace should adapt itself to the user speed of reaction. An important aspect of our design was to decide what components to display and how often a picture should be updated. Thus, we decided not to display the evolution of the state of an object, because that would have slowed down the application too much. On the other hand, we chose to adjust activities animation speed to the current demonstration speed. Besides, we provide the user with the means of filtering out some superfluous components, so that he can limit the number of components and graphical traces displayed by the Observer.

A second aspect we consider as important is the control of the execution by the user. It seems essential that the user be able to adjust the speed of the execution as well as to stop and restart it . The control of the application is done by the Observer through the communication object: if no message is withdrawn from the buffer, the application will block as soon as the buffer is full. The execution speed is tuned by setting a delay introduced between the processing of events signalled by the application.

4.4 Performances

To weigh up the performance of an interactive tool is quite difficult. The components one should consider for this purpose can widely change according to the nature of the tool: editing, debugging, drawing, etc. For the Observer, the task is more complex due to the fact that its graphical trace should be visually observable by the user, allowing him to understand the execution. These factors seem contradictory with any performance goal for at least two reasons:

- The user is physically unable to follow the trace of a phenomenon which evolves too quickly.

- The graphical display significantly slows down the execution of the Observer. Moreover the trace generated by the system induces an important overhead on the application. Both facts prevent the application from running at a normal speed.

In addition, the execution may be disturbed, or even altered, by the intrusive nature of the Observer. In particular, a concurrent application may have the execution of its activities completely modified, which may lead to some modifications in their execution flow.

In order to reconcile these aspects, several solutions can be proposed:

- Using a record/replay technique (see section 5.1).
- Sampling, which means stopping the execution at regular intervals and guessing the events that happened, from the current state of the application.
- Tracing the execution in real time.

The overhead imposed on the application can vary approximately from 10% to a factor of 10, depending on the chosen solution; the above solutions are in increasing order of overhead. On the other side, the implementation cost can also significantly vary, and from that point of view, the third solution presented is the cheaper.

For the Observer, we do not think that the disturbance of the execution is a significant factor, at least for the goal that we want to achieve. Furthermore, a great part of the sequential and even concurrent applications are not sensitive to such a disturbance. So we have adopted the last solution: tracing in real time.

It then becomes difficult to speak about performance. The only element which could be taken into account is the response time of the tool in comparison with the user ability for following graphical trace. Such an element is very hard to quantify.

5 Future work

After first experiences with the Observer for several non trivial applications, we are investigating its use for other purposes than pure observation. We have identified three research directions: debugging, profiling and restructuration of an application.

5.1 Debugging applications with the Observer

Cyclic debugging is the most usual method for finding errors in sequential programs. The user executes his program, tracing statements and putting breakpoints with increasing levels of detail, to obtain more information about the causes of an error. This technique is effective only for deterministic programs, where each execution follows the same execution flow for a given input. Unfortunately, many distributed and parallel programs are not deterministic because of race conditions existing among their processes; furthermore, the intruding action of the Observer may strongly disturb the relative speed of processes that compose the application, making more difficult the reproduction of its execution. A solution to this well-known problem in parallel debugging consists in breaking the process in two phases, as proposed in [12] and [15]:

- The recording or monitoring phase: during normal program execution, the debugger saves the relative order of relevant events as they occur. Events to be recorded are restricted to a handful: process management, process communication, access to shared objects, etc. Data associated with these events is not saved, in order to reduce the time and space necessary to this recording; they will be recomputed during the replay phase.
- The replay or debugging phase: subsequent program executions will be done under the control of the debugger, so as to reproduce the execution behaviour of the first phase, following exactly the order of events recorded but without any timing constraints. In this way, cyclic method becomes possible and can be used to debug a program.

We are currently working on a new structure for the Observer, based on this approach; this structure will be used to do debugging as well as to observe applications. We are also working on new features to be added to the Observer to facilitate program debugging: displaying object values

in real-time, better visualization of these values, version management for objects modified by the application, reverse execution, etc.

5.2 Profiling applications with the Observer

Another functionality which can be achieved with the Observer is application profiling ([8], [18]). As the Observer traces each elementary action performed at application execution time, it is able to construct a graph of this execution and extract from this graph different kinds of statistics: calls per method and per object, object instantiations per class, object loading and migration, synchronization events, etc. These informations provide a useful support for application analysis aiming at enhancement or redesign.

However, we are aware of the current limitations of the Observer for timing measures ([18]). The overhead brought by our tool seriously disturbs these measures and makes them useless. Work currently done for debugging should allow us to improve this situation.

5.3 Using the Observer for restructuring applications

Profiling information as well as different execution traces provided by the Observer could be used for restructuring an application. At least two kinds of uses seem to be of interest:

- Objects preloading: the Guide system is based on the dynamic binding of objects. An application is defined externally only by its entry point (first method executed on the initial object); in order to reduce the cost of dynamic loading without changing the object model or the source code, the Observer could be used to identify the objects involved in an application and automatically generate a preloading phase for future executions.
- Application configuring for efficient use of the distribution, as in the Meta system [14]. Observing an application allows the identification of subsets of objects heavily related: objects of a same subset should be placed on the same node, so as to convert their mutual invocations from remote into local, thus reducing the global cost of the application.

6 Summary and conclusion

This paper has outlined some of the features of the Observer, a tool for the visualization of applications running on an object-oriented distributed system. We have shown its main features for visualizing, animating, filtering and grouping objects, providing multiple views of the same application and controlling the execution speed. We have also identified its current limitations in the perspective of a debugging tool, and we have mentioned how it should evolve to take into account debugging constraints.

We have learned several practical lessons from the construction and use of the Observer. They may be summarized as follows:

- The high degree of abstraction of an object model, combined with location and execution transparency, hide the inherent internal complexity of the execution of a distributed application based on object sharing. The Observer is an ideal tool to exhibit this complexity. We indeed learned a lot about the operation of our system by observing a few simple applications.
- The choice of the information to display and of its rate of change are decisive factors for a convenient operation, to prevent the user from being overwhelmed by the rate of production and the amount of information. Tuning mechanisms, such as object grouping and filtering, and tuning of execution speed, are essential to the operation of a really usable tool.
- Multiple views of the same application (e.g. showing or hiding distribution) provides convenient levels of abstraction for the observation.
- The object and execution model of the Guide system proved to be useful in the design and implementation of the Observer itself, both for organizing the information transmitted by the application to the Observer (association via symbol table objects) and for synchronizing the flow of control of this transmission.

The Observer has been used for several public demonstrations (e.g. the European ESPRIT technical week at Brussels, November 1990). These early experiences showed that the operation of

distributed applications can be clearly explained with such a graphical tool. A second result of the Observer, which is important in the context of a research project on distributed systems, is its ability to visualize system entities, allowing easy demonstrations of the functionalities of our system and its internal operations. This point is actually very interesting because the demonstration of a system is a particularly difficult task, specially for a system such as Guide, for which we have made the choice to manage at the system level all the problems involved by distribution (remote execution, storage, etc), so as to provide location and execution transparency at the user level.

Acknowledgments

We have been constantly supported by the Guide team while specifying and implementing the Observer. We would like to thank to Rémi Amouroux who took part in the implementation of the Observer. We wish also to thank Sacha Krakowiak and Xavier Rousset for their detailed criticisms of previous drafts of this paper. Project Guide is partly supported by the Commission of European Communities under the Comandos ESPRIT Project (no 2071).

Bibliography

[1] A. Ambler, M. Burnett. Influence of Visual Technology on the Evolution of Language Environments. *Computer*, 22 (10), pp. 9-22, October 1989.

[2] Balter et al. Architecture and Implementation of Guide, an Object-Oriented Distributed System. *Computing Systems*, to appear in 1991.

[3] H-D. Böcker, J. Herczeg. What Tracers Are Made Of. *OOPSL/ECOOP'90 Conference Proceedings*. pp. 89-99, October 1990.

[4] M. H. Brown, R. Sedgewick. A System for Algorithm Animation. *Computer Graphics*, 18(3), pp. 177-186, July 1984.

[5] Th. DeFanti, M. Brown. Visualization. Expanding Scientific and Engineering Research Opportunities. *Computer*, 22(8), pp. 12-25, August 89.

[6] M. B. Feldman, M. L. Moran. Validating a demonstration Tool for Graphics-assisted Debugging of Ada Concurrent Programs. *IEEE Trans. on Software Engineering*, 15(3), pp. 305-313, March 1989.

[7] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, Mass., 1984.

[8] S. L. Graham, P. B. Kessler, M. K. McKusick. Gprof, a Call Graph Execution Profiler. *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, SIGPLAN Notices*, 17(6), pp. 120-126, June 1982.

[9] V. Haarslev, R. Möller. A Framework for Visualizing Object-Oriented Systems. *OOPSLA/ECOOP'90 Conference Proceedings*. pp. 237-244, October 1990.

[10] M. F. Kleyn, P. C. Cingrich. GraphTrace - Understanding Object-oriented Systems Using Concurrently Animated Views. *OOPSLA'88 Conference Proceedings*. pp. 191-205, September 1988.

[11] Krakowiak et al. Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications. *Journal of Object Oriented Programming*, 3(3), pp. 11-22, September/October 1990.

[12] T. J. Leblanc, J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4), pp. 471-482, April 1987.

[13] T. Lehr et al. Visualizing Performance Debugging. *Computer*, 22(10), pp. 38-51, October 1989.

[14] K. Marzullo, R. Cooper, M. Wood, K. Birman. *Tools for Distributed Application Management*. (TR 90-1136), Department of Computer Science, Cornell University, Ithaca New York, July 1990.

[15] B. P. Miller, J. D. Choi. A Mechanism for Efficient Debugging of Parallel Programs. *Proceedings of the Workshop on Parallel and Distributed Debugging. SIGPLAN Notices*, 24(1), pp. 141-150, January 1989.

[16] T. G. Moher. PROVIDE: A Process Visualization and Debugging Environment. *IEEE Trans. on Software Engineering*, 14(6), pp. 849-857, June 1988.

[17] P. O'Brien, D. Halbert, M. F. Kilian. The Trellis Programming Environment. *OOPSLA'87 Conference Proceedings*, pp. 91-102, October 1987.

[18] C. Ponder, R.J. Fateman. Inaccuracies in Program Profilers. *Software - Practice and Experience*, 18(5), pp. 459-467, May 1988.

[19] D. Socha, M. L. Bailey, D. Notkin. Voyeur: Graphical Views of Parallel Programs. *Proceedings of the Workshop on Parallel and Distributed Debugging. SIGPLAN Notices*, 24(1), pp. 206-215, May 1988.

The Observer: A tool for observing Object-Oriented Distributed Applications

1 Introduction . . . 1

2 An observation tool in a distributed environment . . . 2

2.1 Goals of the Observer . . . 3

2.2 The Guide distributed system . . . 4

2.3 An Observer for Guide . . . 5

3 The Observer at work . . . 6

3.1 Observing simple programs . . . 6

3.2 Observing more complex programs . . . 8

3.3 Summary . . . 12

4 Implementation issues . . . 13

4.1 The Observer Architecture . . . 13

4.2 Communicating with the system layer . . . 15

4.3 Visualizing data and execution flow . . . 16

4.4 Performances . . . 16

5 Future work . . . 18

5.1 Debugging applications with the Observer . . . 18

5.2 Profiling applications with the Observer . . . 19

5.3 Using the Observer for restructuring applications . . . 19

6 Summary and conclusion . . . 20