Reversible Execution and Visualization of Programs with LEONARDO

Pierluigi Crescenzi[†], Camil Demetrescu[*], Irene Finocchi[§], and Rossella Petreschi[§]

Abstract

In this paper we present LEONARDO, an integrated environment for software visualization that allows the user to edit, compile, execute, and animate general-purpose C programs. LEONARDO relies on a logic-based approach to visualization: a mapping between concrete and abstract data structures is declared through a logic visualization language and animations are conceived as reflecting formal properties of algorithms. LEONARDO is able to automatically detect visual events during the execution of programs and simplifies the creation of visualizations according to an incremental approach. Moreover, it guarantees the complete reversibility of computations, bounded only by the potentiality of the working machine, and appears simple to be used. The latest version of LEONARDO is currently available over the Internet at the URL `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

1 Introduction

The ability of human beings at processing visual information and the rise of modern human-computer interfaces have led, in the last fifteen years, to a massive growth of software visualization (SV) as an aid for the design and analysis of algorithms and for the development and debugging of programs [34].

Briefly, SV is the static and/or dynamic use of text, images, and sound to point out structural and semantic properties of the source code and to make the data and control flow of sequential or parallel computations directly perceivable to human senses.

SV finds its natural applicability in helping and speeding up the comprehension of algorithms, in checking their correctness and highlighting hidden programming or conceptual errors, and, finally, in illustrating their behavior on specific data sets.

An ideal approach to understanding the behavior of an algorithm is to execute its single steps, leading the mental image of its formal properties to a visual effect and making visible what may be hidden to a first reading. Doing this through a sophisticated animation tool, rather than simply by pencil and paper, makes it possible to analyze problems' instances not limited in size and complexity which even long (and boring) handiwork would not be able to deal with. SV tools have already been used as an aid for teaching algorithms at different insight levels according to a *passive* or *active* usage: in the former case, students use the tool only to watch the execution of the program

[†]Dipartimento di Sistemi ed Informatica, Università degli Studi di Firenze, Via Cesare Lombroso 6/17, 50134 Firenze, Italy. E-mail: `piluc@dsi.unifi.it`

[*]Dipartimento di Informatica e Sistemistica, Università degli Studi di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy. E-mail: `demetres@dis.uniroma1.it`

[§]Dipartimento di Scienze dell'Informazione, Università degli Studi di Roma "La Sapienza", via Salaria 113, 00198 Roma, Italy. E-mail: `{finocchi,petreschi}@dsi.uniroma1.it`

on a fixed input, while in the latter case students may interact with the tool, either testing the program on their own inputs or creating new animations from scratch. The most recent results regarding the usefulness of such tools for teaching algorithms may be found in [19, 33, 35].

A good debugging system should allow the user to discover both errors due to a wrong implementation of an algorithm and, at a higher level of abstraction, errors due to an incorrect design of the algorithm itself. It is common opinion today that not enough effort has been devoted to the development of efficient tools for debugging programs [20]: as a consequence, the application of SV techniques to this field is actually becoming an interesting and rapidly growing research area [37].

At this aim, it is very important for a SV tool to be able to animate not just "toy programs", but significantly complex ones, and to test their behavior on large data sets. Unfortunately, even those systems well suitable for large information spaces often lack in advanced navigation techniques such as changes of resolution and scale, selectivity, and elision of information. Hence, great research effort has been recently devoted to this topic [29, 36].

In this paper we present LEONARDO, an integrated environment concerning both the design and the implementation of programs and their execution and visualization. LEONARDO was originally oriented towards the visualization of graph algorithms, but its graphical vocabulary has been later extended with many other kinds of visual objects in order to deal with general purpose algorithms. Since its execution environment is based on a virtual CPU, its debugging facilities can easily highlight subtle programming errors like memory access faults. Moreover, no size and complexity bounds are imposed on programs and data LEONARDO can handle, even if the current version has no advanced data navigation facilities like zoom controls and course/fine grained views.

LEONARDO's main features are the following:

- Animations are conceived as reflecting formal properties of algorithms: this is achieved by means of a logic visualization language, called ALPHA, that allows users to declare a mapping between concrete and abstract data structures.

- At any time during the execution of a program the successive computation step may lead either to the next statement or to the previous one. This implies a fully reversible execution of programs, bounded only by the size of the secondary memory available on the working machine. The ALPHA-based declarative approach and the complete reversibility of the virtual CPU make it possible to reverse the direction of the animation with no charge, i.e. without the necessity of saving extra information.

- The system has been designed trying not to discourage anyone interested in visualizing his/her own programs. Indeed:

  - Visual events are automatically generated by LEONARDO every time the program being visualized significantly changes the content of its data structures: therefore, the user is not required to have a deep knowledge of the source code and is relieved of the task of localizing and specifying interesting events.
  - An incremental approach can be used to specify animations. A minimal amount of information is required to produce a basic significant image: all missing pieces of information are given by default. Then, it is easy to refine the animation by adding, deleting, and modifying the visualization rules.
  - The program source code is highly reusable: programs animated in LEONARDO can be compiled in other environments that ignore the visualization rules as they are inserted as special comments.

– LEONARDO includes a graph editor that allows the user to test his/her programs on user-friendly defined graphs.

The rest of this paper is organized as follows. Section 2 presents a selection of existing SV tools. In Section 3 an overview of LEONARDO's structure is depicted and a detailed description of its features is given from Section 4 to Section 6. Section 4 concentrates on the declarative approach to visualization, while Section 5 and Section 6 explain how the system works. In more detail, Section 5 describes the execution environment, pointing out how the reversible execution is achieved, and Section 6 concerns the visualization system. Finally, Section 7 contains a complete animation example and Section 8 addresses conclusions and open problems.

2 Some Existing Tools

In the following we briefly describe some existing *live* SV tools, where a tool is said to be live when run-time and visualization-time are considered synonymous, i.e., the user may interact with the visualization as the program is running. In our summary we do not consider algorithm animation systems over the WEB although, in the last few years, the interest in this topic has grown a lot. We refer the interested reader to the corresponding bibliographical items [1, 2, 7, 14, 16].

Zeus [4], the latest evolution of **Balsa** [3, 8, 9], is constructed according to the imperative style (see Section 4) and is based on the idea of "interesting events" that must be annotated by the visualizer in the code. This allows the user to customize the visualization according to his/her effective necessities. At the same time, however, the visualizer must know very well the code that is "invaded" by calls to the graphical primitives made available by the tool. Zeus extensively uses color and sound [5] and deals with three-dimensional objects [6].

UWPI [17] automatically provides visualizations for the high-level "abstract" data structures designed by the programmer. Its heart is an "inferencer" which analyzes the data structures in the source code and suggests a number of plausible abstractions for each of them. UWPI shows graphical representations of data structures with no interaction with the visualizer. Unfortunately, the visualization may not represent exactly what the process is doing, due to the lack of a deep knowledge of the logic of the program.

Pavane [27] is designed to provide declarative three-dimensional visualizations of concurrent programs. In Pavane the visualizer declares a transformation between a fixed set of program variables and the final image by using rules according to a "declarative" style [25]. In contrast with the "imperative" style, this approach requires minimal knowledge of the program and no code modification. This issue can be important in a concurrent framework, since the execution may be non-deterministic due to the fact that invasive visualization code may change the outcome of a computation.

Polka [32] derives from the **Tango** system [31, 30] and is well suitable for animating both serial and parallel computations. It includes a graphical front-end, called **Samba**, that is driven by a script produced as a trace of the execution. Polka relies on an object-oriented imperative approach and supersedes the simple erase-redraw technique allowing the user to realize smooth animations.

ZStep95 [21] is a reversible and animated source code stepper for LISP programs that provides a powerful mechanism for error localization. It maintains a complete history of the execution and is equipped with a fully reversible control structure: the user lets its program running until an error is found and then can go back to discover the exact point in which something was wrong. Moreover, a simple and strict connection between the execution and its graphical output is obtained by elementary clicking actions.

	When	Where	Environment	Language	Visualization Language	Specialty	Visualization Time	Specification Method	Visualization Content	Graphical Vocabulary	Direction Control
Pavane	1992	Washington University	UNIX	C/Swarm	Rules	Concurrent programs, graph algorithms	Live	Declarative	Algorithm	Points, lines, enclosed shapes, position, color ...	No
Polka	1993	Georgia Institute of Technology	Windows '95 / UNIX	C++	C++	General purpose algorithms	Live and post-mortem (using Samba)	Imperative	Algorithm	Text, points, lines, splines, bitmaps, enclosed shapes, position, color ...	No
UWPI	1990	University of Washington	UNIX	Pascal subset	/	Graph and array algorithms	Live	Automatic	Program+ Algorithm	Text, lines, enclosed shapes, position, size ...	No
Zeus	1991	Brown University	UNIX	Modula III	Modula III	Data structures	Live	Imperative	Program+ Algorithm	Text, points, lines, enclosed shapes, position, color, size ...	No
ZStep95	1995	MIT Media Laboratory	MacOS	LISP	/	Debug of LISP expressions	Live	Automatic	Program	Trees and high level debugging diagrams	Yes
Eliot	1996	Helsinki University	UNIX	C	/	Data structures	Live	Variable browser	Program	Text, points, lines, enclosed shapes, position, color, size ...	No
Leonardo	1997	University of Rome "La Sapienza"	MacOS	C	Alpha	Graphs and general purpose algorithms	Live	Declarative	Program+ Algorithm	Text, points, lines, enclosed shapes, graphs/trees, color, size ...	Yes

Figure 1: LEONARDO and other existing SV tools.

In **Eliot** [18], finally, the animation is controlled by the operations on a specific set of built-in data types. The user needs to write no additional code: in other words, animation is embedded in the implementation of data type operations. The graphical presentation is then based on a "theater metaphor" where the script is the algorithm, the stages are the views, the actors are program's data structures depicted as graphical objects, and the director is the user.

Almost all these systems support multiple views of different programs, guarantee a high level of interactivity, and allow the user to control the speed of execution.

According to the three taxonomies of [22, 23, 26], in Figure 1 we summarize the main features of LEONARDO and of the other tools presented in this section. Looking at the table, it is worth pointing out that:

- The "language" programs to be animated are written in is not always the same as the "visualization specification language", if any.
- In general, tools can animate every program written in the appropriate language, but most of them are more suitable for specific problem domains.
- Tools that are not completely automatic can be partitioned into two groups, according to their imperative or declarative style. In Section 4 we present the main features of these two approaches: some of them have been already pointed out during the description of Zeus and Pavane, respectively.
- A visualization system can handle either the code and data flow of a program or the steps of the algorithm and the evolution of the related data structures. Dynamic properties of the

execution of a program are well captured by the notion of state transformation, where the state is characterized by the code, the program counter, and the memory image. Program visualization systems usually present information about the state in a graphical readable form: for example, they highlight the current instruction or display the content of variables in a structured form. On the other hand, the visualization of an algorithm should be able to display something not explicitly coded in the state information, or to interpret the meaning of a state transformation.

- Live tools can seldom run programs in backward mode since a lot of extra information about execution states needs to be saved. The task is usually even more complicated when the system produces animations.

3 An Overview of LEONARDO

Figure 2 shows the overall structure of LEONARDO. The figure is divided into a compile-time and a run-time part, according to the fact that LEONARDO is an integrated environment. In the remainder of this section we first describe these two parts and then we present LEONARDO's underlying logic. We do not consider aspects related to the implementation of LEONARDO, and we limit ourselves to describe its main components and to present its philosophy from a high-level point of view.

3.1 The Compile-Time Part

In the compile-time part of Figure 2 it is worth observing the presence of two compilers: one for the ANSI C language, and one for the declarative language ALPHA used for specifying the visualization (see Section 4.1). An interesting feature of ALPHA is the possibility to be embedded into C programs, making it feasible to interpret the state of the computation through declarative instructions.

LEONARDO features several standard facilities common to professional development environments, such as a text editor with syntax coloring, an on-line language reference, multiple errors and warnings management. A snapshot of the programming environment is shown in Figure 3.

3.2 The Run-Time Part

In the run-time part of Figure 2 we distinguish two kernels: the *virtual execution environment* and the *visualization system* (see Section 5 and Section 6, respectively).

The virtual execution environment features some typical operating systems facilities. Actually, LEONARDO is a multiprogrammed time-sharing environment: it is possible to have many processes (denoted as P_1, ..., P_n in Figure 4) running on a virtual reversible CPU. These processes are allowed to do system calls to LEONARDO's operating system that, in turn, acts as a bridge to the underlying real one.

As shown in Figure 2, LEONARDO's user can interact with the virtual execution environment and the visualization system through five different modules:

- The *control bar* allows the user to change the state of a process, to switch the execution direction, and to vary the animation speed. It also shows some execution statistics, like the number of low-level executed instructions.

- The *console window* acts as a standard terminal window.

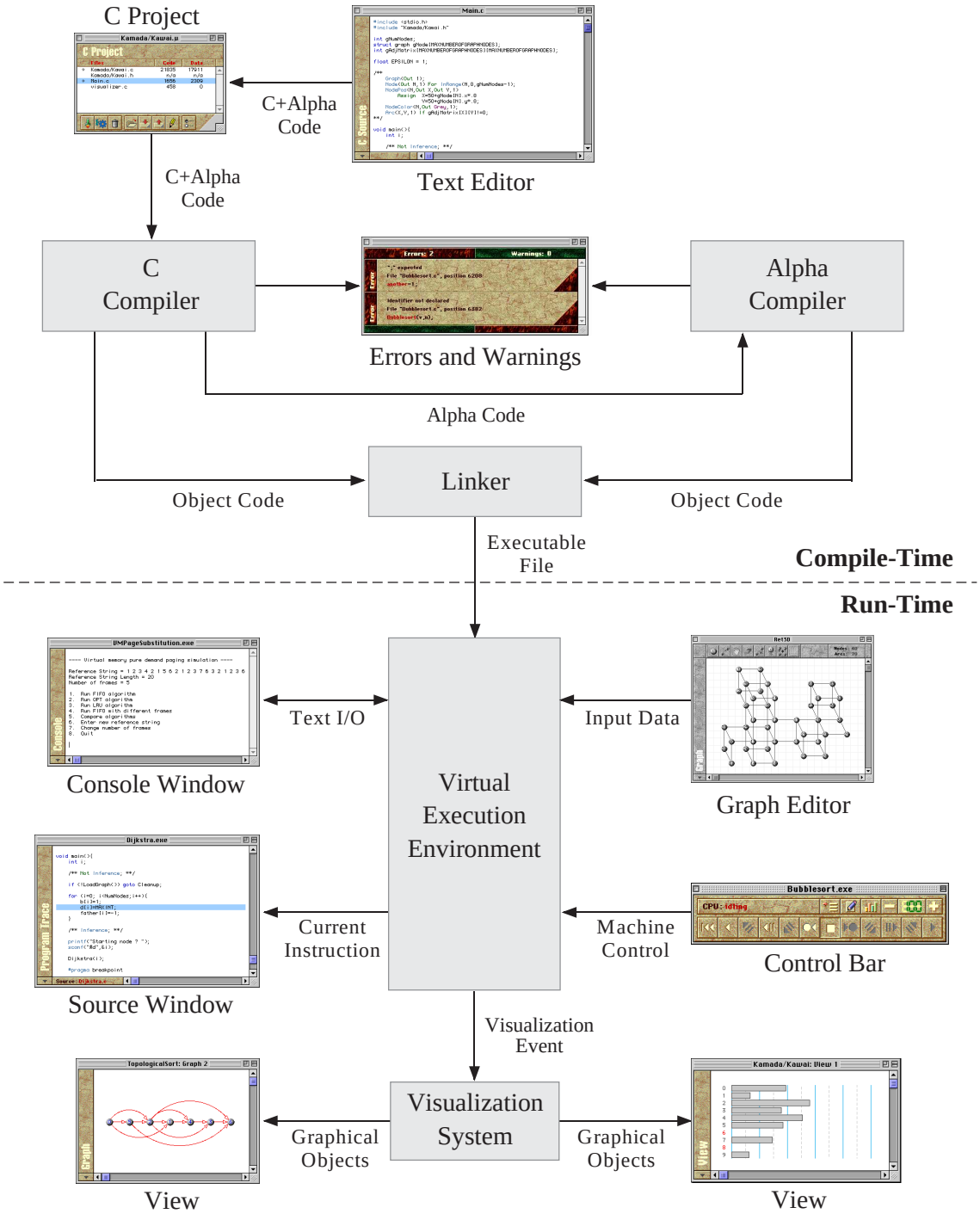

Figure 2: High-level architecture of LEONARDO.

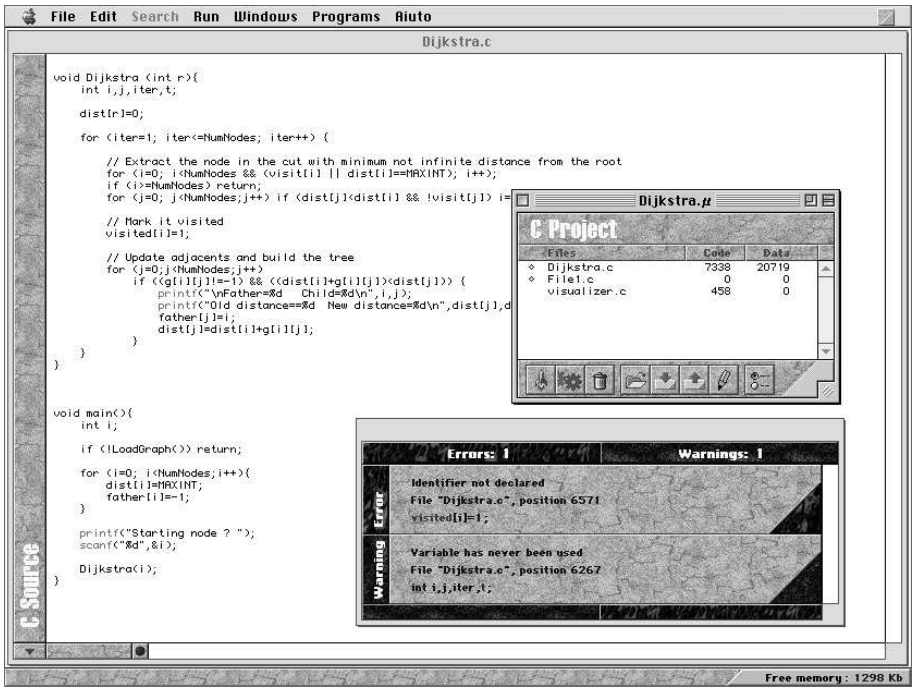

Figure 3: LEONARDO's editing and compiling environment.

- The *source trace window* highlights the current program instruction.
- *Multiple visualization windows* are supported. They are treated with the offscreen drawing technique in order to guarantee non-flickering images transformations.
- The *graph editor* (see Figure 5) provides a user-friendly approach for editing graphs: it is possible to create new graphs and to manipulate existing ones by simple click and drag actions.

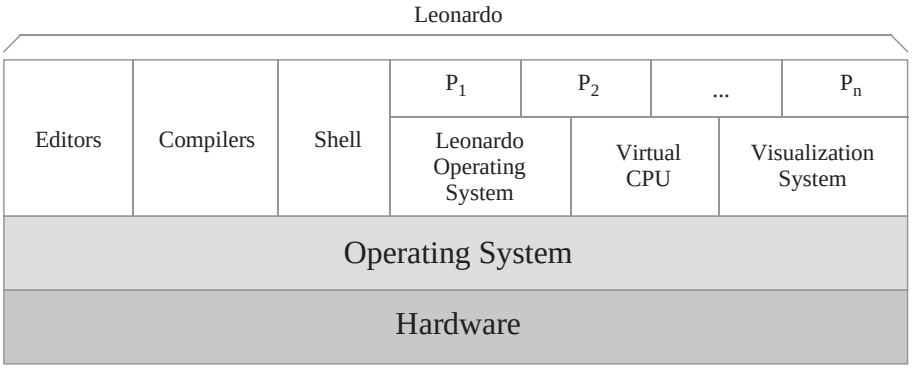

Figure 4: LEONARDO and the environment.

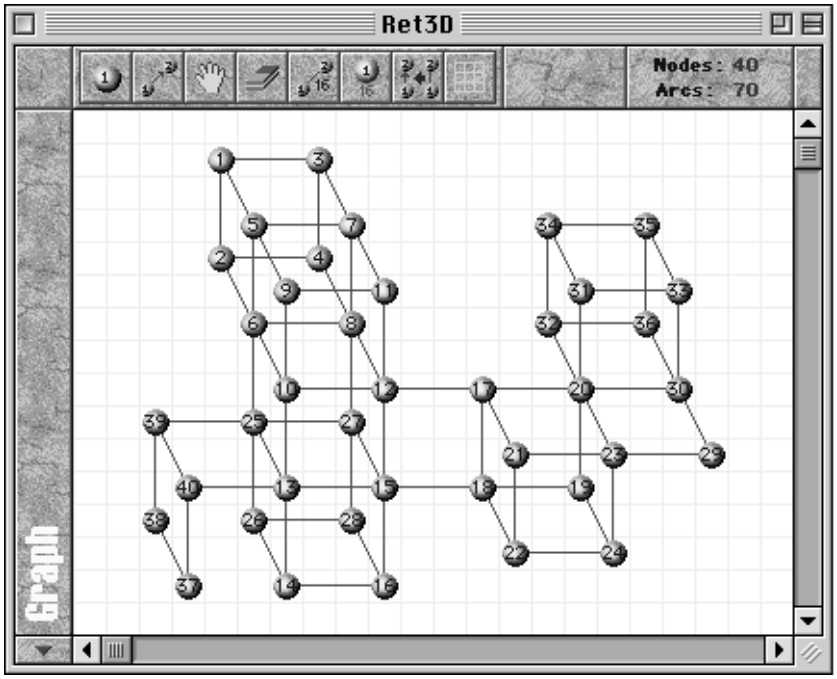

Figure 5: LEONARDO's graph editor.

3.3 LEONARDO's Approach to Visualization

The basic idea behind LEONARDO's approach to visualization is to enrich a generic C program with ALPHA declarations that specify a graphical interpretation for its variables. In this way, the user can get back his/her mental image of program's data structures, making up for the loss of abstraction due to the implementation [13].

Significant changes to the content of data structures during the execution of the program are automatically detected by the run-time environment that raises a visualization event. Graphical objects are then generated by the visualization system according to ALPHA declarations.

4 Declarative Visualization

According to the definition given in [26], we "consider program visualization as a mapping from some aspect of a program (or execution of a program) to a final image". All visualization systems realize such a transformation from programs or processes to images. The way the mapping specification is realized, either explicitly or not, tells declarative systems from imperative ones.

In systems based on interesting events, processes directly modify the image space by means of procedural actions, so that they are not only responsible of themselves, but also of driving the animation. Hence, we may represent the *imperative systems* as shown in Figure 6, where the process turns out to be the core of the system and visual objects have not their own life without its creative action. Clearly the process, being the visualization agent, is required to deviate its control flow to send editing commands to the visual world. This implies both that it has to know everything about visual objects and that these objects do not reflect at any time the process internal state, but are simply a consequence of the last visual editing action.

On the other hand, the core of a *declarative system* is the "declaration world", that may even

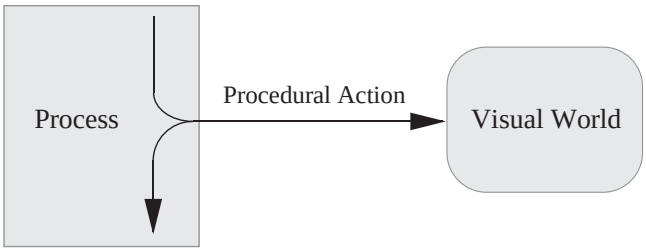

Figure 6: Conceptual model of imperative SV systems.

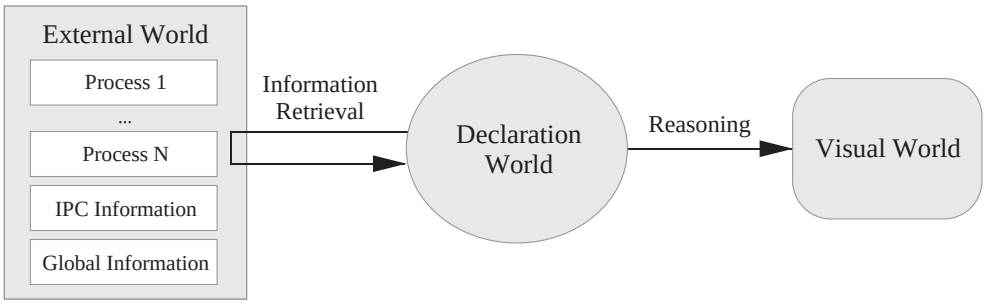

Figure 7: Conceptual model of LEONARDO as a declarative SV system.

exist as a stand-alone universe (see Figure 7): the visual world is described by means of declarations concerning the existence of visual objects, their interrelations, and their relations with the external world.

In particular, the declarative approach of LEONARDO makes it possible to obtain both a satisfying level of automation and an excellent capacity of synthesis. LEONARDO's visualization system has to maintain the visual world consistent with its declarative description: at this aim, it extracts information from the external world and synthesizes this information into graphical objects by exploiting the logical dependencies through reasoning processes.

In contrast with imperative systems, where the external world consists mainly of the process driving the animation, in LEONARDO the external world components may be extremely various and extendible. The main component is, typically, the process memory image, but other components may be added: the status of the controls of a graphic user interface (such as checkbuttons) and the data structures of LEONARDO's operating system's kernel (such as the scheduling queue) are just two possibilities. As a consequence of this variety of components which the visual world may depend on, it should be easy to extract global information about concurrent processes and then to animate non-sequential computations.

The declarative approach is well suitable for making up for the loss of abstraction related to the algorithm implementation [13]. As already stated, LEONARDO's environment makes it natural to get back the abstract interpretation of concrete data structures. Moreover, the visualization system accepts every kind of data structure freely implemented by the programmer, without the limitation of choosing only among the built-in structures provided by libraries as in most imperative systems.

A natural realization of the declarative approach, that makes it possible both to reason about objects and to synthesize information from the external world, is based on the use of a logic programming language in order to declare facts and rules regarding graphical objects. As a matter of fact, in LEONARDO declarations are expressed in a simple logic programming language called ALPHA.

4.1 The ALPHA Language

An ALPHA program is a sequence of assertions, called *predicates*. They are embedded into a host C program as special comments, they cannot modify the content of its variables nor call C functions: these choices guarantee that they are "semantically" invisible to the host language. In this way, ALPHA affects neither the expressivity nor the computational power of the host language. On the other hand, ALPHA predicates can directly read the content of all C program variables, according to their scoping rules, making it natural to specify the mapping between computation state and visual objects.

ALPHA predicates are dynamically inserted into the declaration world as they are encountered by the process control flow. Moreover, ALPHA features the possibility of grouping sets of predicates in order to easily insert/delete them into/from the declaration world. Indeed, since ALPHA predicates are supposed to describe the visual world, we may want this description to change according to the currently running code segment.

We do not formally define the ALPHA language, but we limit ourselves to give two simple examples of ALPHA programs (a more sophisticated example related to visualization is presented in Section 7). The interested reader can find a formal definition of ALPHA in [11, 12].

4.1.1 Computing the Factorial

The following ALPHA program computes the factorial of an integer number:

```
fact(0, Out 1) Moreover
fact(N, Out F) If N>0
               For Z: fact(N-1,Z)
               Assign F=N*Z;
```

As already stated, an ALPHA program consists of a sequence of predicates. Syntactically, a predicate is defined by means of a sequence of *head-body* pairs. The head specifies the name of the predicate and its input and output formal parameters, while the body specifies the computations which either define abstract objects or verify relationships between them. The keyword `Moreover` distinguishes each head-body pair by the successive one.

In the above example we have two pairs: the first one has just the head (that is, `fact(0,Out 1)`), while the second one has both the head (that is, `fact(N,Out F)`) and the body (that is, `If N>0 For Z: fact(N-1,Z) Assign F=N*Z;`). The first pair defines the base computation step, declaring the factorial of `0` to be equal to `1`. The second pair introduces an auxiliary variable `Z`, that receives the value of the factorial of `N-1`. The factorial of `N` is then assigned to `F` and is correctly declared as `N*Z`. Notice that, in the second head of the example, we have `N` as input parameter and `F` as output parameter: the keyword `Out` prefixed to `F` means that this parameter is introduced by name.

4.1.2 ALPHA Standard Predicates

LEONARDO's visualization system is able to recognize a set of ALPHA *standard* predicates and to compute them in order to retrieve information about what has to be visualized. The heads of these predicates are predefined and the user interested in realizing a new animation has only to fill-in their bodies according to the program requirements.

Standard predicates can be classified as: (a) animation control predicates (e.g., `Inference`), (b) enumerative predicates defining objects and their structural properties (e.g., `Graph`), and (c) predicates for attributes and retinal properties (e.g., `NodeColor`).

It is worth noticing that standard predicates are similar to the `main` function of the C language, that is, a special function recognized and automatically called by the environment. Due to their multiplicity, they are hierarchically organized according to their logical dependencies: for instance, it does not make sense for the user to define `NodeColor` without having defined `Node`. The system, in turn, computes first `Node` and then `NodeColor`.

In general, the order followed by the system to compute standard predicates is defined by a built-in *query algorithm* [12], that is used to build up a hierarchical data structure representing graphs, trees, and geometric models to be visualized.

4.1.3 Animating the Bubblesort Algorithm

Figure 8 shows the complete code to animate in LEONARDO the classical bubblesort algorithm. The C sorting program has been annotated with ALPHA predicates able to generate a sequence of rectangles having heights proportional to the integer values in the array `v`. In particular, five predicates are used, all standard but one (that is, `Max`).

- The enumerative predicate `View(Out ID)` implies the opening of a window labeled with the number `ID`.
- `Rectangle` is an enumerative predicate, too. Its first five formal parameters represent the identification number, the `X` and `Y` coordinates of the left-top corner, and the length and height of a rectangle, respectively; the last one is the identification number of the window that must contain the defined rectangles.
- `RectangleColor` is an optional predicate that specifies a retinal property of rectangles: if missing, a default color is provided by the visualization system.
- `VisualUpdate` is an animation control predicate whose truth value enables/disables the update of the screen.
- Finally, `Max(Out M)` is a non-standard predicate introduced to simplify the definition of `Rectangle`. It computes the maximum value in the array `v` and returns it in `M`; observe that it has been implemented by inserting a C block into an ALPHA predicate body.

Note that ALPHA predicates never modify the underlying program's structure in any way. Adding `/** Not VisualUpdate; **/` and `/** VisualUpdate; **/` is a successive refinement that the user may introduce in order to elide redundant visualization frames.

The sequence of images related to the execution of the animated bubblesort on the input array `[20,30,10,40]` is given in Figure 9.

5 The Virtual Execution Environment

In Figure 4 we have shown how LEONARDO is positioned with respect to the underlying environment. We are now ready to give a detailed description of both the virtual execution environment and the visualization system, showing all the components and the connections hidden so far. In this section we describe the *virtual execution environment* and, in particular, we point out our attention on the reversibility of LEONARDO's *virtual CPU*. In the next section we present the *visualization system*.

```
#include <stdio.h>

#define MAX_NUM         100

int n;
unsigned int v[MAX_NUM];

/**
    #define Dx          20
    #define Dy          4

    View(Out 0);

    Max(Out M)
        Assign M In {
            int i;
            M=v[0];
            for (i=1;i<n;i++) if (M<v[i]) M=v[i];
        };

    Rectangle(Out 1, Out X, Out Y, Out L, Out H, 0)
        For M: Max(M)
        For N: InRange(N,0,n-1)
        Assign  X=Dx*N
                Y=(M-v[N])*Dy/2
                L=Dx-8
                H=v[N]*Dy;

    RectangleColor(1, Out LightGrey, 0);
**/

void main(){
    int i,another;

    printf("Enter array size: ");
    scanf("%d",&n);
    if (n>MAX_NUM) return;

    printf("Enter values: ");
    for (i=0;i<n;i++) scanf("%d",&v[i]);

    do {
        another=0;
        for (i=1;i<n;i++)
            if ((v[i-1]>v[i])) {

                unsigned int temp;   /** Not VisualUpdate; **/
                temp=v[i-1];
                v[i-1]=v[i];
                v[i]=temp;           /** VisualUpdate; **/

                another=1;
            }
    } while (another);
}
```

Declare a view with label 0.

Max computes the maximum value of array 'v' that will be used by predicate Rectangle.

Declare by enumeration 'n' rectangles labeled 1 and belonging to view 0. The rectangles are centered along the y axis according to the maximum value in array v. Their heights are proportional to the values in v.

Declare the color of rectangles that have label 1 in view 0.

Swap items to eliminate inversions: the screen will not be updated during the execution of this operation.

Figure 8: The code for animating the bubblesort algorithm.

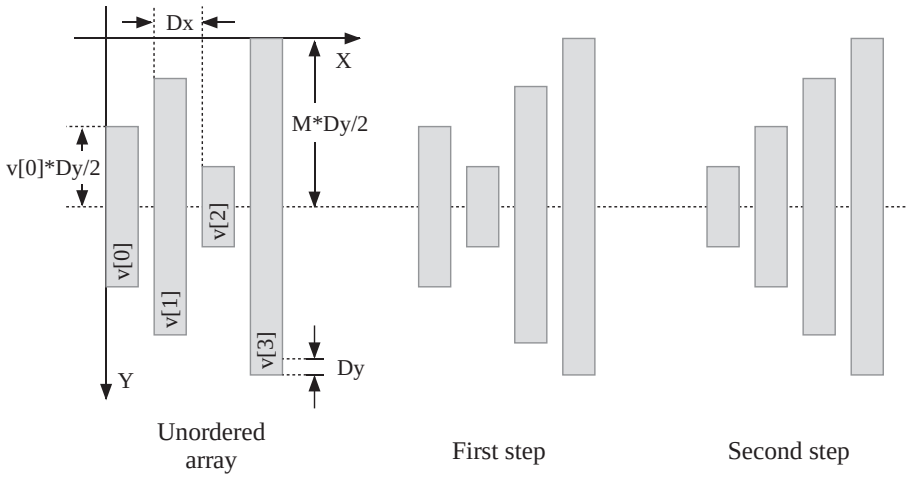

Figure 9: Bubblesort visualization on input [20,30,10,40].

As in any traditional operating system, LEONARDO has many specific *managers* devoted to give processes all resources they need (see Figure 10). These modules manage the virtual CPU scheduling and any memory allocation, I/O, and file access request.

It is interesting to observe that, due to the necessity of executing programs in backward mode, all LEONARDO's system calls are fully reversible. For example, sending a stream of characters to the console window through the I/O manager is reversed by removing them back from the stream and therefore by erasing part of the screen.

5.1 The Reversible Execution of Programs

The virtual CPU has been suitably designed to support the reversibility. It is a stack-based machine with only three registers: a program counter, a stack pointer, and an accumulator register. Its instruction set may be divided into four groups: (a) control flow instructions, (b) data flow instructions, (c) logic-arithmetic instructions, and (d) cast instructions (a complete list can be found in [15]).

The virtual machine can be used both to execute programs in a bidirectional way and to unidirectionally compute ALPHA predicates. Therefore, the virtual CPU has been designed to run in two different modes: (a) *user mode* to execute programs, and (b) *supervisor mode* to evaluate ALPHA predicates. The set of low-level instructions executable in user mode is a proper subset of those executable in supervisor mode.

The format of LEONARDO's virtual CPU instructions is:

`opcode` (1 byte) - `operands` (variable number of bytes) - `opcode` (1 byte)

The repetition of the `opcode`, due to the fact that instructions have different lengths, has been chosen for optimization reasons. Actually, there are two distinct control flows: the program counter is always positioned either on the first or on the second `opcode` according to the direction of the execution (see Figure 11). This choice greatly improves the execution speed.

Each sequence of forward execution steps followed by the same number of backward steps must be invariant with respect to the computation state. Since we have chosen to implement reversibility at virtual machine level, the correctness of the inversion of a high-level instruction (compiled as a sequence of low-level ones) and, therefore, of the whole program, follows by induction.

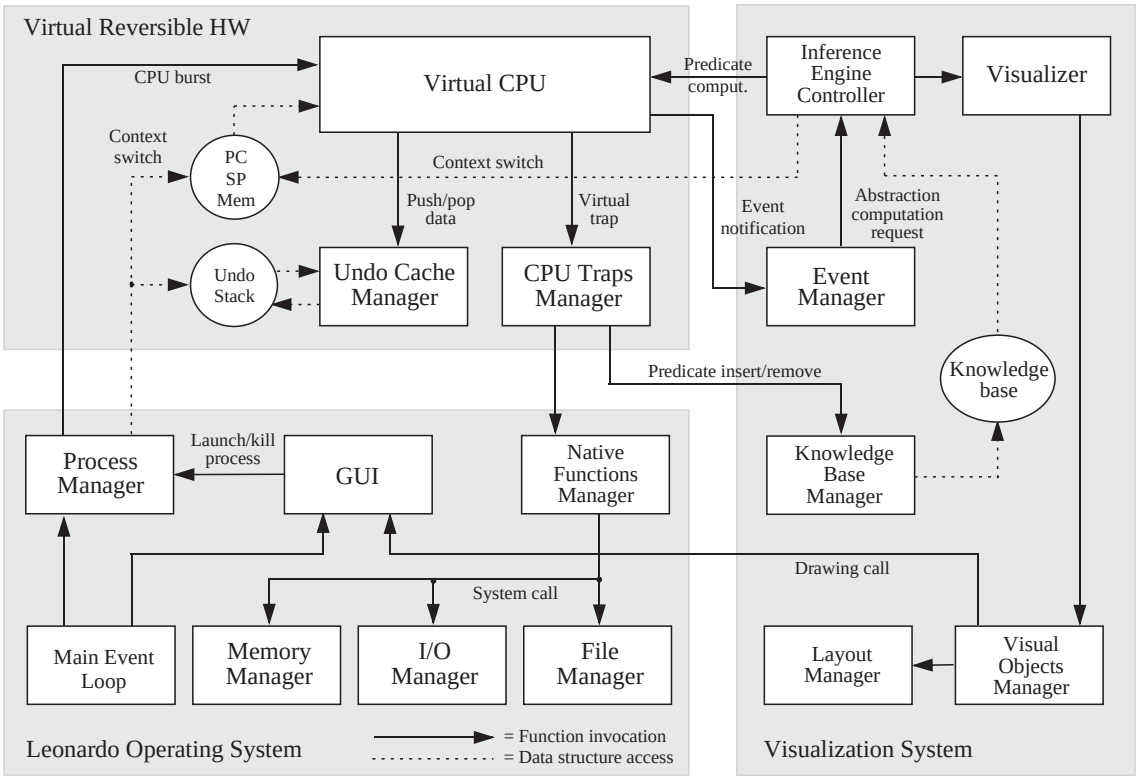

Figure 10: The components of LEONARDO's run-time environment.

With respect to reversibility, it is natural to split up the set of instructions into two groups, *destructive* and *non-destructive* ones, according to the fact they overwrite or not memory locations [28]. Reversing destructive instructions clearly requires to stack the information that is going to be destroyed. This may imply considerable memory demand: hence, both the possibility to inhibit reversibility and the optimization of memory requirements are valuable features.

In LEONARDO, it is possible to choose unidirectional executions when the ratio between program complexity and available secondary memory becomes unmanageable. Moreover, the information to be stored is put into a temporary file in secondary memory and the information flow is managed through a circular RAM-cache: while the virtual CPU is writing to (respectively, reading from) one half of the RAM-cache, the other half is being written to (respectively, read from) the temporary file in asynchronous way. In this way the execution speed is further optimized. However, some work remains to be done regarding both the optimization of the size of data to be stored and the possibility of reversing a group of instructions as it were an atomic macro-step.

6 The Visualization System

In this section we describe the last component of LEONARDO, that is, the *visualization system*.

In order to understand how this module works, let us first introduce a key concept: the notion of *event*. Basically, an event in LEONARDO is either the execution of an assignment instruction that changes the memory image of a process or the activation/deactivation of an ALPHA predicate, regardless of the direction of the execution. In a general setting, however, many other events may be treated: the click of the user on a check box belonging to the graphic user interface or the sending

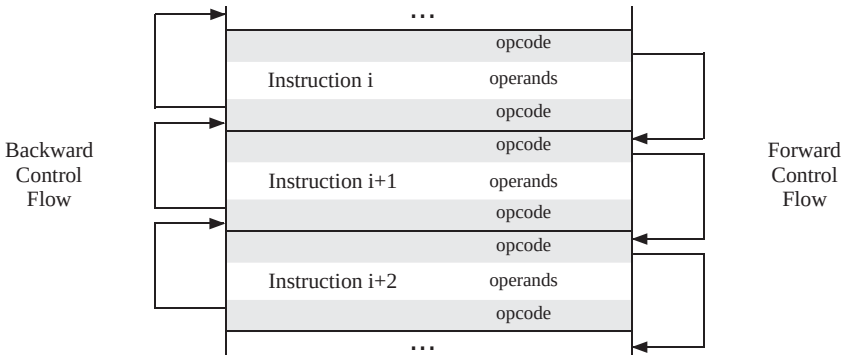

Figure 11: Symmetric control flows.

of a message from one process to another in a concurrent computation are just two examples.

An event is *significant* if it causes a change in the visual world: this may happen if and only if an active ALPHA predicate changes its output values with respect to its last computation. Obviously, not all events are significant: for example, if a variable is referred to by no ALPHA predicate, no change of its content will ever trigger a visualization event.

The *event manager* of LEONARDO's visualization system is responsible of deciding which events are significant and which ones can be ignored from a visualization point of view. When the virtual CPU (or any other module) generates an event, a message is sent to the event manager. If the event is recognized to be significant, the *inference engine controller* asks the virtual CPU to reevaluate only those predicates whose output values may have changed with respect to their last computation. This is done by sending a "virtual interrupt" to the virtual CPU that enters the supervisor mode. A context switch is performed in order to save the computation state of the underlying program execution that has been interrupted. ALPHA predicates are evaluated and their outcome is then sent to the *visualizer*, which is the back-end of LEONARDO's visualization system. This module redraws only really changed visual objects. When the visualizer terminates its task, the virtual CPU gets back into the user mode and the control returns to the interrupted program: a new context switch occurs and the original state is restored.

These operations are automatically executed by the system, providing the user with a real-time feedback of significant events that take place during the execution of a program (see Figure 12).

Since complete automation in generating animation events implies their maximum frequency, LEONARDO gives the user the possibility of skipping animation frames in order to have a sequence of images not excessively detailed. As an example, see the item swapping in the bubblesort code in Figure 8.

To conclude, it is worth noticing that, although inference and drawing computations are very time-consuming, experimental tests show that all optimizations performed by LEONARDO's visualization system greatly speed up animations and, in most cases, they have even to be slowed down by the user in order to be understandable.

7 An Animation Example

A complete example of the animation of a C program has been already given at the end of Section 4.1. Here we focus on the benefits of using a logic programming language, i.e. ALPHA, both to express formal properties of algorithms and to interpret concrete data structures. In particular,

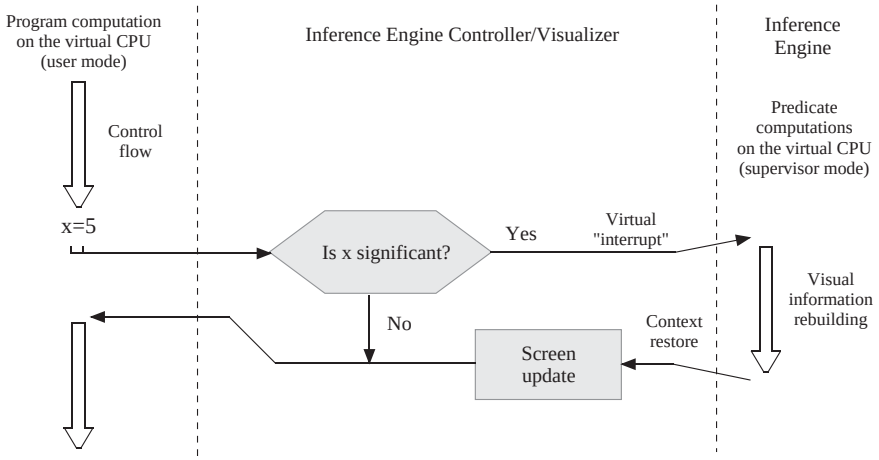

Figure 12: Management of significant events.

we present the visualization of a well-known algorithm, that is, the depth-first visit of a directed graph (see [10]). After a brief description of the data structures and the routines of the code, we enrich it with ALPHA predicates according to an incremental approach: starting from a simple visualization, we progressively add details making it more appropriate and significant. The complete implementation and visualization code is given in Appendix A. Two snapshots of the animation are shown in Figure 13 and in Figure 14.

7.1 C Implementation

Our recursive implementation of the depth-first visit is straightforward. The graph, whose number of nodes is contained in the variable `gNumNodes`, is represented by means of an adjacency matrix `g`, while the depth-first tree is held in the array `parent`. The algorithm also computes a depth-first numbering: the variable `theDFNNum` is incremented each time a new node is visited and the depth-first number of a node is stored in the array `DFN`. The whole code consists of just three functions: `LoadGraph`, `DFS`, and `main`.

`LoadGraph` uses some LEONARDO's library functions in order to get the input graph. The user is allowed to choose graphs either created with the graph editor or produced by other processes running in LEONARDO. The latter option allows the user to concatenate the actions of different processes as in a pipeline.

The function `DFS` visits the current node assigning it the current depth-first number `theDFNNum` and then scans all its adjacent nodes, recursively calling itself on those not yet visited.

The function `main`, finally, loads the graph, initializes the data structures, asks the node which the visit has to start from, and launches the visit on that node. It also launches the visits from nodes not reachable from the starting one, choosing them in increasing order. In other words, the visit does not necessarily produce a tree, but it may output a spanning forest.

7.2 Some Useful ALPHA Predicates

Before describing the visualization predicates, it is useful to build up a small library of non-standard predicates expressing logical and invariant properties of the algorithm in terms of its data structures. Such predicates are used in the rest of the code in order to make the visualization predicates more readable.

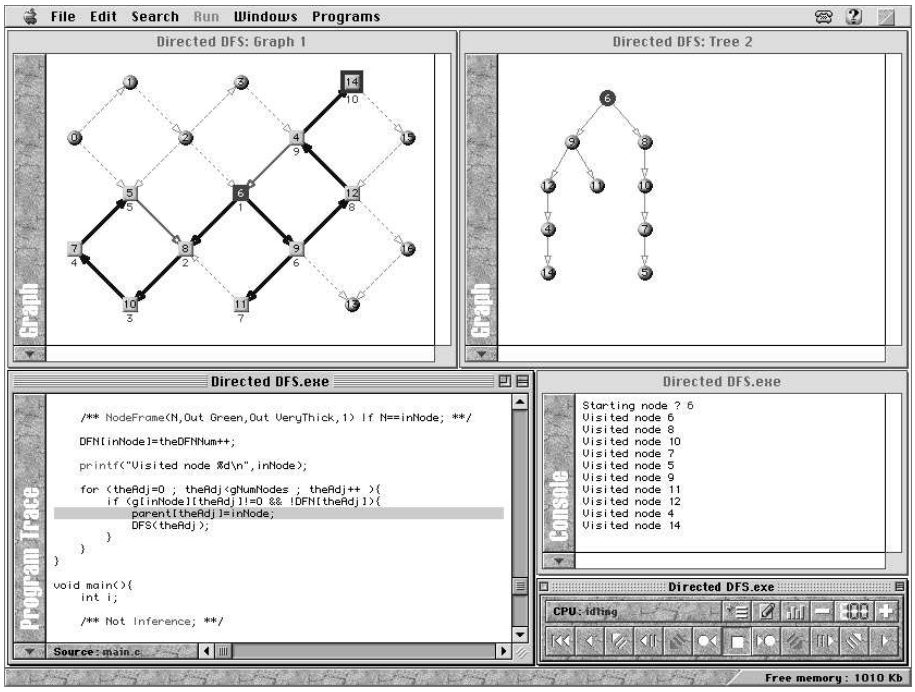

Figure 13: A snapshot of the execution.

7.2.1 The `TreeRoot` Predicate

The roots of the depth-first forest can be easily recognized looking at the array `parent`. Indeed, the parent of any non-root node is updated by the program with a positive value, while the parent of any root node maintains the initialization value (that is, `-1`) during the whole execution. According to this convention, the definition of a predicate that checks if a node is a root is straightforward:

```
TreeRoot(N) If parent[N]==-1;
```

7.2.2 The `Visited` Predicate

A node has been visited if its depth-first number has been set up to a positive value (note that the depth-first numbering starts from `1`). The definition of a predicate that checks if a node has been visited is thus the following:

```
Visited(N) If DFN[N]>0;
```

7.2.3 The `IsAncestor` Predicate

A node `X` is an ancestor of a node `Y` if `X` is `Y`'s parent or `X` is an ancestor of `Y`'s parent. A predicate that verifies the ancestor relation can be recursively defined as follows:

```
IsAncestor(X,Y) If X==parent[Y] Moreover
                If !TreeRoot(Y) && IsAncestor(X,parent[Y]);
```

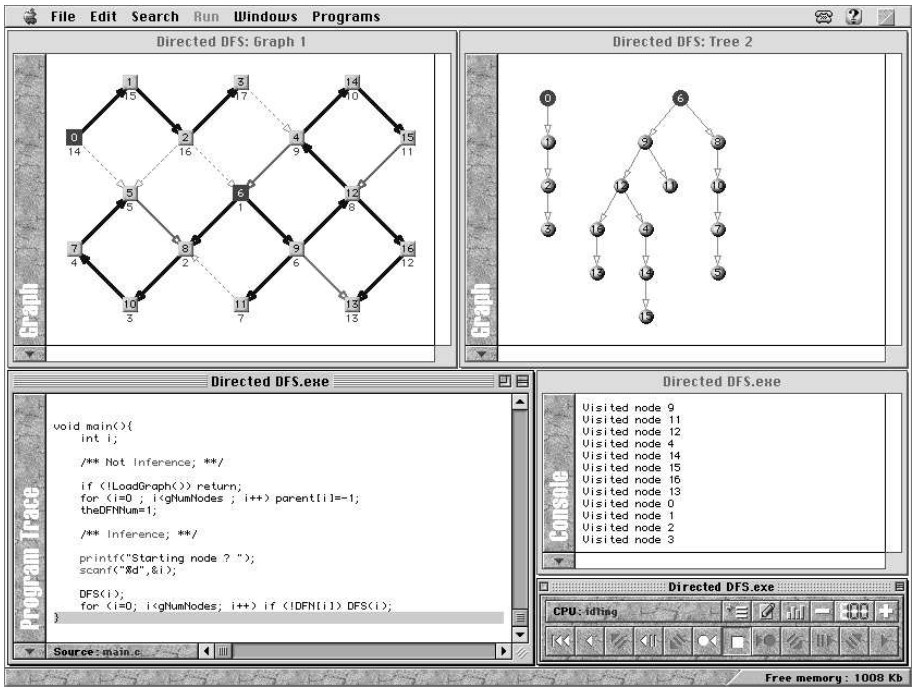

Figure 14: The final image.

7.2.4 Predicates for Partitioning Arcs

The depth-first visit induces a partition of arcs into four classes: (a) tree arcs, (b) forward arcs, (c) backward arcs, and (d) cross arcs. The following predicates describe the logical properties that realize this partition as presented in [10]:

```
TreeArc(X,Y)     If X==parent[Y];
ForwardArc(X,Y)  If Visited(X) && Visited(Y) && DFN[X]<DFN[Y] && !TreeArc(X,Y);
BackwardArc(X,Y) If IsAncestor(Y,X);
CrossArc(X,Y)    If !TreeArc(X,Y) && !ForwardArc(X,Y) && !BackwardArc(X,Y);
```

7.3 A Basic Visualization

We want our basic visualization to show a directed graph whose depth-first tree arcs are progressively highlighted and colored as the program runs. The following predicates accomplish this task:

```
Graph(Out 1);
Directed(1);
Node(Out N,1) For N:InRange(N,0,gNumNodes-1);
Arc(X,Y,1) If g[X][Y]==1;
ArcColor(X,Y,Out Blue,1) If TreeArc(X,Y);
```

These predicates declare a directed graph labeled `1`. Nodes of graph `1` are numbered from `0` to `gNumNodes-1`, and its arcs depend on the adjacency matrix `g`. Arcs in the depth-first tree automatically assume a blue color (the default color is red) as they are progressively added to the tree.

As a first refinement, in order to highlight the logical partition of arcs induced by the depth-first visit, we can modify the last predicate as follows:

```
ArcColor(X,Y,Out C,1)
    If TreeArc(X,Y)      Assign C=Blue        Moreover
    If ForwardArc(X,Y)   Assign C=Cyan        Moreover
    If BackwardArc(X,Y)  Assign C=LightGreen  Moreover
    If CrossArc(X,Y)     Assign C=Grey;
```

We can also modify the thickness and the style of the arcs by making tree arcs very thick, forward and backward arcs thick, and cross arcs dashed.

```
ArcThickness(X,Y,Out VeryThick,1) If TreeArc(X,Y);
ArcThickness(X,Y,Out Thick,1)     If ForwardArc(X,Y) || BackwardArc(X,Y);
ArcStyle(X,Y,Out Dashed,1)        If CrossArc(X,Y);
```

Finally, we highlight all visited nodes by giving them a squared shape and by labeling them with their depth-first numbers:

```
NodeShape(N,Out Square,1)     If Visited(N);
NodeLabel(N,Out Int,Out L,1)  If Visited(N)  Assign L=DFN[N];
```

All the visualization predicates described so far read the content of global variables and remain present in the declaration world during the whole execution. On the other hand, we would also like to know which node is being currently visited. This information is stored in the formal parameter `inNode` of the function `DFS`, whose content changes at each recursive call. Hence, at the very beginning of the `DFS`'s body, we can add the following predicate that surrounds the currently visited node with a green thick frame:

```
NodeFrame(N,Out Green,Out VeryThick,1) If N==inNode;
```

7.4 A Refined Visualization

In order to further refine the visualization, it may be useful to see the tree not just growing on the graph, but also in a separate view and with a typical downward tree layout. At this aim, we define the following predicates:

```
Tree(Out 2);
Directed(2);
Node(Out N,2) For N:Node(N,1) If Visited(N);
Arc(X,Y,2) If TreeArc(X,Y);
ArcColor(X,Y,Out Grey,2);
```

These predicates declare a directed tree labeled `2`. There is a node in the tree for each visited node in the graph. Arcs are defined according to the outcome of the non-standard predicate `TreeArc` and all of them are grey.

Observe that the declaration of a tree is very similar to that of a graph, but using the predicate `Tree` instead of `Graph` forces the visualization system to apply a variant of the tree drawing algorithm described in [24].

At this point, we could decide to make the roots of the depth-first forest red:

```
NodeColor(N,Out Red,_) If TreeRoot(N) && Visited(N);
```

The underscore acts as a wild card meaning "in each tree and in each graph". In our case, the roots become red both in graph `1` and in tree `2`.

8 Conclusions

In this paper we presented the logic and the architecture of LEONARDO, an integrated environment for software visualization. We highlighted that its main features concern both the reversible execution of programs and the use of a logic-based language for specifying visualizations.

LEONARDO has been implemented for the MacOs by using the C++ language within the Metrowerks PowerPlant framework. The latest version of LEONARDO is currently available over the Internet at the URL `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

We are currently testing the usability of LEONARDO on undergraduate classes: students are allowed both to observe animations and to create from scratch new ones. According to this experimentation, the system appears simple to understand and to use, but still lacks in some traditional valuable algorithm animation features, such as smooth animations, interactivity, and representations of the computation history.

In particular, the first two features only need to be formalized and implemented, as they do not conflict with LEONARDO's logic approach. On the contrary, the concept of history of computation appears not to be captured by the instantaneous atemporal predicates' logic; however, the possibility of reversing the execution partially makes up for this lack.

Another interesting open problem is related to the debugging of ALPHA predicates. Indeed, both the program and the visualization languages are prone to errors, making sometimes debugging tricky. Up to this time, LEONARDO is able to detect only some logical errors, such as the declaration of arcs of a tree forming a cycle. An improved error localization mechanism could represent a valuable contribution.

Since at this moment we consider our system a work in progress, perhaps the interested reader will find LEONARDO's WEB site up to date.

Acknowledgments

We thank all LEONARDO's users. We are also grateful to Giuseppe Di Battista, Walter Didimo, Giuseppe Liotta, Umberto Nanni, Maurizio Patrignani, Adolfo Piperno, Maurizio Pizzonia, and Andrea Sterbini for encouraging our work with their friendly appreciation. We finally thank the anonymous referees: their comments were interesting and appropriate, and their suggestions were really useful for improving the quality of the paper.

References

[1] J.E. Baker, I. Cruz, G. Liotta, and R. Tamassia. Algorithm Animation over the World Wide Web. In *Proceedings of the 1996 ACM Workshop on Advanced Visual Interfaces*, pages 203–212, 1996.

[2] G. Bongiovanni, P. Crescenzi, and G. Rago. JAZ: Java Algorithm Visualizer. A Multi-Platform Collaborative Tool for Teaching Graph Algorithms. In *Proceedings of the 6th International Conference in Central Europe on Computer Graphics and Visualization (WSCG'98)*, pages 73–80, 1997.

[3] M.H. Brown. *Algorithm Animation*. MIT Press, Cambridge, MA, 1988.

[4] M.H. Brown. Zeus: a System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pages 4–9, 1991.

[5] M.H. Brown and J. Hershberger. Color and Sound in Algorithm Animation. *Computer*, 25:52–63, 1992.

[6] M.H. Brown and M. Najork. Algorithm Animation Using 3D Interactive Graphics. In *Proceedings of the 1993 ACM Symposium on User Interface Software and Technology*, pages 93–100, 1993.

[7] M.H. Brown and M. Najork. Collaborative Active Textbooks: a Web-Based Algorithm Animation System for an Electronic Classroom. In *Proceedings of the 1996 IEEE International Symposium on Visual Languages*, pages 266–275, 1996.

[8] M.H. Brown and R. Sedgewick. A System for Algorithm Animation. In *Proceedings of ACM SIGGRAPH'84*, pages 177–186, 1984.

[9] M.H. Brown and R. Sedgewick. Techniques for Algorithm Animation. *IEEE Software*, 2:28–39, 1985.

[10] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[11] C. Demetrescu. Un Ambiente Integrato per lo Sviluppo e l'Animazione di Algoritmi: il Sistema di Visualizzazione. Master's thesis, University of Rome "La Sapienza", Italy, 1997.

[12] C. Demetrescu and I. Finocchi. A General-Purpose Logic-Based Visualization Framework. In *Proceedings of the 7th International Conference in Central Europe on Computer Graphics, Visualization and Interactive Digital Media (WSCG'99)*, pages 55–62, 1999.

[13] C. Demetrescu and I. Finocchi. A Technique for Generating Graphical Abstractions of Program Data Structures. In *Proceedings of the 3rd International Conference on Visual Information Systems, LNCS 1614*, pages 785–792, 1999.

[14] J. Domingue and P. Mulholland. Teaching Programming at a Distance: The Internet Software Visualization Laboratory. *Journal of Interactive Media in Education*, 7, 1997. Available at http://www-jime.open.ac.uk/97/1/.

[15] I. Finocchi. Un Ambiente Integrato per lo Sviluppo e l'Animazione di Algoritmi: la Macchina Virtuale ed i Compilatori. Master's thesis, University of Rome "La Sapienza", Italy, 1997.

[16] J. Haajanen, M. Pesonius, E. Sutinen, J. Tarhio, T. Teräsvirta, and P. Vanninen. Animation of User Algorithms on the Web. In *Proceedings of the 1997 IEEE International Symposium on Visual Languages*, pages 360–367, 1997.

[17] R.R. Henry, K.M. Whaley, and B. Forstall. The University of Washington Program Illustrator. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 223–233, 1990.

[18] S.P. Lahtinen, E. Sutinen, and J. Tarhio. Automated Animation of Algorithms with Eliot. *Journal of Visual Languages and Computing*, 9:337–349, 1998.

[19] A.W. Lawrence, A.N. Badre, and J.T. Stasko. Empirically Evaluating the Use of Animations to Teach Algorithms. In *Proceedings of the 1994 IEEE International Symposium on Visual Languages*, pages 48–54, 1994.

[20] H. Lieberman. The Debugging Scandal and What to Do About It. *Communications of the ACM*, 40:27–29, 1997.

[21] H. Lieberman and C. Fry. Zstep95: A Reversible, Animated Source Code Stepper. In : [34], pages 277–292.

[22] B.A. Myers. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1:97–123, 1990.

[23] B.A. Price, R.M. Baecker, and I.S. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4:211–266, 1994.

[24] E. Reingold and J. Tilford. Tidier Drawings of Trees. *IEEE Transactions on Software Engineering*, 7(2):223–228, 1981.

[25] G.C. Roman and K.C. Cox. A Declarative Approach to Visualizing Concurrent Computations. *Computer*, 22:25–36, 1989.

[26] G.C. Roman and K.C. Cox. A Taxonomy of Program Visualization Systems. *Computer*, 26:11–24, 1993.

[27] G.C. Roman, K.C. Cox, C.D. Wilcox, and J.Y Plun. PAVANE: a System for Declarative Visualization of Concurrent Computations. *Journal of Visual Languages and Computing*, 3:161–193, 1992.

[28] C. Ruknet. The Design of a Processor Architecture Capable of Forward and Reverse Execution. In *Proceedings of SoutheastCon 91*, 1991.

[29] M. Sarkar and M.H. Brown. Graphical Fisheye Views. *Communications of the ACM*, 37:73–84, 1994.

[30] J.T. Stasko. The Path-Transition Paradigm: a Practical Methodology for Adding Animation to Program Interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, 1990.

[31] J.T. Stasko. TANGO: A Framework and System for Algorithm Animation. *Computer*, 23:27–39, 1990.

[32] J.T. Stasko. A Methodology for Building Application-Specific Visualizations of Parallel Programs. *Journal of Parallel and Distributed Computing*, 18:258–264, 1993.

[33] J.T. Stasko. Using Student-Built Algorithm Animation as Learning Aids. In *Proceedings of the 1997 ACM SIGCSE Conference*, pages 25–29, 1997.

[34] J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price. *Software Visualization*. MIT Press, Cambridge, MA, 1997.

[35] J.T. Stasko and A. Lawrence. Empirically Assessing Algorithm Animations as Learning Aids. In : [34], pages 419–438.

[36] J.T. Stasko and J. Muthukumarasamy. Visualizing Program Executions on Large Data Sets. In *Proceedings of the 1996 IEEE International Symposium on Visual Languages*, pages 166–173, 1996.

[37] D. Ungar, H. Lieberman, and C. Fry. Debugging and the Experience of Immediacy. *Communications of the ACM*, 40:38–43, 1997.

A The Depth-First Visit Code

```
#include <stdio.h>

#define MAX_NODES 100

int gNumNodes;
char g[MAX_NODES][MAX_NODES];
int parent[MAX_NODES];
int DFN[MAX_NODES];
int theDFNNum;

/**
    TreeRoot(N)       If parent[N]==-1;
    Visited(N)        If DFN[N]>0;
    IsAncestor(X,Y)   If X==parent[Y] Moreover
                      If !TreeRoot(Y) && IsAncestor(X,parent[Y]);

    TreeArc(X,Y)     If X==parent[Y];
    ForwardArc(X,Y)  If Visited(X) && Visited(Y) && DFN[X]<DFN[Y] && !TreeArc(X,Y);
    BackwardArc(X,Y) If IsAncestor(Y,X);
    CrossArc(X,Y)    If !TreeArc(X,Y) && !ForwardArc(X,Y) && !BackwardArc(X,Y);

    Graph(Out 1);
    Directed(1);

    Node(Out N,1)                For N:InRange(N,0,gNumNodes-1);
    NodeColor(N,Out Red,_)       If TreeRoot(N) && Visited(N);
    NodeShape(N,Out Square,1)    If Visited(N);
    NodeLabel(N,Out Int,Out L,1) If Visited(N) Assign L=DFN[N];

    Arc(X,Y,1) If g[X][Y]==1;
    ArcColor(X,Y,Out C,1)
        If TreeArc(X,Y)      Assign C=Blue        Moreover
        If ForwardArc(X,Y)   Assign C=Cyan        Moreover
        If BackwardArc(X,Y)  Assign C=LightGreen  Moreover
        If CrossArc(X,Y)     Assign C=Grey;

    ArcThickness(X,Y,Out VeryThick,1) If TreeArc(X,Y);
    ArcThickness(X,Y,Out Thick,1)     If ForwardArc(X,Y) || BackwardArc(X,Y);
    ArcStyle(X,Y,Out Dashed,1)        If CrossArc(X,Y);

    Tree(Out 2);
    Directed(2);
    Node(Out N,2) For N:Node(N,1) If Visited(N);
    Arc(X,Y,2) If TreeArc(X,Y);
    ArcColor(X,Y,Out Grey,2);
**/
```

```
int LoadGraph(){
    int i;
    if (!OpenGraph()) return 0;
    gNumNodes=GetNodesCount();
    for (i=0; i<GetArcsCount(); i++) {
        long theStart,theEnd;
        GetArc(i,&theStart,&theEnd);
        g[theStart][theEnd]=1;
    }
    CloseGraph();
    return 1;
}

void DFS(int inNode){
    int theAdj;

    /** NodeFrame(N,Out Green,Out VeryThick,1) If N==inNode; **/

    DFN[inNode]=theDFNNum++;

    printf("Visited node %d\n",inNode);

    for (theAdj=0; theAdj<gNumNodes; theAdj++){
        if (g[inNode][theAdj]!=0 && !DFN[theAdj]){
            parent[theAdj]=inNode;
            DFS(theAdj);
        }
    }
}

void main(){
    int i;

    /** Not Inference; **/

    if (!LoadGraph()) return;
    for (i=0; i<gNumNodes; i++) parent[i]=-1;
    theDFNNum=1;

    /** Inference; **/

    printf("Starting node ? ");
    scanf("%d",&i);

    DFS(i);
    for (i=0; i<gNumNodes; i++) if (!DFN[i]) DFS(i);
}
```