Dynamic inheritance: a powerful mechanism for operating system design

Benoît Sonntag, Dominique Colnet and Olivier Zendra

LORIA — INRIA Lorraine
615 Rue du Jardin Botanique, BP 101,
54602 Villers-Lès-Nancy Cedex
FRANCE
Email: {bsonntag, colnet, zendra}@loria.fr

Abstract

The design of the Isaac operating system comes from several years of reflexion and implementation on the need for flexibility and dynamism in future operating systems. Our goals progressively lead us towards the object-oriented concepts. Prototype-based languages appeared the most elegant manner to materialize our vision of operating system. These, coupled with a powerful language allowing changing inheritance dynamically, made it possible to create the innovating Isaac OS (http:www.IsaacOS.com).

Keywords: object-oriented language, prototype-based language, operating system, dynamic inheritance, Isaac

1 Introduction: the Isaac OS project

The very nature of current operating systems comes from studies, languages, hardware and requirements going back to about 20 years.

The evolution of programming languages currently fulfills nowadays data-processing needs and constraints in terms of software design and production. However, modern languages (i.e. object-oriented languages), did not bring a real alternative to procedural programming languages like C in the development of modern operating systems. These OSes require high performance in terms of execution speed and memory usage, but also simple, efficient, internal low-level operations. We thus believe that an object-oriented operating system should not be on top of a virtual machine, but directly installed on hardware components, to reach the very best performance. It is desirable and possible to fully use the hardware in order to provide, at the operating system level, services that are currently supplied by software layers (type-oriented file system, dynamic graphic management, dynamic link between software components, common memory buffers, ...).

Historically, during the creation of an operating system, constraints related to hardware programming have been systematically fulfilled with a low-level language as the C language. This choice generally leads to a lack of flexibility that can be felt at the application level.

The purpose of our Isaac OS project ([4]) is to break with the internal rigidity of current OS architecture that mainly depends, in our opinion, on the low-level languages that have been used to write them. Conversely, Isaac has been fully written with a high-level prototype-based language and makes full use of its powerful features. Although it is conceptually close to the Merlin project (see [2]), Isaac goes into a different direction. For example, it does not rely on a virtual machine, but is compiled. Moreover, we included in Isaac a hardware-enforced protection mechanism (see [5]) and a distinction between two kinds of objects, macro-objects and micro-objects (see [6]).

We thus decided to create a new object-oriented, prototype-based language called Lisaac (see [6]),

providing extra facilities for the implementation of operating systems, such as privilege levels controlled at the processor level, and dynamic inheritance. Lisaac is a powerful tool for the creation of an efficient, flexible and cleanly design operating system. In comparison with Self, our Lisaac language is compiled, not executed on a VM. The quality of the binary code produced by a system-wide analysis allows us to envision good performance for the whole system.

In this paper, we speak mainly about dynamic inheritance. The possible problems on the level of the abnormal inheritance are partly detected during compilation. The other problems are collected at runtime during the lookup algorithm. The use of the material exceptions makes it possible to manage the integrity of the system.

Our addition of flexibility with the dynamic inheritance is another approach of the reflection in class-based languages (see [1]). The performances of reflexivity are often to question (see [3]).

In the following sections, we focus on inheritance – traditional inheritance and dynamic inheritance as well – with three examples of its use in Lisaac for the implementation of parts of the Isaac OS.

2 Hardware components *versus* software components

In the Isaac OS, each hardware component, like for example a mouse or a screen is represented by its corresponding prototype. When one wants to add (or remove) some new hardware component at runtime, this is achieved by adding (or removing) the corresponding prototype simultaneously. What is true for hardware object is also true for software components. As an example a file or a bitmap object is represented by its corresponding prototype (one prototype for each file and one prototype for each bitmap object). For example, removing some file is simply performed by removing the corresponding prototype from the universe.

As one may expect, the traditional inheritance mechanism is useful to organize things, especially for operating system design. In the object hierarchy for the Isaac system, true physical "*hardware objects*" (keyboard, mouse, memory, ...) are distinct from "*software objects*" (file, vector, bitmap, ...). In a very natural way, traditional inheritance is used to separate these hardware objects from software objects. The reason of this segregation is that hardware objects are not clonable. For example, a screen object can't be cloned if there is no new physical screen. Hardware components are obviously critical resources. Conversely, software components, SOFT_OBJECT (fig. 1), inherit the traditional `Clone` method. Also note that that both SOFT_OBJECT and HARD_OBJECT inherit the most common OBJECT prototype.

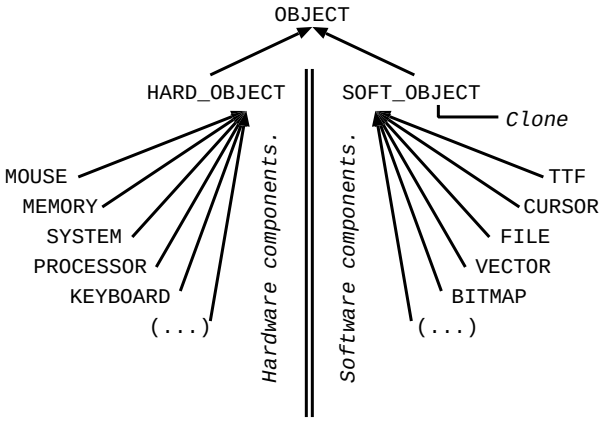

Figure 1: Segregation between hardware components and software components

Traditional inheritance (like the one that can be found in class-based languages e.g. Java) is largely enough to implement segregation between hardware and software objects. As we will see in the next examples, when the organization is not as simple, dynamic inheritance, if not necessary, appears to be a powerful tool.

3 Dynamic inheritance for video drivers

Figure 2 represents the Isaac OS video architecture. The VIDEO object can change dynamically its parent slot to inherit BITMAP_15 or BITMAP_16 or BITMAP_24 or BITMAP_32 as well. Actually, dynamic inheritance is obviously used to dynamically change the bitmap resolution. The reference to the VIDEO object remains unchanged for clients allowing the resolution mode to be changed transpar-

ently (i.e. only the parent slot of the VIDEO object is modified).

Moreover, the VIDEO object can redefine any BITMAP functionalities in order to take advantage as much as possible of the hardware graphic device (graphic accelerator embedded on the board, color depth, ...).

The VIDEO object represents the hardware driver for the physical video card. This object representation as numerous advantages:

- The methods called by a client on a SCREEN object, which inherits from VIDEO, naturally use the methods of the appropriate VIDEO type. As the VIDEO object represents the video driver, these methods are the most adapted implementation to the physical video card.
- When the VIDEO object doesn't redefine a method A (such as RECTANGLE) but redefines a method B which is used by A (such as LINE), late binding causes the use of the method B of VIDEO while using the method A of one of its parents (BITMAP_15 or BITMAP_16 ...). This has for interesting consequence the maximal use of the performances of the video card at every time.
- Another example, also related to inheritance use in the graphic management, is that if we add methods in the base object BITMAP, these methods will be automatically usable by our client object.

Lisaac offers the capacity to *dynamically* define or redefine the parents of an object (see [6]). When we define our client object SCREEN, the parent slot is a block of code, which finds the reference object representing the video driver. This block of code is evaluated when the client object SCREEN is loaded. Thus, the SCREEN object inherits the current video card.

4 Dynamic inheritance for file systems

The design of the file system is an important part of an OS design. The Isaac OS does not define its own disk format but uses existing ones: the windows FAT or NTFS format or the Unix format based on

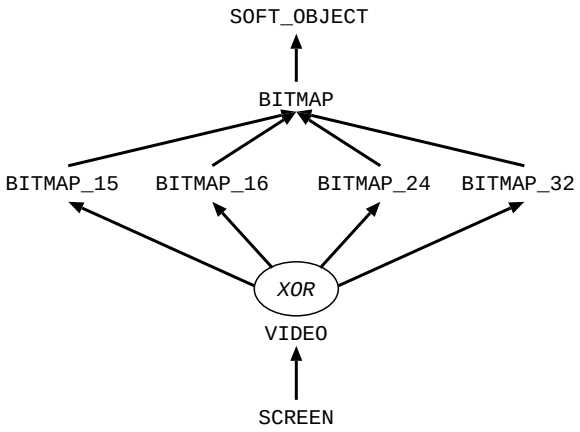

Figure 2: Dynamic inheritance to select the appropriate VIDEO DRIVER.

inodes and blocks. As for all other system the disk itself can be handled by a floppy disk controller or with an IDE controller for example.

Once again, dynamic inheritance appears to be a powerful tool to implement this aspect. As shown on figure 3, a FILE object or a DIRECTORY object inherit the abstract system-independent INODE object. This abstract INODE object may inherit FAT_16BITS when the corresponding file is on a windows partition or may inherit EXT_2FS when the corresponding file is on a Linux partition.

A comparable dynamic inheritance scheme is used to determine the drive controller. Thus, the inheritance link follows FLOPPY prototype when the file is on some floppy disk or follows the IDE prototype when the file is stored on some hard disk. Thanks to dynamic inheritance, this link may change at run time when the file is moved from the floppy disk to the hard disk and conversely.

As explained previously each file is represented by one prototype. This is also true for directories (each directory is associated with its own prototype). Furthermore, the prototype structure is used to add extra high-level information for files or directories.

As an example, the content of a directory (i.e. entries of the directory) is represented using one slot for each entry of the directory. Thus, a DIRECTORY which contains one file "foo" and one subdirectory "bar" has two corresponding slots, one to

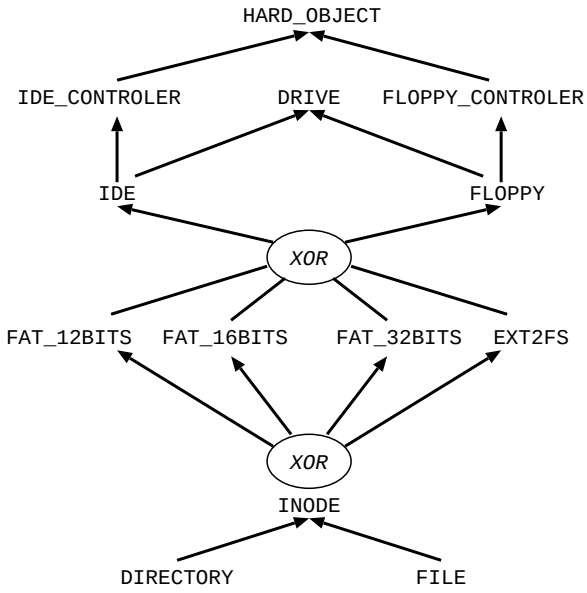

Figure 3: File system selection with dynamic inheritance

access the "foo" file and another to access the "bar" subdirectory. The prototype structure is directly mapped on the content of the directory. Access into the hierarchy of directories is achieved using normal slot read-write operations.

Extra slots are also added for file prototypes in order to add high-level semantic operations. First the '`buffer`' slot is always available for every FILE prototypes. This slot gives access to the raw binary content of the file. If the file has to be viewed as a text file, a parent '`txt`' slot can be added. Actually, in order to avoid data redundancy, this '`txt`' slot is a third party object which convert the raw binary '`buffer`' information into traditional text line information.

In the same way, it is possible to add other parent objects to handle extra high-level information. As another example, if a file can also be viewed as an HTML file, the '`html`' slot is simply added to the parent list. This way, the file can then be manipulated easily by a WEB browser.

File and directory compression is also handled by this powerful mechanism. For example, when some file is compressed, a '`zip`' parent is added dynamically to the FILE prototype. Note that the memory address of this FILE prototype is not changed. The user of the file can still manipulate the content of the file without knowing the fact that the file has been compressed!

For users, all the files are accessible transparently via a cache system. Of course, only the files actually in use are represented in memory at a given time.

5 Conclusion

The set up of our Isaac operating system, led us to conceive a new object-oriented language, called Lisaac [6]. This Lisaac language, not described in this paper, is uniform and very close to Self [7], another prototype-based language.

Lisaac constitutes a powerful tool in the making of an efficient, flexible, and cleanly design operating system. The architecture of our object operating system takes fully advantage of the possibilities offered by prototypes and especially by dynamic inheritance.

We have presented in this paper three main examples of inheritance usage from the Isaac operating system implementation. The first example of section 2 use traditional, non-dynamic, inheritance to separate hardware objects from software components. The second example of section 3 present dynamic inheritance used to implement graphical video operations. Finally, the last example at section 4 introduce the important problem of file system management.

Thanks to the Isaac OS experiment, we are now convinced that a prototype-based language like Lisaac is a very good candidate to replace low-level languages like C for operating system implementation.

Currently, Isaac is still very much in development. Our prototype is thus not yet sufficiently advanced to carry out meaningful performance tests. We thus still have to test the real performances of Isaac OS compared to other systems.

References

[1] Daniel Barbou. Inheritance Hierarchy Automatic (Re)organization and Prototype-based languages. In *Workshop 16, Objects and clas-*

sification : a natural convergence, 14th European Conference on Object-Oriented Programming, (ECOOP 2000), 2000.

[2] E. Morais J. Mattos de Assumpcao, L. Campos de Carvalho. `http://www.lsi.usp.br/ jecel/merlin.html`. Site web: Merlin Home Page, The merlin Object-Oriented System., 1998.

[3] Francisco Ortn Soler and Juan Manuel Cueva Lovelle. Building a Completely Adaptable Reflective System. In *4th ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'2001)*, 2001.

[4] B. Sonntag. `http://www.isaacOS.com`. Site web: Isaac (Object Operating System)., 2000.

[5] B. Sonntag. Article in French about: Usage of the processor memory segmentation with a high-level language. In *2ime Confrence Franaise sur les systmes d'Exploitation, (CFSE'2)*, pages 107–116. ACM Press, 2001.

[6] B. Sonntag and D. Colnet. Lisaac: the power of simplicity at work for operating system. In *Technology of Object-Oriented Languages and Systems, (TOOLS 2002)*, volume 10, pages 45–52. Australian Computer Society Press, 2002.

[7] D. Ungar and R. Smith. Self: The Power of Simplicity. In *2nd Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 227–241. ACM Press, 1987.