Lisaac: the power of simplicity at work for operating system

Benoît SONNTAG, Dominique COLNET

E-mail: {bsonntag, colnet}@loria.fr
LORIA
UMR 7503
(INRIA - CNRS - University Henri Poincaré)
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex
FRANCE

Abstract

The design as well as the implementation of the Isaac operating system [Son00] led us to set up a new programming language named Lisaac. Many features from the Lisaac language come from the Self programming language [US87]. Comparing to Self's skills, Lisaac integrates communications protection mechanisms as well as other tools related to operating systems' design. System interruptions support as well as drivers memory mapping have been considered in the design of Lisaac. The use of prototypes and especially dynamic inheritance, fits a flexible operating system in the making. First benchmarks of our compiled objects show that it is possible to obtain high-level prototype-based language's executables as fast as C programs are.

1 Introduction

The very nature of current operating systems comes from studies, languages, hardware and needs going back to a score of years. The purpose of our project is to break with the internal rigidity of current operating systems architecture that mainly depends, in our opinion, on low-level languages that have been used to write them. Our Isaac operating system has been fully written with a high-level prototype-based language.

The evolution of computer's programming languages currently fulfills nowadays data-processing's needs and constraints in terms of software conception and production. Nevertheless, modern languages (i.e. object-oriented languages), did not bring a real alternative to procedural programming languages as the C language is in the development of a modern operating system. They require high performances in terms of execution speed and memory usage, but also simple inner low-level operations. We also believe that the object-oriented operating system must not be on top of a virtual machine but directly installed on hardware components. Indeed, it is essential to be able to reach the very best performances. It is desirable and possible to fully use the hardware in order to provide, at the operating system's level, services that are currently supplied by software layers (garbage collector, inter process communication, common memory buffers, ...).

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

Our thought process led us to set up a new object-oriented language with extra facilities useful for the implementation of an operating system. In order to achieve that point, we started to look for an existing object-oriented language with powerful characteristics in terms of flexibility and expressiveness. Actually, two languages are at the origin of Lisaac: the Self language [US87] for its flexibility and the Eiffel language [Mey94] for its static typing and security. Our language comes also from an experiment in the making of an operating system based on dynamic objects; its possibilities are a subtle mix of Self with Eiffel and we add it some low-level capabilities of the C language. Compared to Self, it is a little bit limited especially in the way that source files are compiled. From the Eiffel language, we borrowed a kind of static typing form. As for it, the implementation is particularly original on many points.

The remainder of this paper is organized as follows. section 2 describes the environment of Isaac, the compilation and the execution of objects in memory. Then, section 3 takes place with a description of the Lisaac language, followed by its semantics described in section 4. Next, section 5 will validate our model with relevant examples of the system's implementation diagrams as well as the first benchmarks of the compiler. We conclude in section 6.

2 The Lisaac language and the Isaac system: an overview

It is clear for us that an operating system must use all the power of hardware components. For this reason, the Isaac operating system does not run on top of a virtual machine but directly on hardware components (i.e. directly on the micro-processor), hence the choice of a compiler approach to avoid – at run time – the interpretor overhead.

As in the case of the Eiffel language, an Isaac object is defined by its source code stored in a file of the same name as the object it defines. As in the traditional approach, this file is processed by our Lisaac compiler in order to produce the corresponding executable binary file.

To facilitate the portability of the system, our compiler operates closely with GCC [sc00] and the ELF [elf95] binary linker. A compatibility at the source level for applications written in C has been studied, but the mechanisms involved are not in the scope of this article, nevertheless, we want to stress out that the interfacing used remains coherent with the Isaac object model.

As in Self, a prototype may have a name or can be an anonymous one. All prototypes are clonable and may be modified at runtime. Even the inheritance relationship between prototypes may change at runtime.

Mostly for security reasons as well as for organization of our model, there are two kinds of objects: PRIVATE objects (also called micro-objects) and other objects (called macro-objects). A macro-object is usually used to represent a complex object like some hardware component (e.g. mouse, keyboard, ...) or some complex software component (file, bitmap, ...). Micro-objects are PRIVATEs and are composed in order to constitute some macro-object. Micro-objects are usually simple (e.g. integer, string, ...) and are used for macro-objects implementation. Nevertheless, micro-objects may represent more complex data like linked lists or dictionaries for example.

Actually, as we will see later, the syntax to describe macro-objects and micro-objects is the same. The major difference comes from the compilation strategy used (fig. 1). Each macro object is compiled separately to produce the corresponding executable object. All interactions between macro-objects are dynamically typed (i.e. no verification at compile time). Inside some macro-object, all PRIVATE objects are checked statically using an Eiffel-like approach [ZCC97] [CZ99].

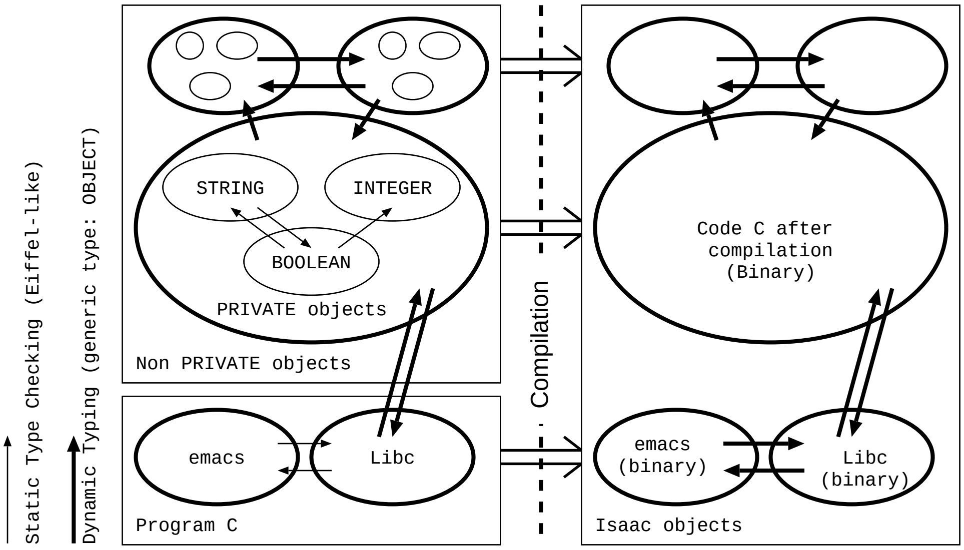

Figure 1. Only PRIVATE objects are checked statically when other objects are typed dynamically.

Thus, interactions between macro-objects are checked only at runtime. For example, on the Intel architecture [Pro], this is achieved using hardware mechanism. Interactions between micro-objects are checked only at compile time. Those objects are compiled in the context of some macro-object which use them and do not require the creation of a genuine Isaac prototype as such. Finally, to ease the implementation of containers like arrays, linked lists and dictionaries for example, we also added a form of genericity such as the one defined in Eiffel [Mey94].

Dynamic typing allows flexible interaction or communication between operating system components. In Lisaac source code, all macro-objects are all declared with the type mark OBJECT, the most general prototype.

3 Lexical and syntax overview

Most features of the Lisaac language come from Self. Like Self, Lisaac does not have hard-coded instructions for loops or hard-coded instructions for test statements.

The following syntax of Lisaac is described using "Extended Backus-Naur Form" (EBNF). Terminal symbols are enclosed in single quotes or are written using lowercase letters. Non-terminal are written using uppercase letters. The following table describes the semantic of meta-symbols used:

Meta-symbol	Function	Description
(...)	grouping	a group of syntactic constructions
[...]	option	an optional construction
{ ... }	repetition	a repetition (zero or more times)
|	alternative	separates alternative constructions
→	production	separates the left and right hand sides of a production

The following table draws up the list of the final syntactic elements of the grammar:

Symbol	Description	Syntax
identifier	type, slot name, ...	[A-Za-z_][A-Za-z0-9_]*
characters	constant of CHARACTER type	'*ASCII string*'
string	constant of STRING type	"*ASCII string*"
external	external C code	‘*ASCII string*‘
integer	constant of INTEGER type	0x[0-9A-Fa-f]+ ou 0[0-7]* ou [1-9][0-9]*
operator	unary/binary operator symbol	[!@#$%∧&<|*+-=∼/?> \ ]*

In order to clarify the presentation for human reading, the following grammar of Lisaac is ambiguous. (Actually, the Lisaac parser use precedence and associativity rules to resolve ambiguities.) The Lisaac grammar is immediately followed by an example to illustrate the syntax:

```
PROGRAM        →  { 'section' identifier { SLOT } }
SLOT           →  TYPE_SLOT [ ':' TYPE ] [ '=' { '+' LOCAL ';' } EXPR ] ';'
TYPE_SLOT      →  identifier [ L_LOCAL { identifier L_LOCAL } ]
               |  operator [identifier ':' TYPE ] [ ASSOCIATIVITY ]
ASSOCIATIVITY  →  ( 'left' | 'right' ) [ integer ]
L_LOCAL        →  { LOCAL ',' } LOCAL
LOCAL          →  { identifier ',' } identifier ':' TYPE
TYPE           →  identifier [ '[' { identifier ',' } identifier ']' ]
EXPR           →  { EXPR_PREFIX operator } EXPR_PREFIX
EXPR_PREFIX    →  { operator } EXPR_MESSAGE
EXPR_MESSAGE   →  EXPR_BASE { '.' SEND_MSG }
EXPR_BASE      →  EXPR_PRIMARY | SEND_MSG
ARGUMENT       →  EXPR_PRIMARY | identifier
EXPR_PRIMARY   →  integer | characters | string | external
               |  '(' EXPR ')' | '{' { EXPR ';' } '}'
SEND_MSG       →  TYPE [ L_ARGUMENT { identifier L_ARGUMENT } ]
L_ARGUMENT     →  { ARGUMENT ',' } ARGUMENT
```

The quick sort example in Lisaac:

```
 section HEADER
  name = QUICKSORT;
  category = PRIVATE;
  date = "Oct 28 2001";
  version = 1;
  comment = "Example for TOOLS.";
  author = "Benoît Sonntag.";

section PRIVATE
  size = 20000000;
  tableau:NATIVE_ARRAY[CHARACTER];
```

```
my_qsort tab:NATIVE_ARRAY[CHARACTER] from low:INTEGER to high:INTEGER =
  + i, j : INTEGER;
  + x, y : CHARACTER;
  {
    i = low;
    j = high;
    x = tab.item ((i + j) >> 1);
    {
      (tab.item i < x).while { i = i + 1; };
      (x < tab.item j).while { j = j - 1; };
      (i <= j).if
        {
          y = tab.item i;
          tab.put (tab.item j), i;
          tab.put y, j;
          i = i + 1;
          j = j - 1;
        };
    }.while (i <= j);

    (low < j) .if { my_qsort tab from low to j;  };
    (i < high).if { my_qsort tab from i to high; };
  };
```

4 Some aspects of the semantics of Lisaac

To make short the presentation of the semantics of Lisaac, we consider in the following section that the reader is familiar with the rudimentary notions of the Self language, main source of inspiration of Lisaac. Because it is not possible to present here all details of the semantics of our language, we emphasize only the differences between Lisaac and Self. These differences are of course related to the fact that Lisaac is a language aiming at implementing Isaac, our operating system.

As in Self, a prototype can change its behavior during the execution, either by adding, redefining or suppressing slots.

4.1 Section identifiers

The identifier of a section makes it possible to choose the interpretation of the slots which are in this section. The interpretation of the slots relates to various aspects:
• heading and versioning information (cf. 4.1.1).
• the mode of application of the *lookup* mechanism: inheritance slot (cf. 4.1.2) and traditional message slot.
• the protection and the level of accessibility of the slots (cf. 4.1.3).
• the compilation mode of the code (remote code mode, close code mode or hardware interruption/exception mode 4.1.4, data structure mapping mode 4.1.5, ...).

4.1.1 The HEADER section

The HEADER section is used to enumerate the general parameters of the prototype. This section is mandatory and must include the `name` slot which indicates the name of the prototype itself. The `category` slot indicates the category of the prototype with respect to its level of protection related to the other prototypes (cf. 4.1.3). Other optional slots can be added in this section to comment on the prototype. In this section, only the slots containing constants (character string, or numerical constants) are authorized.

In addition, some conventions on the names of the slots have been fixed for the purpose of maintenance and to ensure consistency of the information of the HEADER section:

Slot name	*Type*	*Description*	*Level*
name	string	name of the prototype	mandatory
category	PRIVATE, KERNEL, DRIVER, PUBLIC	protection level	default is PUBLIC
version	integer	version number	mandatory
date	string	release date	optional
comment	string	Comment	optional
author	string	author's name	optional
bibliography	string	programmer's reference	optional
language	integer	encoding country language	optional
bug_report	string	bugs report list	optional

4.1.2 The INHERIT section

This section describes the inheritance slots of the object. As in Self, a prototype can have several parent slots (i.e. multiple inheritance). The slots of this section being mostly used by the *lookup* mechanism, only slots without arguments are authorized.

Most of the time, a slot of the INHERIT section indicates another prototype simply by using its name. It is also possible to define a parent slot using an instruction block. When this is the case, the value of the parent slot is initialized with the result value of the evaluation of this block during the first use of the prototype. The instruction block which describes the value of a parent slot is evaluated only once. In case of further cloning of this prototype, no reevaluation of the block is carried out.

As in Self, the assignment of a parent slot may occurs at any time to change dynamically the ancestors of the prototype. A parent slot with no value at a given time is simply ignored by the *lookup* algorithm. Adding a new inheritance slot during the execution is not allowed in Lisaac. Slots of the INHERIT section are not visible from outside of the object itself. Accessing a parent slot simply yields the corresponding parent object if any.

The order in which the slots are declared is obviously important for the *lookup* algorithm while seeking a message. The inheritance slots are examined with respect to the order in which the source text is written, in a deep-first way, without taking account of possible conflicts.

A message call applied to some parent slot is the natural mechanism to achieve he equivalent of *super* in Smalltalk or *resend* in Self. This means that the message is sent to the parent with the current object context (*self* on the figure 2).

4.1.3 Sections PRIVATE, KERNEL, DRIVER, PUBLIC and NETWORK

These names of section define the level of accessibility and protection of the corresponding slots. For example, the slots defined in a section PRIVATE are visible only inside this prototype. They are not accessible from another place, even by descendants (one does not inherit the section PRIVATE).

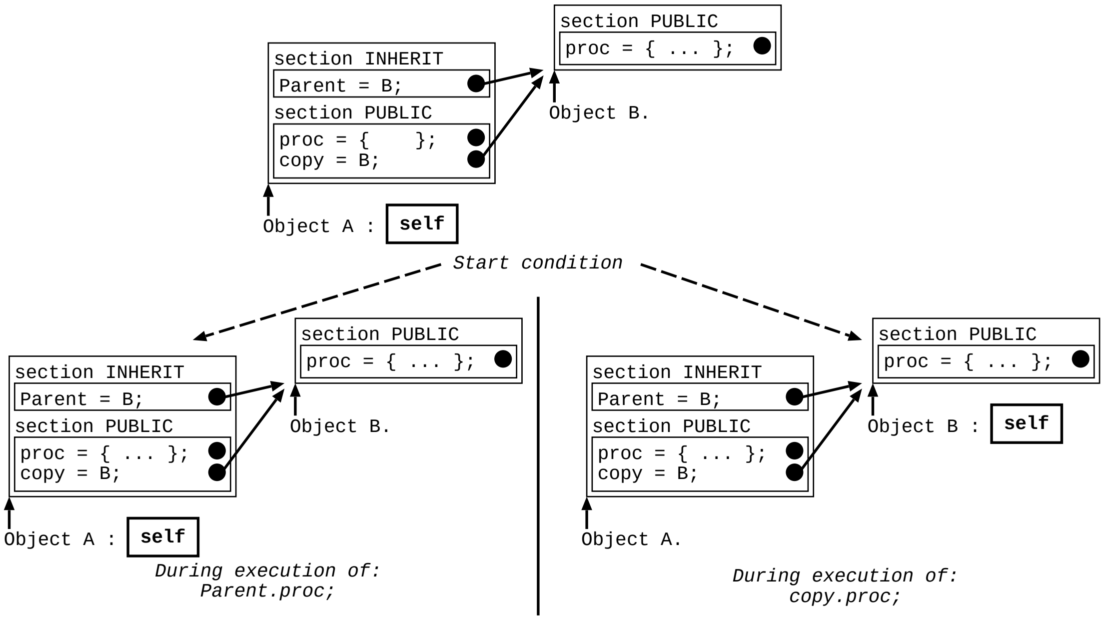

Figure 2. The equivalent of $super$ in Smalltalk or $resend$ in Self.

The other sections (KERNEL, DRIVER, NETWORK and PUBLIC) make it possible to fix the policy of interaction between prototypes knowing that: the PUBLIC category is reserved for the unreliable objects, the KERNEL category is dedicated to the critical objects of the system, the DRIVER category is reserved for *hardware drivers* objects, and finally, the NETWORK category is reserved for the objects coming from a network or being carried out on a distant machine.

Without presenting here in details all the rules of communications (fig. 3), let us quote for example that an unreliable object (category PUBLIC) cannot be used by a critical object (category KERNEL). The goal of this rule is to avoid putting in danger the integrity of the system at the time of the call of a functionality of the core.

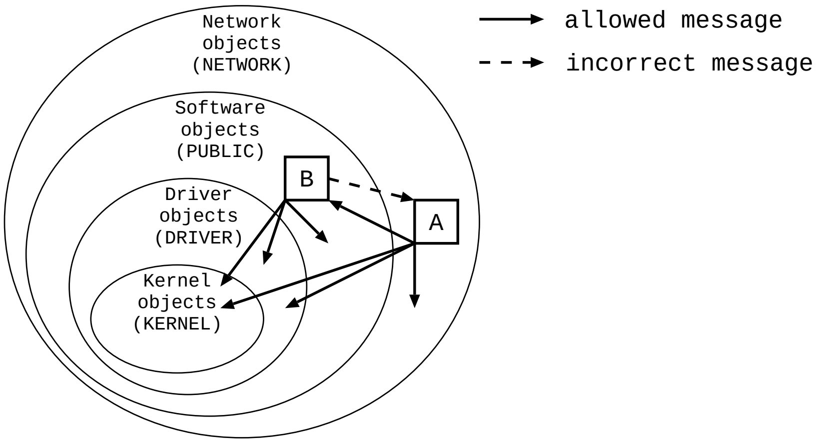

Figure 3. The four levels to protect objects communication.

Thus, there are four levels of protection checked at runtime: KERNEL, DRIVER, PUBLIC and NETWORK. Checking can be achieved by using built-in hardware protection mechanism mixed with software protection when the processor has only two hardware protection levels (intel processor have four level of protection while motorola has only two levels). An assumption of responsibility of protections by the processor and a single space of addressable memory allow a protected and powerful communication between the objects [Son01].

4.1.4 Section INTERRUPT

The goal of the INTERRUPT section is to handle hardware interruptions. In such a section code slots are allowed. Each slot is associated with one of the processor's interruptions [Hum90]. These slots differ from others in their generated code. For example, their entry and exit codes are related to the interrupt processing. Their invocations are asynchronous and borrow the quantum of the current process. Generally, these slots are little time consumers and they don't require specific process' context for their executions. It is thus necessary to be careful while programming such slots to ensure the consistency of the interrupted process.

4.1.5 Section PACKAGE

The PACKAGE section purpose is to format data slots description according to some fixed hardware data structure. The main goal of PACKAGE section is to describe in Lisaac device drivers. In such a section, the compiler follows exactly the order and the description of slots as they are written to map exactly the corresponding hardware data structure. Thus, one is able to write data slots description according to the hardware to handle. Slots inside some PACKAGE section are considered private for any other objects.

4.2 Message passing

Syntax of message calls in Lisaac strongly looks like message calls in Self. The arguments may be separated by commas or may use keywords as well (the method name is splitted into words to separate arguments). As in Self, the semicolon ';' means that the same receiver is use again for the second method invocation (i.e. sequence).

4.2.1 Binary messages

Compared to Self, we added the possibility to chose the associativity and the priority of operators as in the ELAN language [PBy00]. To select the associativity of an operator, one must use the keyword *left* or the keyword *right*. The default associativity is *left* and the default priority is 1 (the highest one).

4.2.2 Unary messages

Only the prefixed unary operators are allowed. The '$\wedge$' operator is reserved for returning a value inside a method body. As usual, the type of the expression after '$\wedge$' must be compatible with the type signature of the method.

Moreover, the unary operator '?' is used to allow a rudimentary contract-programming mechanism. It is more like the `assert` mechanism of the language C than the powerful `require`/`ensure` Eiffel mechanism. Once the object has been tested, the programmer can withdraw these assertions in the final delivery version by using a simple option of the compiler.

4.3 Blocks

Instruction blocks are defined by the '{' ... '}' notation and are BLOCK objects. In Lisaac, block objects cannot have arguments or local variables. Their evaluations are carried out in their definition environment. This decision is different from the Self language in order to facilitate their use which is similar to instruction blocks in ordinary C code.

5 The design of Isaac in Lisaac

The intent of this section is to show that a high-level prototype-based language fits very well with an operating system design and implementation. Here are some selected examples from the Isaac operating system as it is implemented in Lisaac.

5.1 Hardware components *versus* software components

In the Isaac system objects hierarchy, true physical hardware objects (e.g. keyboard, mouse, memory, ...) are distinguished from system-software objects (e.g. file, vector, bitmap, ...). In a very natural way, inheritance is used to separate hardware objects from software objects. The reason of this segregation is that hardware objects are not clonable by another PUBLIC object. Using other words: one cannot clone a screen if the physical new screen does not exists. Hardware components are obviously natural critical resources. Conversely, software components, SOFT_OBJECT (fig. 4), inherit the traditional `Clone` method with PUBLIC accessibility. Also note that the common set of named object is available to all thanks to the general OBJECT prototype.

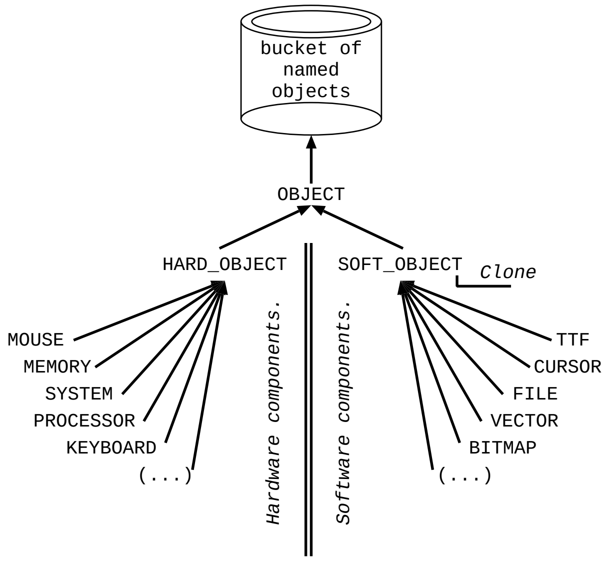

Figure 4. Hardware components and software components segregation.

5.2 Dynamic inheritance at work for video drivers

Figure 5, represents the Isaac's video architecture. The VIDEO object can change dynamically its parent slot to inherit BITMAP_15BITS or BITMAP_16BITS or BITMAP_24BITS or BITMAP_32BITS as well. Actually, dynamic inheritance is obviously used to change dynamically the bitmap resolution (one or more times). The reference to the VIDEO object remains unchanged for clients allowing the resolution mode to be changed transparently (i.e. only the parent slot of the VIDEO object is modified).

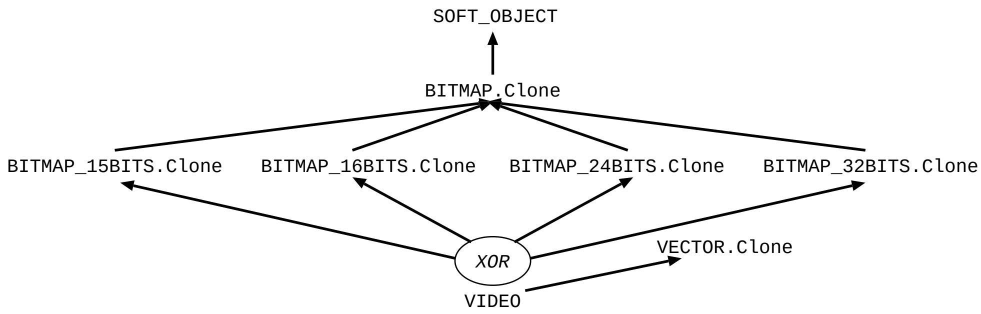

Figure 5. Dynamic inheritance to select the appropriate VIDEO DRIVER**.**

Moreover, the VIDEO object can redefine any BITMAP's functionalities in order to take as many advantages as possible from the hardware graphic device (graphics accelerator embedded on the board, bit depth, . . .).

5.3 The Isaac file system

The implementation of the Isaac file system is another example of dynamic inheritance usage (fig. 6). The abstract INODE object is inherited by FILE and DIRECTORY as well. The actual INODE object parent slot indicate the appropriate file representation: FAT_16BITS or EXT_2FS for example. Then, the file representation itself may inherit FLOPPY when this inode is on a floppy disk or IDE in the case of some hard disk. Once again, dynamic inheritance is extremely useful and flexible in this case.

5.4 The quick-sort benchmark

Our compiler, while complete, is still under development, and it is not possible to carry out many reliable tests of performances. Nevertheless, a simple test of the quick-sort algorithm is very promising since the performances are equivalent to those of a similar C program. In spite of the absence of hard-coded test/loop statements in Lisaac, the generated C code is very similar to the hand-written C code, hence the similar performances. To achieve the translation from Lisaac to C, our compiler use traditional removal of recursivity as well as inlining, code specialization and data flow analysis. Most of our compilation strategy come from our experiment with the SmallEiffel compiler [ZCC97] and from the *Cartesian Product Algorithm* of Ole Agesen for the Self language [Age95].

Figure 7 shows user times which we obtained on the same algorithm of quick-sort out of C, SmallEiffel and Lisaac. Those tests were carried out on INTEL Pentium II to 333 MHz with 128Mo of RAM memory under Linux 2.2.17 and GCC 2.95.2..

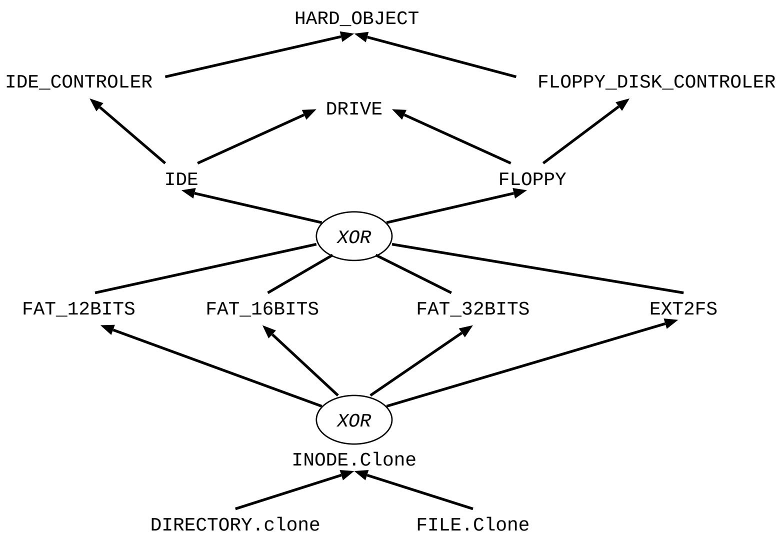

Figure 6. Another example of dynamic inheritance (file system selection).

Compiler	user time (-O0)	user time (-O3)
gcc 2.95.2	84.030 s	33.840 s
SmallEiffel -0.75	87.920 s	36.850 s
Lisaac	82.980 s	33.620 s

Figure 7. Execution time of the quick-sort benchmark.

6 Conclusion

The set up of our Isaac operating system, led us to conceive a new object-oriented language, called Lisaac (cf. 3 and 4). While remaining compact, uniform and very close to Self, another prototype-based language, our Lisaac language differs from Self by adding a reliable policy for communications and protections (cf. 4.1.3).

Lisaac is also conceived to manage programming of interrupt vectors (cf. 4.1.4) as well as other various architecture specific tables handling. Isaac is currently running on Intel processors but the design of Lisaac make it easily portable on other architectures (cf. 4.1.5).

The Lisaac language is compiled using C as an intermediate assembly language. Lisaac constitutes a powerful tool in the making of an efficient, flexible, and cleanly design operating system. As Self is, Lisaac is fully prototype-based and extremely flexible and dynamic. The very novelty of Lisaac is to take in account protections mechanism as well as operating system tools. The architecture of our object operating system takes fully advantage of the possibilities offered by prototypes and especially by dynamic inheritance. (cf. 5.2 and 5.3).

The firsts benchmarks of the generated code's execution are very encouraging (cf. 5.4) and let us think that it is quite possible to obtain Lisaac objects running as fast as C programs are.

References

[Age95] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Sciences*, pages 2–26. Springer-Verlag, 1995.

[CZ99] D. Colnet and O. Zendra. Optimizations of Eiffel programs: SmallEiffel, The GNU Eiffel Compiler. In *29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99), Nancy, France*, pages 341–350. IEEE Computer Society PR00275, June 1999. LORIA 99-R-061.

[elf95] Tool Interface Standard (TIS) Committee. *Executable and Linking Format (ELF) Specification*, 1995. v1.2.

[Hum90] R. Hummel. Interruption and exception. In *Intel486 Microprocessor Family Programmer's Reference Manual*, pages 83–104, 1990.

[Mey94] Bertrand Meyer. *Eiffel, The Language*. Prentice Hall, 1994.

[PBy00] H. Dubois ... P. Borovansk y, H. Cirstea. Library reference manual. In *ELAN*, pages 20–24, 2000.

[Pro] Intel Processor. http://www.sandpile.org/docs/intel/80386.htm.

[sc00] EGCS steering committee. `http://gcc.gnu.org`. Site web : GNU Compiler Collection., 2000.

[Son00] B. Sonntag. `http://www.isaacOS.com`. Site web: Isaac (Object Operating System)., 2000.

[Son01] B. Sonntag. Article in French about: Usage of the processor memory segmentation with a high-level language. In *2ième Conférence Française sur les systèmes d'Exploitation, (CFSE'2)*, pages 107–116. ACM Press, 2001.

[US87] 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.

[ZCC97] O. Zendra, D. Colnet, and S. Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, volume 32, number 10 of *SIGPLAN Notices*, pages 125–141. ACM Press, October 1997. LORIA 97-R-140.