QUATORZIEME COLLOQUE GRETSI - JUAN -LES-PINS - DU 13 AU 16 SEPTEMBRE 1993

A SIMPLE AND EFFICIENT OBJECT ORIENTED BLOCK PROGRAMMING (OOBP) PARADIGM FOR SIGNAL PROCESSING SOFTWARE DEVELOPMENT

Thierry DUTOIT, Vincent FONTAINE, Henri LEICH.

Service de Théorie des Circuits et de Traitement du Signal,
Faculté Polytechnique de Mons,
31 boulevard Dolez, B-7000 Mons, Belgique.

RESUME

La programmation est une composante essentielle du développement d'applications en Traitement du Signal. Après un bref rappel de quelques critères de programmation, qui permettent de souligner l'importance d'une description hiérarchique des systèmes, nous introduisons un paradigme de programmation original basé sur la Programmation Orientée Objet par Bloc. Nous en exposons les trois niveaux de description empruntés à VHDL (la spécification d'entité, l'architecture, et la configuration), et en commentons l'usage des concepts d'encapsulation, d'héritage, et de polymorphisme. Nous montrons qu'il permet de satisfaire à nos critères de programmation sans nécessiter l'intervention d'un noyau de contrôle caché. Enfin, nous en proposons des extensions, qui offrent des facilités de contrôle supplémentaires.

ABSTRACT

Digital Signal Processing scientists are increasingly involved in programming, After a brief recall of some programming quality criteria, in which the importance of a hierarchical description of systems is underlined, this paper introduces an original Object Oriented Block Programming (OOBP) paradigm for DSP software development. The use of encapsulation, inheritance, and polymorphism by its VHDL-like three descriptive levels (namely entity specifications, architectures, and configurations) is exposed. It is shown to fulfil the aforementioned criteria, without the need to incorporate complex kernels to control data flows. Extensions are finally proposed, which provide programmers with optional monitoring facilities.

1. INTRODUCTION

More than any other discipline, Signal Processing is now thoroughly involved in Computer Science. As a result, the art of programming has become an important component of our everyday life as Signal Processing scientists, for at least three reasons :

1. **Simulation and Research**, for which we often relax some feasibility constraints, such as real time or memory constraints, to focus on functionalities.

When a digital system has been successfully simulated, it can be turned into a concrete application-oriented system. This may currently be done in two ways, both of which are again highly dependant on programming strategies:

2. **Software Development**. Digital Signal Processors are extensively used in this area, whether it was through general purpose DSP boards or application specific ones. Tools for compiling structured languages are now available for most of them, with a clear dominance of C, for serial and parallel computing.

3. **Hardware Development**. ASICs recently appeared to be an interesting economical solution for implementing algorithms. Again, Hardware Development Languages (HDLs, as opposed to Software Development Languages, SDLs) have emerged, reducing most of the work into programming, and turned into a world-wide standard : VHDL ([IEEE 87], [Airiau et al 90]).

For all these reasons, it is worth considering, at least for a moment, the form of our programs rather than their content.

2. PROGRAMMING PARADIGMS

Let us first consider some constraints encountered when programming digital systems (through S or HDL), and recall the pros and cons of some programming paradigms currently available. Constraints are often be referred to as :

- *programming speed* and *error correction speed,* which mostly depend on the existence of error free libraries;

- *computation speed*, which can be increased in three ways : by the programmer himself (often at the expense of the portability and readability of his source code), by the compiler if optimization tools are available, or again through the use of pre-optimized libraries (the only solution in the case of HDLs, VHDL 's intensive use of packages being a good example).

- *readability :* the closer the code is from its functionality, the better it can be understood, and used, by others.

- last but not least : *portability*. Porting software applications on DSP platforms is mostly ensured by the use of the C language (and probably C++ in a near future). Implementing C programs into ASICs, however, is greatly facilitated if the original material has been written in a strongly structured way, by grouping logical functionalities into blocks with simple and clear I/O, so that the final code closely matches its block-diagram representation.

Structured programming is the most popular programming paradigm today in the Signal Processing society. It gave birth to Signal Processing libraries, that greatly increase the programming speed, while slightly affecting the computing speed, given the time required to load/unload stacks. Error correction itself is accelerated, since well-designed libraries are generally error-free. Data accesses are restricted to their visibility, and functionalities are hierarchically grouped, so that readability, and extendibility are also increased.

Problems, however, mainly arise when trying to design libraries for complex functionalities. Most commonly available C libraries for example, include functions that perform simple tasks (in terms of control statements only; the mathematical complexity of 'simple tasks' can be enormous !). Describing intricate processes in terms of structured programming generally requires the creation of functions with kilometric parameters lists. Even though, a slight modification of a sub-process would most often force the programmer to rewrite a complete version of the process.

Other problems are encountered when functionalities have a memorizing comportment. We all have produced code describing some filter as a function of its input, output, coefficients and internal variables, even if the last ones were left untouched at run time (after the filter initialization). The bigger the process, the more internal variables it will include, most of them generally being perfectly useless outside the process itself... Data definition and accesses are then clearly performed out of their logical visibility.

Object Oriented Programming (OOP) languages, under the banner of C++, are a very flexible solution to these impediments. Encapsulation allows memorization, inheritance avoids rewriting, and polymorphism is the key to building libraries of complex systems. The point is that they are not yet supported by DSP compilers, mostly because they have only been partially adopted by our community. In the particular case of C++ however, commercially available pre-processors allow to transform any Object-Oriented code into standard C. Portability to DSP platforms is thus possible, but errors are difficult to correct, since the resulting C code is hardly readable. Optimized OOP compilers would help.

On the opposite of a structured programming fanatic, the adept of OOP spends most of its programming time thinking of the best way to describe the process he is interested in, the best inheritance scheme to adopt, so that the code he will have to write will be maximally "logical". The point is that there are many "logics" that would be worth to follow. Should objects be defined around data, or around functions ? Which methods to define ? When to use inheritance ? When applied to Digital Signal Processing in general, this mental task may last months. Such intensive thinking gave birth to some interesting frameworks, that will now be quickly recalled.

The concept of Object Oriented Signal (i.e. OOP defined around data) has been imagined and developed since the eighties. Systems like SPLICE [Myers 86] and QuickSig [Karjalainen et al. 88] are good examples. A comprehensive approach of this paradigm is given in [Karjalainen 90]. QuickSig itself, which is based on an Object Oriented version of the LISP language, was recently adapted so that LISP atoms could automatically be turned into compiled code for the TMS320C30 DSP.

Block Processing itself (i.e. OOP defined around functions) is an old concept ([Goldberg & Rader 69], [Covington et al. 87], [Zissman & O'Leary 87]) that naturally accounts for the modularity of signal processing systems. It tends to define any operation through (and only through) a block or data-flow diagram, and is now used extensively by commercially available DSP graphical compilers, such as Comdisco 's Signal Processing Worksystem, HP 's Visual Engineering Environment [Beethe 92], or National Instrument 's LabView, often presented as "alternatives to cumbersome text-based programming". These are very interesting tools for learning, and for producing some specific DSP software. Researchers, however, are often reluctant to use them for intensive programming, because they loose some control on the actual execution of their code, where important "hidden" kernels play a non-trivial role.

Being faced ourselves with the practical need to adopt an OOP approach for Digital Signal Processing, and aware of the high acceptance of C in the Signal Processing area, of the high degree of standardization of C++, and of the possibility to port it to DSP platforms, we decided to develop our own OOP paradigm for DSP software development. The criteria we adopted were the previously introduced constraints, plus one important aspect : ideas should be kept simple. No complex kernel should have to be available to run our objects, neither should it require intricate formulations. What 's more, any intermediate approach between structured programming and ours should still be possible, so that OOBP may be progressively used in a progressive way. To achieve this, as clearly appears in the previous discussion, a Functionality Oriented design was more than welcome. This resulted in an Object Oriented Block Programming (OOBP) approach that will be detailed in the next paragraphs. For practical reasons, examples will be given in C++, but the ideas are language-independent.

3. OBJECT ORIENTED BLOCK PROGRAMMING.

Most of our work resides in a multilevel description of processes. Starting from VHDL ideas, we exploit hierarchical recursive structuration by describing any DSP task as block object, which may itself include other blocks defined as sub-blocks for the current task (it should be noted that inclusion, in this context, means that any block owns its sub-blocks; it has nothing to do with inheritance). Sub-blocks are themselves DSP tasks (i.e. blocks), and so on. We define the *order* of a block as the order of its highest-order sub-block plus 1, or 0 for final (or *base)* blocks, which have no sub-block. The highest order block is called the *main block*. Running the program is equivalent to launching its process.

Blocks exchange information through external data structures, for which OOP can also be used, but it is not essential. These have their own existence in memory, independently of the blocks that address them. They are generally dynamically created (and disposed) by higher order blocks, in which they are considered as *internal variables*.

A second and more abstract classification is performed, independently of the notion of order. As in VHDL, any task may be considered at three levels, namely the *entity specification* level, the *architecture* level, and the *configuration* level. They correspond to different levels of knowledge on the actual task of a block. Configurations inherit from architectures, which themselves inherit from entity specifications. This completely captures our use of the **inheritance** concept.

3.1. Entity Specifications.

The entity specification level describes a task as a black box. It restricts its knowledge of a block to its functionality, i.e. the type of its inputs and outputs (though virtual types are admitted), and the symbolic operation (as opposed to the precise algorithm) by which they are related to one another through the process. The basic idea is that two descendants of a common entity specification can always be exchanged without affecting the global functionality of the higher order blocks that include them : **polymorphism** ensures compatibility for the outer world.

A typical constructor has the form (written with LL1-like abstractions for convenience) :

```
SomeEntity::SomeEntity( {Parameters} )
```

in which, in order to allow genericity, some parameters may be passed, provided they are strictly related to the block 's functionality. For reasons that will be clarified later, entity constructors do not define their actual I/O : these are not passed to the constructor, but rather defined through a specialized method :

```
SomeEntity::IO_Def( {Inputs},{Outputs} )
```

where "Inputs" and "Outputs" stand for references to external data structure(s) used as input and output buffer(s). All entities inherit from a common ancestor, the *block* object, which only declares a virtual 'Process()' method :

```
class Block :
        { virtual void Process() {} ; };
```

'Process()' does nothing by default, but will be overridden by the corresponding method of the architecture level, to implement the actual algorithm of the block. Overriding cannot be performed at the entity level, since entities have no information on the algorithm used to complete their task. Similarly, no explicit destructor is needed, since entities don't own their I/O. Finally, any entity specification object has the form :

```
class SomeEntity : public Block
        {[ parameters values]
         [ pointers to the I/O data structures ]
         SomeEntity(...) ; // constructor
         void IO_Def(...) ; // I/O definition method
        };
```

3.2. Architectures

Architectures go one level deeper. An architecture knows the functional block-diagram chosen to perform the current task. It includes references to all its sub-blocks and internal variables. Its main task is contained in its (overridden) Process() method, which may itself call the sub-blocks 'Process()' methods.

Sub-blocks are **not** created by the architecture. They are dynamically constructed outside of it, and passed to architecture 's constructor. The key point is that instances of configuration level blocks (see below) with undefined I/O are passed, though they are seen as entity specifications by the architecture. This is a necessity for the aforementioned interchangability : architectures access their sub-blocs as black boxes. It also motivates, a posteriori, the existence of a virtual 'Process()' method in the Block object.

Architectures are totally responsible for their internal and transfer variables. These are created by the architectures during their construction, and disposed during their destruction, so that **encapsulation** is observed. Consequently, sub-blocks I/O can only be defined after transfer variables have been created, i.e. after sub-blocks have been constructed. That is why the IO_Def method was designed at the entity level : it is called by architectures to fix their transfer variables as I/O for their sub-blocks.

The general structure of an architecture is thus given by :

```
SomeArchitecture : public SomeEntity
        {[ pointers to (entity specification) sub-blocks]
         [ pointers to internal and transfer variables ]
         SomeArchitecture( [{SubBlocks},] [{Parameters}] );
         ~SomeArchitecture(); //deletes int. var. +sub-blocks
         virtual void Process();// algorithm
        };
```

3.3 Configurations.

In order to run a block, its is necessary to construct an instance of one of its existing architectures. This requires the prior construction of all its sub-blocks, since architectures never construct them. Constructing non-base sub-blocks may itself involve the creation of lower level sub-blocks, and so on. To avoid this tedious cascaded construction chain, pre-defined configurations (i.e. configured architectures, which completely own their sub-blocks and fix up virtual types) should always be associated with architectures. These may then be used as base sub-blocks when configuring higher order blocks. Configuration object have a very simple form :

```
SomeConfiguration : public SomeArchitecture
        {SomeConfiguration ( { Parameters } ); };
```

where {Parameters} is generally a copy of the architecture 's parameters list. No explicit destructor is needed : the inherited architecture 's one makes all the job.

3.4. Comments

The OOBP paradigm that was exposed in the previous paragraphs fully satisfies our initial criteria :

• As such, it allows the conception of block libraries that have a much broader application range than with structured programming and the same extendibility as in VHDL.

• Readability is also ensured since, as in VHDL, our OOBP formalism is based on a clear description of processes in the form of configurable architectures.

• Computation speed is better than with structured programming. The absence of parameters in the 'Process()' method (no more stack loading/unloading) more than compensates for its virtuality (functions calls are indirect).

• Portability needs a special attention. When developed in C++ (as we did), our paradigm benefits from the best available SDL portability, for two reasons : 1. As mentioned earlier, passages exist to DSP implementations. It is clear that possible further developments of OOP compilers for DSP will be for C++. 2. Our three level approach constitutes the first step of a descriptive programming of system, as opposed to a procedural one. As such, most of the mental work that is done when systems are structured into hierarchically related blocks is language-independent. For the same reason, portability to HDLs is highly increased : our structured representation of processes is precisely the first step to their VHDL description ! The main difference between the OOBP paradigm proposed and VHDL descriptions actually is that our architectures explicitly control the way their sub-blocks are processed, though this is done automatically by VHDL simulators, by taking a virtual 'time' parameter into account. This difference is clearly the result of a choice : explicit control is an essential information to ensure a high computing speed on sequential machines.

• Finally, simplicity is respected. The ideas and the way they are expressed in the code are almost straightforward. No kernel is needed to run and handle blocks : just instanciate a configuration, supply it with external inputs and outputs, and run its 'Process()'.

As such, however, we have no yet fully exploited the power of our objects... Some optional but very helpful extensions are welcome.

POSSIBLE EXTENSIONS.

Highly interesting features may be added to our blocks, by simply extending the competence of the Block object. Some more general-purpose methods can allow an external kernel to provide *additional* and *optional* control on any block, while insignificantly decreasing the computing speed, and slightly increasing the code size.

A run-time error handling method, which reports run-time-errors and associates them with the block in which they occur, is a good example. It can be economically implemented if each block stores its name in one of its fields.

One can then use the Block object to provide each block with viewing facilities for its inputs, outputs, transfer and internal variables. This is performed simply, by introducing some slight modifications in our basic organization :

• Calls to the 'Process()' method of sub-blocks should be changed to calls to an intermediate 'Compute()' method, which is defined in the Block object, and has the same external behaviour as 'Process()', as far as computation is concerned. It effectively runs the 'Process()' method, and then checks if viewers have been opened for the current block, in which case it refreshes them.

• In order for the 'Compute()' method to detect viewers, a dynamic array of pointers to viewers is declared in the Block object, the content of which is automatically actualized whenever an architecture activates or closes one of its viewers. Viewers activation itself may be virtually declared in the Block object, and practically overridden in architectures.

In a nutshell, the Block object can be defined so as to centralise all the information that is necessary to monitor the execution of its task, with no real interference with the task itself. Such facilities are *optional* in the sense that it is possible to define several compatible implementations of the Block object, so that the same OOBP source code can be compiled with or without them.

The OOBP Signal Processing library we designed at the FPMs TCTS labs currently includes a minimal (DOS compatible) version of the Block object, in which the 'Compute()' method is directly (inline) mapped to the 'Process()' one, and a Windows version with full facilities. Programmers may use both indifferently, with NO modification in their OOBP source code.

BIBLIOGRAPHY

[Airiau et al. 90] R. Airiau, J.-M. Bergé, V. Olive, J. Rouillard, *VHDL, du langage à la modélisation*, Presses polytechniques et universitaires romandes, Lausanne, 1990.

[Covington et al 87] C.D. Covington, G.E. Carter, D.W. Summers, "Graphic Oriented Signal Processing Language, GOSPL", *ICASSP 87*, Dallas.

[Gold & Rader 69] A. Goldberg & C. Rader, *Digital Processing of Signals*, McGrawHill, New York, 1969.

[IEEE 87] *IEEE Standard VHDL Language Reference Manual*, IEE standard 1076-1987,

[Karjalainen et al. 88] M. Karjalainen, T. Altosaar, and P. Alku, "QuickSig : An Object-Oriented Signal Processing Environnement", *ICASSP 88*, New York.

[Karjalainen 90] M. Karjalainen, "DSP Software Integration by Object Oriented Programming : a Case Study of QuickSig", *IEEE ASSP Magazine*, vol. 7, N°2, pp. 21-31, April 90.

[Myers 86] C. Myers, *Signal Representation for Symbolic and Numerical Processing*, Ph. D. Thesis, MIT Technical Report 521, august 1986.

[ZissMan & O'Leary 87] M.A. Zissman, G.C. O'Leary, "A Block Diagram Compiler for Digital Signal Processing MIMD Computers", *ICASSP 87*, Dallas.