PLUG AND PLAY SOFTWARE FOR DESIGNING HIGH-LEVEL SPEECH PROCESSING SYSTEMS

T. Dutoit(,+) , J. Schroeter(+)*

(*)TCTS Lab,
Faculté Polytechnique de Mons
31 Bvd DOLEZ, B-7000 Mons, Belgium

(+)AT&T Labs-Research
180 Park AV, PO Box 971
Florham Park, NJ 07932-0971

ABSTRACT

Software engineering for research and development in the area of signal processing is by no means unimportant. For speech processing, in particular, it should be a priority: given the intrinsic complexity of text-to-speech or recognition systems, there is little hope to do state-of-the-art research without solid and extensible code. This paper describes a simple and efficient methodology for the design of maximally reusable and extensible software components for speech and signal processing. The resulting programming paradigm allows software components to be advantageously combined with each other in a way that recalls the concept of hardware plug-and-play, without the need for incorporating complex schedulers to control data flows. It has been successfully used for the design of a software library for high-level speech processing systems at AT&T Labs, as well as for several other large-scale software projects.

1. INTRODUCTION

Let us face it: speech technology is progressively coming to the point where the software implementation of a new idea becomes at least as valuable as the patent on the idea itself[1].

Linear prediction, which was invented some thirty years ago, can be implemented with only a few tens of lines of code. In contrast, the software implementation of the multiband excited (MBE) model requires a lot of care, and several thousands of lines of code, although the underlying analysis algorithm (which is already ten years old [1]) can be summarized in a few pages of text and a dozen of equations. Yet these examples are only related to acoustic analysis of speech. Higher-level speech processing systems (whether for speech coding, synthesis, or recognition) will increasingly make use of phonetic, lexical, syntactic, and semantic information, which will still make them more intricate to program. Hence these two statements of faith:

1. Future milestones in speech processing will come from labs with strong commitment to solid, portable, and extensible code;
2. Speech scientists and software engineers will soon be the same people.

We start in section 2 with a short introduction to the use of block-diagram abstractions (and the associated *dataflow* semantics) for DSP software development and highlight the importance of *declarative* (as opposed to *procedural*) programming. This section contains definitions that are used in sections 3 and 4. In section 3, we introduce a simple and efficient way of using object oriented languages as an extension to function-based programming for programming high-level signal processing systems. This programming paradigm, called Object Oriented Block Programming (OOBP), mimics the inclusion and abstraction properties of block-diagrams by allowing processes to be included into higher-level processes and by describing each process at three different levels of abstraction. The use of encapsulation, inheritance, and polymorphism by its VHDL-like three descriptive levels (namely *entities*, *architectures*, and *configurations*) completely captures the spirit of object oriented programming. Applying the same goal to the description and implementation of data objects leads us to define, in Section 4, a stream-based class hierarchy for use as IOs by the above-mentioned OOBP objects. We conclude in section 5 by commenting on the enhanced reusability and extensibility properties of applications developed using such “plug-and-play (PnP)” software components.

2. DECLARATIVE DATAFLOWS

Although procedural languages can be used to mimic block-diagram descriptions, they explicitly overspecify processes by imposing not only *which* functions to call, but also *when*. Such overspecification can be avoided with specially designed *declarative* languages, which more closely mimic block diagram descriptions, by implementing a process as a list of interconnected sub-processes. Programmers no longer have to specify how processes get fired. In counterpart, they loose some control on their code which is left to compilers, when firing decisions can be made *statically* (i.e., once and for all before the program is run), or by *dynamic* schedulers, if not. Such declarative languages are now extensively used by commercially available DSP graphical programming environments, such as Comdisco's Signal Processing Worksystem, HP's Visual Engineering Environment [2], National Instrument 's LabView [3], or the Mathworks SimuLink, often presented as "alternatives to cumbersome text-based programming". The associated semantics is known as the *dataflow* semantics. Blocks in a dataflow diagram exchange information through data structures. Exchanges are directional (hence, one block is the *sender*, and the other is the *receiver*. Block-programming systems mainly differ by the way they organize block firing (data-driven vs. time-driven, static vs. dynamic scheduling,

[1] The other really valuable thing these days being a large labeled database to put the idea into practice.

centralized vs. distributed control), and data control (direct vs. buffered access).

2.1. Time-driven vs. data-driven

Block processing applications can be classified as *data-driven* if their scheduler is based on the existence of a *time* variable. Typical examples of time-driven environments are VHDL and Comdisco's Signal Processing Worksystem (SPW). In contrast, data-driven systems assume that blocks can be fired as soon as they have enough data at their inputs, without the need for a clock control. Such a strategy has been implemented in environments like Ptolemy [5], the object oriented Visual Engineering Environment of HP, MathWorks SimuLink, or IRCAM's MAX [6].

2.2. Static vs. dynamic scheduling

Since block processing languages generally do not explicitly specify the order in which blocks have to be fired, some *scheduling* is needed. It is shown in [4] and [7] that synchronous data flows (SDF)[2] can be scheduled *statically*, that is, at compile time. The compiler is then responsible for finding a sequence of process calls which loops endlessly to execute the application. Such a strategy has been put to use in many block programming environments based on a graphical description (synchronism is almost always implicitly assumed in this case [8]). Things get more complex when systems include asynchronous data flows (ADF). In this case, a block can only be fired when there is enough data at its input, *and* when there is enough room in its output to store the output data. As a matter of fact, since the number of data items in an asynchronous data flow cannot be known at compile time, it must be controlled at run time before running either the receiver or the sender. This is known as *dynamic* scheduling.

The advantage of static scheduling is that it requires no processing overhead. It is thus a must when describing low-level blocks, the processing of which is rather elementary. One would not imagine, for example, that each and every adder, multiplier, and delay of a digital filter would check its input before processing. In contrast, organizing high-level blocks in a dynamic way is almost transparent in a computation speed perspective, since the overhead related to data or space availability checking is minor in comparison with the complexity of the processes themselves.[3]

2.3. Centralized vs. distributed control

Clearly, with asynchronous data flows, ultimate decisions are left for run-time. Up to now, we have suggested that they were made by the scheduler itself by checking some conditions on block inputs. This is known as *centralized control*. One can also imagine to create a *static* sequence of *possible* firings for a given block-diagram, and still require some firing conditions (on its input and output dataflows) to be checked before actually firing a block. In this case there is no need for a centralized software component to organize firings: firing conditions can be checked by blocks themselves. This is known as *distributed control.*

2.4. Direct vs. buffered access to data

Access to dataflows can be seen as *direct memory access*, i.e. dataflows are supposed to be implemented as simple memory locations, loaded or unloaded by blocks. The exact memory locations where blocks have to read/write data has to be taken care of by the scheduler, which therefore has to know about the implementation of dataflows: scheduling needs to be centralized.

Dataflows can be given some responsibility as well, such as that of organizing FIFO *buffers*, which can be read from and written to by blocks without imposing neither a scheduler nor blocks themselves to know about the actual implementation of dataflows. This buffering strategy is needed for dynamic scheduling based on distributed control. This is the one we use in PnP software.

3. OOBP

The foundations of Object Oriented Block Programming (OOBP) can be found in [9] (see also the related web site http://tcts.fpms.ac.be/oobp/oobp.html). The OOBP paradigm has been used in 1994 for a first version of the C++ library designed at Faculté Polytechnique de Mons. It has served as a basis for the implementation of the signal processing software library developed in the context of the ESPRIT HIMARNNET project. It has recently been used for the development of a speech recognition software toolkit combining HMMs and Neural Networks: STRUT (http://tcts.fpms.ac.be/speech/strut.html). It has also been put to use for the development of a software library for high-level speech components at AT&T Labs – Research. The next paragraphs give a summary of it.

OOBP is a methodology for defining *processes* as software classes. It is easy to use: combining OOBP objects into a program is like combining LEGOs into a building (OOBP objects have all the same kind of interface). It is not the ultimate solution to programming problems (some arbitrary choices still have to be made), but at least it allows the construction of large software processes as components of a library. What is more important, OOBP is by no means exclusive: every combination of structured programming and OOBP is possible, from functions-only programming to objects-only programming; most of the time, people mix OOBP with function-oriented programming.

OOBP is *modular* and *multilevel*. Modularity is achieved by organizing speech processing systems (blocks) into objects (sub-blocks) related to sub-tasks; it is multilevel in that each sub-task can be seen as either:

1. An abstract black box, which only knows about the type of its input(s) and output(s) (although even inputs and output types can

[2] Two blocks are said to be *synchronous* when they fire according to a fixed firing sequence. The are *asynchronous* otherwise. For a good introduction to dataflow semantics, see [4].

[3] Interestingly enough, most of the processor industry is progressively adopting dataflows with dynamic scheduling. The Intel Pentium Pro, for instance, handles internally about 20 data nodes at a time, which makes it run about 30 percent faster than the Pentium. The recent HP 28000 chip handles 56 nodes [10].

also imply some abstractions; see section 4) and by its abstract process (the abstract operation performed on the inputs to compute the outputs). This is what people familiar with VHDL[4] call an ***entity***. Entities are implemented in C++[5] as pure virtual objects that declare two methods:

```
    virtual void ioDefine(a_list_of_typed_IOs)=0;
    virtual int process()=0;
```

2. An ***architecture*** (as it is also called in VHDL) of such an abstract black box which additionally defines the process from the inside. Implementations, however, typically only define sub-blocks as entities (i.e., leave the implementation of sub-blocks open). *Internal data* are generally defined at the architecture level. Architectures define the two methods declared by entities.

3. A ***configuration*** (cf. VHDL again), which additionally specifies the implementation of all sub-blocks down to elementary blocks (with no sub-blocks). Configurations are the only components that can be run. They build configured sub-blocks and pass them to the underlying architecture.

Last, but not least, OOBP implements ***data-driven, dynamic scheduling***, with ***distributed control***. Blocks are connected together via *transfer data* objects (the interface and implementation of such data objects is discussed in section 4). The order in which blocks are given permission to fire is fixed at compile time, but blocks only fire at run-time if data is available at their input and when memory space is available at their output (hence the *dynamic* feature of OOBP scheduling). Ultimate decisions are being made by blocks themselves (hence the *distributed* control).

OOBP uses object oriented programming as follows:

• Configurations inherit (in the OOP sense) from architectures, which themselves inherit from entities. This completely captures the OOBP use of the OOP ***inheritance*** concept.

• Architectures are totally responsible for their *internal* and *transfer* data. These are created by architectures during their construction, and disposed of during their destruction, so that ***encapsulation*** is used to prevent internal variables from being visible from outside the block they are part of.

• Last, but not least, ***polymorphism*** is accounted for by the fact that sub-blocks of a block are defined as entities in architectures of that block while an actual configured sub-block is passed to the architecture to create a configuration of that same block. The basic idea is that two descendants of the same entity can always be exchanged without affecting the global functionality of the higher order blocks that include them: polymorphism ensures maximum re-usability and extendibility of software components.

[4] VHDL is a hardware programming language (for the design of ASICs mainly) that borrowed a lot of its syntax to ADA. The fact the OOBP has a lot in common with VHDL is an additional proof of its usefulness for designing large-scale systems while minimizing the differences between research and development code.

[5] Notice the OOBP paradigm can also be used with Java or even Matlab v.5.0. All it requires is an object-oriented language.

4. STREAMS AS IO

In order for OOBP to be able to handle asynchronous processes (the most general case), we have recently developed a data class hierarchy which implements ***buffered access to data***. Since we wanted to minimize overspecification of our OOBP blocks (which frequently happens when blocks embody too much knowledge of their IOs), we defined minimal interfaces for our data objects. Thus, similarly to what we do for block objects, we thus use interface (i.e., purely virtual) objects as ancestors of our class hierarchy, and all kinds of implementation objects to practically (as opposed to *virtually*) handle the data.

4.1. Minimal block-data semantics

Assuming asynchronous mode as a standard, all blocks in an OOBP library basically have to:

- check for the availability of data at all their inputs,
- check for the availability of memory space at their outputs,
- if the previous two conditions are fulfilled, read the input data,
- after processing the input data, send the output data (if any) to their output(s).

This typically implies that blocks know the type of data they handle, but not how it is stored in, and retrieved from, their inputs and outputs. In addition, if one assumes that blocks with memory (i.e., blocks that either need to memorize their past or store future inputs to process correctly) are responsible for their internal memory, then there is no need for blocks to access their inputs or outputs randomly; sequential access is sufficient. Summarizing, data objects need to be checked for reading and writing, read sequentially, written to sequentially. Hence, they can be viewed as *typed streams*.

4.2. The stream hierarchy

Two generic and pure virtual interface objects are defined:

- `InputStream<T>`, which only declares reading of data items of type T:

```
virtual short readable()=0;
virtual void read(T* buffer, long count)=0;
virtual void maxReadable(long maxReadable) = 0;
```

- `OutputStream<T>`, which only declares writing:

```
virtual long writable() = 0;
virtual void write(T* buffer, long count)=0;
virtual void maxWritable(long maxWritable) = 0;
```

`readable()` and `writable()` are the methods by which blocks query dataflows at run time to check the availability of data at their input(s) and the availability of memory space at their output(s), before entering their `process()` methods and using `read()` and `write()` on their IOs.

`maxReadable()` and `maxWritable()` are run by architectures at construction time, to be sure that no blocking situation will arise. This is typically the case when a receiver block cannot run its `process()` method because it has not enough data at one of its input, but the sender cannot run its `process()` method itself, because there is not enough memory available in the buffer used by the dataflow). In order to avoid such conflicts, architectures ask their sub-blocks about the maximum amount of data items they will require to read at once at run time, and about the maximum number of data items they will want to write at once at run time. Once these bounds are known for each dataflow, the architecture can impose their size.[6]

All kinds of implementations can then be defined for these two objects. As opposed to interface objects, however, which only need to be either read from or written to, implementations of these interfaces typically need to be read from by a block *and* written to by another. Hence, they inherit from both `InputStream<T>` and `OutputStream<T>`. The main three such objects are:

- `DataQueue<T>`, which defines an `InputStream<T>` and an `OutputStream<T>` that stores data items in a queue located in memory.
- `DataFile<T>`, which does the same for data items in a file on disk.
- `DataSocket<T>`, which does the same for data items sent to and received from a socket.

These classes can then be further refined so as to provide implementations of standard data formats, such as all kinds of audio files for instance.

It is important to understand that OOBP entities declare their IOs as InputStreams and OutputStreams, so that architectures and configurations see them as such, too. In contrast, the actual data objects passed to the IODefine method of blocks can only be implementations of InputStreams and OutputStreams. Hence, the knowledge an architecture has of its IOs is really minimized: blocks know how to ask for data and pass data to their IOs but not how IOs actually handle the data. This is central for the design of large software libraries.

The resulting programming paradigm, which combines OOBP with a class hierarchy of typed streams allows software components to be advantageously combined with each other in a way that recalls the concept of hardware plug-and-play, without the need to incorporate complex schedulers to control data flows. We therefore call the programming paradigm “Plug and Play (PnP) software”.

5. REUSABILITY AND EXTENSIBILITY

PnP software helps organizing processing into clearly defined subtasks, programmed as blocks. Blocks are seen as either *entities* (purely virtual objects; the only item they define is the type of IOs), *architectures* (first-level block diagram; they see their sub-blocks as entities they have, and know how to run them), or *configurations* (architectures with configured sub-blocks; the only blocks that can really be run).

Classical high-level systems for speech processing can be programmed once for all, as architectures that know what to ask from their sub-blocks, and when, but not how, sub-blocks actually will do it. This is resolved at run-time, by use of pointers to virtual functions.

Designing a block as a PnP software component does not imply using PnP software for its sub-blocks. In practice, the `process()` method can be everything from a list of `process()` calls for sub-blocks to plain C or C++ completely describing the process. Reciprocally, the PnP software implementation of blocks is by no means a constraint on their use. It can be mixed with C in every possible way. This is precisely why PnP software is just a *paradigm*, not a complex *environment*.

Another advantage of PnP software is that it can easily be extended to provide programmers with very efficient viewing and debugging tools. If some implementations of `InputStream<T>` and `OutputStream<T>` are added a visualization facility, for instance, it is very easy to enable *all* high-level objects to display their content when they are processed (without the need to even recompile blocks).

REFERENCES

1. Griffin, D.W., (1987), *Multi-Band Excitation Vocoder*, Ph.D. dissertation, MIT, Cambridge.
2. Beethe, D. C., HP VEE : “A Dataflow Architecture”, *Hewlett-Packard Journal*, (october 1992), pp. 84-88.
3. Vose, G.M., Williams, G., “Labview Laboratory Virtual Instrument Engineering Workbench”, *Byte* (1986).
4. Lee, E.A., Messerschmitt, D.G., “Synchronous Data Flow”, *Proc. of the IEEE* (1987), **75**, n° 9, pp. 1235-1245.
5. Buck, J., Ha, S., Lee, E.A., Messerschmitt D.G., “Multirate Signal Processing in Ptolemy”, *Proc. ICASSP 91* (1991), pp. 1245-1258.
6. Puckette, M., “Combining event and signal processing in the MAX graphical programming environment”, *Computer Music Journal* (1991), **15**, n° 3.
7. Lee E.A., Messerschmitt D.G., “Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing”, *IEEE Transactions on Computers* (1987), **C-36**, n° 1, pp. 24-35.
8. Knoll, A., Nieberle, R., “CADiSP - A graphical compiler for the programming of DSP in a completely symbolic way”, *Proc. ICASSP 90* (1990), pp. 1077-1080.
9. Dutoit, T., and V. Fontaine, "The Object Oriented Block Programming (OOBP) paradigm: a VHDL-like object oriented approach toward developing efficient DSP software libraries", *Annals of Telecommunications* (France), March-April 1995, pp. 365-378.
10. Scanlan, J., “Off the clock: Letting dataflow drive microprocessors”, *Wired magazine*, August 1997, p. 74.

[6] The size of dataflow buffers *cannot* be allowed to grow indefintely at run-time, since dataflows in PnP software are assumed to be blocking.