SPEECH PROCESSING IN THE OBJECT-ORIENTED DSP ENVIRONMENT QuickSig

Matti Karjalainen, Toomas Altosaar, Priiavo Alku, Lauri Lehtinen and Seppo Heile
Helsinki University of Technology
Acoustics Laboratory
Otakaari 5 A, SF-02150 Espoo, Finland

ABSTRACT

Several new software techniques are available that can be used to enhance the productivity and flexibility of speech signal processing especially in research and exploratory programming of new algorithms. This paper describes an object-oriented signal processing environment QuickSig and how it is applied in various speech processing tasks. The notion of signals as objects and operations as generic functions is presented. Other features to be described are event-based analysis of signals, representation of symbolic structures, graphic programming and block diagram description of algorithms as well as code generation for DSP processors from high-level specifications. Examples are given to characterize the use of the system in speech synthesis, analysis and recognition.

INTRODUCTION

Programming and software tools are an increasingly important part of speech signal processing development. Fortran and C have been the main high-level languages for this purpose and assemblers have been used for signal processor programming. Support by program libraries improves productivity but the challenge for a multiple-level, integrated software environment for speech processing remains. Several new software techniques have emerged that can be used to enhance the productivity and flexibility of speech signal processing especially in research and exploratory programming of new algorithms.

Our approach to better tools is based on object-oriented programming and Common Lisp as the system implementation language. QuickSig [1] is a new experimental signal processing environment developed at the Helsinki University of Technology to explore the possibilities of this new methodology and to build a powerful tool for DSP and speech processing applications. It is based on the latest advances in object-oriented programming (New Flavors and Genera on the Symbolics Lisp machines). The design philosophy of QuickSig has been to extend the Lisp language first by a layer of general DSP constructs (data types and functions) and then by more specific constructs for speech processing. In this sense it differs favourably from traditional DSP software packages that are more like programs or routing libraries. From the user's point of view the system can be considered as an expert workstation for DSP and speech processing development.

Traditional DSP is based almost entirely on numeric computations using simple data structures like scalar numbers, arrays and specific file formats for signals, spectra, etc. This formalism does not easily exhibit the clarity of abstract concepts inherent in signal processing. The higher abstraction levels and symbolic manipulations of signals remain in the mental processes of the programmer, not being an integral part of the traditional programs and the programming environments. The Common Lisp language [2], as the main language in artificial intelligence and one of the most general purpose languages available, is found to be very useful in adding symbolic computation features to DSP programming.

The object-oriented programming environment is a good platform for experimenting with several new paradigms and programming features that are not found in traditional software tools. We have developed extensions to the QuickSig kernel that support e.g. symbolic and structural representation of signals by event structures [3], graphic programming and block diagram description of algorithms [4], and code generation for DSP processors from high-level specifications [5]. These will be described briefly and examples are given to characterize the use of the system in speech synthesis, analysis and recognition.

OBJECT-ORIENTED AND KNOWLEDGE-BASED SIGNAL PROCESSING

Object-oriented programming is one of the main new trends in modern software development. It allows for better integration and reusability of complex software and in many application areas it better exhibits a conceptually clear picture of the inherent problem structure. Smalltalk is the best known "puristic" object language that is used also for experimental DSP systems. We have used the New Flavors [6] (developed for the Symbolics Lisp machines) that is close to the emerging object standard CLOS [7]. Object-oriented programming is becoming available and popular also in the realm of such languages as C and Pascal.

The main features of typical object languages are the following. Object classes are defined as a model form for concrete object instances. The classes (e.g. Flavors) define variables internal to the objects. The instances keep an internal state by the values of these instance variables (= state variables, slots). Instances of any object class can be created and deleted at runtime. Computation is localized into the objects by defining method functions specific and common to the instance objects of the class. These method functions also form a communication interface that hides the implementation details.

Hierarchical inheritance of properties (instance variables, method functions) between classes (superclass vs. subclass) makes the object-oriented programs systematic and compact. In the most advanced systems an object class can inherit properties from multiple superclasses. The outside view of comparison is based on calling the method functions of instance objects. This can be realized in two forms: (a) message passing by explicitly "sending messages" to be captured by objects and executed by method functions, or (b) by generic functions (globally named functions) that call the local method functions with the same name according to the class type of the first argument. The syntax related to the generic functions has good compatibility with the Lisp language.

There have been several attempts to implement signal processing environments based on object-oriented programming. G. Kopeck [8] formulated the concept of signals as objects. Later
Kopec has introduced ISP (Integrated Signal Processing System) [9], SRL (Signal Representation Language) [10] and SDB (Signal Data Base) especially for speech processing research and applications. KBSP (Knowledge-Based Signal Processing System) from MIT was a more general approach by Myers et al. The present version of KBSP is called SPLICE [11]. In these approaches signals are seen as abstract data constructs that are similar to functions in mathematics. The best known similar system specifically for speech research purposes is SPIRE from MIT [12].

AN OVERVIEW OF THE QuickSig SYSTEM

QuickSig is an experimental DSP programming environment that is more engineering oriented than e.g. SPLICE. It is based on the latest object formalism (New Symbols, Symbolics Inc.) which is close to the emerging object standard CLOS. The QuickSig package can be considered as a signal processing extension to the Lisp language.

QuickSig is written in Common Lisp and New Flavors and the system is easily extendable. The hardware environment is the Symbolics 3670 Lisp machine with 470 Mbytes of disk memory (160 MBytes of virtual memory) and a UNIBUS-option for interfacing peripherals like 16 bit A/D and D/A converters for full-range audio signal input and output. Parts of the software run also on the Apple Macintosh II (Allegro Common Lisp).

One of the main features in object-oriented signal processing is the object-oriented structure of the QuickSig system to retain the simple syntax of Lisp, like in scalar computations, e.g.,

\[(+ 1 2) => 3\]

in the domain of signal processing. In the case of signal objects we can define the function "add" to mean additive mixing of the signals, sample by sample, i.e.,

\[(\text{add} \text{sig1} \text{sig2}) \Rightarrow \text{sig3}\]  (a new signal object)

whatever the internal representations of the signals sig1 and sig2 may be. This generic function add can be applied as well to scalar numbers as combinations of scalars and signals. In the latter case the meaning is to add a constant offset to the signal.

Span processing

Signals that are stored as sample sequences are always limited by the index or time interval. Due to this it is important to consider that virtually infinite intervals exist as to make possible e.g. the addition of two signals with different intervals. We use the concept of span for this purpose. A span object specifies an index interval, while a scale-span object specifies an index interval as well as a scale like time or frequency, and a scaler which indicates the sampling rate. Signal objects are constructed upon the scale-span object.

The determination of resulting spans for signal processing operations is called span processing. The signals are normally assumed to have the value 0.0 outside the specified span. If two signals are summed, the resulting signal will have the union span of the arguments. For the case of multiplication the result will correspond to the intersection span of the argument signals.

The function add was given above as an example of an array-oriented signal operation. There are also many functions that return scalar values. Some of these return properties that are results of vector-like operations: e.g. (rms sigx) returns the rms-value of sigx calculated over the signal span. A very useful class of functions is point operations (in contrast to array operations) like at and at-point:

\[(\text{at sig1 10}) = \text{signal sample at index value 10}\]
\[(\text{at-point sig1 10.01}) = \text{signal sample at point (time) 10 ms}\]

The value can be changed by the form

\[(\text{setf (at sig1 10) 1.25})\]

QuickSig object classes

QuickSig consists of object classes that inherit properties from more simple ones and add new features, especially method functions to be more specific. The object hierarchy starts from span and scale-span which describe integer- or real-valued intervals. Based upon these the object class signal is defined with an array to keep the samples corresponding to its span. Windows, correlates, m-signals (multi-channel signals), s-signals (signal-valued signals) and more complicated objects are inherited from the signal class (see Fig. 1). The QuickSig system contains also objects and functions for digital filters and LPC processing, signal databases and event-based symbolic signal representations. The user interface in QuickSig is graphically oriented. There are several standard presentations for signal objects, invoked by function forms like (s-draw sigx) to draw sigx as a curve with coordinates.

![Fig. 1. A part of the QuickSig object inheritance hierarchy.](image-url)

FILTER OBJECTS, FILTERING AND LPC

Digital filtering is one such area where we can successfully benefit from object oriented programming. Not only are input and output signals represented by objects but the filter can also be seen as an object which not only adds and multiplies but also possesses internally a wealth of knowledge.

Filter objects can be implemented in different ways. Our filter structure is based on lower level object classes that were introduced in Fig. 1. When designing new higher level elements we have to keep the design as simple and general as possible while still retaining all necessary information for filtering. So far two kinds of filters have been implemented: a) Basic-filter - a class which includes Direct Form II filters, and b) Lattice-filter - a class for digital lattice-filters. Both of these classes have been implemented using a common inheritance hierarchy (s-polynomial and polynomial), which can be seen in Fig. 2.

![Fig. 2. Inheritance hierarchy of filter objects](image-url)

The class of polynomials (s-polynomial) is based on sequence (signal with scale equal to NIL, see Fig. 1). Poly-ratio is an object class which describes the transfer function having both a numerator and a denominator polynomial (denseq and numseq). Tempseq is an instance variable (an array) which contains the delayed samples within the filter, and end-expand is used for controlling the length of the output.

Filtering is performed by the generic function filter which is implemented on the more specific functions bfilter and lfilter, depending upon the filter class. When calling these functions it is possible for example to define the span of the output signal by using the keyword out-span and to give the initial state of the filter with the keyword initial-state. The filtering function also makes decisions regarding the fastest way of filtering. Methods for graphical z-domain presentation of filters are available.

Linear prediction is one application for the filter object formalism in QuickSig and has been implemented using an object oriented strategy. A natural result of applying LPC-analysis to a signal frame is a filter object (an LPC inverse filter) and the LPC residual signal. When analyzing a complete signal a list of two s-signals (see Fig. 1) is returned. The first is a sequence of inverse filters and the second a sequence of time domain signals representing the LPC residuals. This is a good example of conceptual clarity gained by the object-based abstraction mechanism. All the details of LPC analysis are easily and flexibly available by using the object hierarchy.
EVENT-BASED SIGNAL REPRESENTATIONS

The full power of object-oriented signal processing is not only in the systematic way of implementing numeric algorithms but in the ability to express qualitative, structural and symbolic aspects as well. The objects and functions described above are primarily numerically oriented but they include many features of symbolic nature. In QuickSig there are also approaches that are intended specifically for structural and rule-based representation of signals.

One approach is based on the use of events and event structures to represent signals [13]. An event is conceptually defined as any relatively prominent structural unit (in time or frequency or both). In our system event analysis is carried out by a method that we call multiple-resolution analysis. It has resemblance to scale-space filtering proposed by Witkin [14] and to wavelet theory [15]. An extra dimension of resolution (or scale) is introduced. The input signal is processed by resolution filters that emphasize the structures with corresponding resolution. The events are detected as extrema (maxima, minima, zero-crossings) of the filtered responses. Finally, the events are parsed to form composite events and event structures according to the rules of the application domain.

Our applications of the methodology are related to auditory modeling and speech recognition. Fig. 3 shows an example of an event map for the auditory spectrum of a vowel. The bottom window shows the auditory spectrum, while the top window displays resolution vs. frequency. The circles indicate the most prominent spectral points. We have found that different resolution levels are applicable in different contexts so that for instance resolution of approximately 4.0 gives a good overall characterization of the coarse structure of the auditory spectrum.

Events and rule-based processing in speech recognition

Event-based multiple-resolution analysis and representation of speech signals is well suited to knowledge-based speech recognition. We have developed a methodology for and a preliminary prototype of unlimited vocabulary recognition of Finnish [16]. Auditory spectrum analysis is used as pre-processing and several levels of event-based representations are created. Matching of diphone prototypes leads to phoneme string hypotheses that are finally processed by a morphological filter to find the best word match in Finnish.

The recognizer can be considered as a rule-based system where signal and event objects of the QuickSig system are used extensively. The rules for recognition were written in procedural form as method functions for the objects. This was important due to achieve a reasonable computational efficiency since rules written in the form of production systems or logic programming tend to be relative slow. A general rule formalism should be included, however, in a future version of the QuickSig system.

REPRESENTING RELATIONS AND OBJECT STRUCTURES

Structural descriptions of objects are common in symbolic manipulation of speech representations. The only relations that object-oriented languages like Flavors or CLOS include are the superclass-subclass inheritance and the class-instance relation. For a systematic support of more advanced relations and structures we have programmed a package of so called relation-mixins to be included to object class definitions in order to add desired relation properties. The most important relation-like features are:

- parts-mixin, part-of-mixin, part-mixin
- next-mixin, prev-mixin, seq-mixin
- type-mixin
- prop-mixin

Relations part and part-of are used for structure-forming objects. Both of them are included in part-mixin. Relations next and prev and their combination as seq-mixin support one- or two-way linked sequences of objects. Type-mixin is used to describe more flexible class relationships than is supported by normal inheritance. It allows for fuzzy membership functions that stand for e.g. to estimate the phoneme class of a phoneme object in speech recognition. All relation-mixins support also a special feature called aspect in order to have any number of parallel relations of same type but different aspect. One more important mixin is prop-mixin that is used to add a flexible property list to desired objects.

Relations and structures in speech synthesis by rule

The relation-mixins are used e.g. in speech synthesis by rule to to form an explicit structural representation of units in speech [17]. The input text is analyzed and parsed into a structure as shown in Fig. 4. Part-mixin as a part of objects allows for a hierarchical structural representation of speech units and seq-mixin is used to link sequential relations of objects on each level. By proper method functions attached to the objects it is possible to manipulate easily the structures as well as to apply rules to specify the context-dependent parameters that are needed for the synthesis process.

Fig. 3. An event map of a vowel auditory spectrum as an example of event-based signal analysis. Bottom: auditory spectrum of vowel /a/. Top: most prominent events (circles) and scale-space contours in frequency vs. resolution domain.

Fig. 4. A hierarchical structure of speech units used in speech synthesis by rule.

OBJECT-BASED ALGORITHM SPECIFICATION AND CODE GENERATION

To further enhance the flexibility and interactivity of algorithm development we have been experimenting with a block diagram compiler with graphic editor facility and code generation for DSP processors. This part of QuickSig is still preliminary and not fully integrated to other parts of the system.

Several software implementations for graphics-based block diagram computations exist: see e.g. systems by Zissman & al. [18] and Covington & al. [19]. More recently Lee & al. have described GABRIEL [20] that supports also code generation for signal processors.

The experiments of block diagram compilation in the QuickSig system are based on extensive use of object-oriented
programming. There are separate block object classes for all basic DSP operations like constant block, unit delay, adder, multiplier, generalized function block, etc. These blocks can be wired to form diagrams. They can further be named and defined as new classes of composite blocks. Other basic objects are nodes and io-ports (input ports, output ports). Wires between nodes are also objects but they are used only for the user interface, not for computation logic.

Each computation block has a description of its internal structure. It consists of input and output ports (as objects), internal variables to keep special definitions (e.g., constant value for a constant block), and a list of graphic presentations of the object for user interface.

A unique feature of the block objects is their ability to generate computation forms and corresponding compiled functions. Each block class includes method functions that can manipulate Lisp expressions and compile them in a way that is specific to the class. The main idea is to generate and attach a Lisp form to each input and output port of a computable block. This means that after the compilation of a block a computation process is initiated and can be used for computation logic.

To have the highest flexibility for DSP programming the environment must include all programming levels from assembly code to high-level abstractions. For this we have been integrating the assembler and other low level tools for DSP chips to the QuickSig system [5]. Currently the main interest is to generate code for the third generation signal processor TMS320C30 that supports also floating-point arithmetic. This part of the software environment can be linked to the block diagram compiler so that highly efficient machine code will be generated directly from block diagram specifications of DSP programs.

The main idea for the integration of high- and low-level programming in our system is to use a modified assembler syntax for the TMS320C30 that is entirely compatible with Lisp notation. This immediately makes it possible to call any assembler instruction from the Lisp interpreter (interactive simulation) and to include instructions in any high-level Lisp programs that are written in source-code level simulation including QuickSig-level functions. The modified list notation vs. normal assembly syntax is compared in the following examples:

**Normal syntax:**
```
LD F -12.3, R0
ADD F3 *AR0 + (IR0), R0, R1
```

**Lisp syntax:**
```
(LDF -12.3 'R0)
(ADD F3 '(AR0) + (IR0) ', R0 ', R1)
```

These instructions are for loading a register (R0) with a constant and for adding two numbers (indexed indirect and register RO) to another register (R1). All Lisp tools are immediately available for the assembler. Writing "intelligent" macros is an especially powerful feature. The next level that we have experimented on top of it is an optimizing Lisp compiler to produce efficient code for numeric computation.

The assembler with Lisp syntax is as such not based on object-oriented programming. It follows normal functional and procedure-oriented syntax so that assembly instructions are list expressions. Due to this the instructions can be evaluated interactively or assembled to machine code. It is possible to simulate or emulate with multiple processors that are real hardware ones or simulated software constructs. By object-based formulation both kinds of processors have very similar appearance and thus they can be called generic processors.

**SUMMARY**

An experimental object-oriented signal processing environment QuickSig is described with some application examples related to speech processing. The system is found to be a very flexible platform for exploratory programming and algorithm development. Presently the system runs on Symbolics Lisp machines and parts of it on Apple Macintosh II. The future versions will be adapted to the emerging object standard CLOS which will make the system portable.

**ACKNOWLEDGEMENTS**

The authors are grateful to the Academy of Finland which has financed this research and to SITRA Foundation that has supported the development of the speech recognition application.

**REFERENCES**


