D8.4

QuickSig - AN OBJECT-ORIENTED SIGNAL PROCESSING ENVIRONMENT

Matti Karjalainen, Toomas Altosaar, Paavo Alku

Helsinki University of Technology
Otakaari 5A
SF-02150 Espoo FINLAND

ABSTRACT

A new object-oriented DSP environment called QuickSig is described. It is based on the latest developments in object-oriented programming (New Flavors on Symbolics Lisp machines). The design philosophy of QuickSig has been to extend the Lisp language by a layer of general DSP constructs; abstract data structures like signals, filters, windows, graphical presentations and related signal processing operations. QuickSig is targeted to be a fast prototyping system for algorithmic development. It is easily extendable to include new ways of modeling signals and signal processing, both numerical and symbolic. This paper describes the main features of the present system and some new features that are under development.

INTRODUCTION

Traditional digital signal processing (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 and do not exist as an integral part of the program or of the programming environment.

The integration of artificial intelligence and DSP is creating a new area of research that shows promising results and perspectives. Knowledge-based and symbolic signal processing, signal interpretation etc. are some of the new terms used to describe this field. Object-oriented programming is one of the most successful approaches as a basis for integrated signal processing systems. A high level of abstraction is reached where not only numeric computation but also rule-based logic is easily applicable.

Object Oriented Programming

Object oriented programming has evolved with advances in AI methodology and modern programming languages. New object formalisms are emerging and the approach is becoming a widely accepted extension to traditional languages like C and Pascal. *Simula* is often referred to as one of the first languages that included object-based abstraction features. *Smalltalk* [1] from Xerox is a well known "puristic" object language, where everything is made of object classes and instances.

Lisp is found to be a good basis on top of which a powerful and practical object formalism can be implemented as an extension of the language. *Flavors* and *New Flavors* [2] are commercially available object extensions on top of the Common Lisp for Symbolics Lisp machines. *Common Loops* is an object environment from Xerox for their Lisp machines. In the near future these and some other object languages will be merged into the *object standard* for Lisp programming. The object orientation is well suited to engineering applications like knowledge-based signal processing. At the present time advanced object languages have started to emerge also in the personal computer domain, e.g. Smalltalk and Lisp implementations on the IBM PC and Apple Macintosh.

A Typical Object Oriented Programming Environment

The objects exhibit an abstraction mechanism where the implementation details are hidden so that the user can rely on a systematic and fairly simple interface between the objects and the external world. The most common features in object languages are:

Definition of *object classes* as a general model form of any instance object of that class. The class definition includes a list of variables internal to the objects. The instances keep an internal state by the values of these *instance variables* (= state variables, ivars). *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 computation 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 latter syntax has good compatibility with the Lisp language. Generic functions are one of the starting points in the realization of the QuickSig system.

SYMBOLIC AND KNOWLEDGE-BASED SIGNAL PROCESSING SYSTEMS

There exist several implementations of signal processing environments based on object-oriented programming. The most advanced systems have used *Flavors* running on Symbolics Lisp machines. G. Kopec [3] formulated the concept of signals as objects. Later Kopec has introduced *ISP* (Integrated Signal Processing System) [4], *SRL* (Signal Representation Language) [5] 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* [6]. It is shown to be easily applicable to the study of practical problems, see Dove [7] and Milios [8].

Signals as Objects

In *signal abstractions by objects* of the ISP and SPLICE systems a signal is seen as a function or mapping from the index (integer) domain into the sample value domain. It is possible to avoid the limited interval of numeric samples by assuming some function or default value that extends the explicitly supported range of the signal virtually to include all index values between $-\infty$ and $+\infty$. Signals are created by objects called systems that are like function generators.

CH2561-9/88/0000-1682 $1.00 © 1988 IEEE

There are several fundamental properties of signals in the SPLICE system. Signals are seen as *immutable* objects that cannot be changed. The delayed or *deferred evaluation* paradigm means that the numeric sample values are computed only when needed. The concept of *deferred array* is used to buffer the computed values for possible future use so as to avoid recomputation. The interval of the buffer array may change according to the needs of computation and the buffering is transparent to the user.

AN OVERVIEW OF THE QuickSig SYSTEM

QuickSig is an experimental DSP programming environment that is general (not application specific) and more engineering oriented than SPLICE. It is based on the latest object formalism (New Flavors, Symbolics Inc. [2]) which is close to the emerging object standard. Common Lisp has been the main programming language due to its flexibility and powerful representation features. The QuickSig kernel can be considered as a signal processing extension to the Lisp language.

The current size of QuickSig is more than 10 000 lines of Lisp code written in Common Lisp and New Flavors. QuickSig 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.

One of the main features in object-oriented signal processing of the QuickSig system is 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.,

(add sig1 sig2) => *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.

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 them comes the object class *signal* 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, a graphical user interface, signal databases, block diagram compilation and event-based symbolic signal representations.

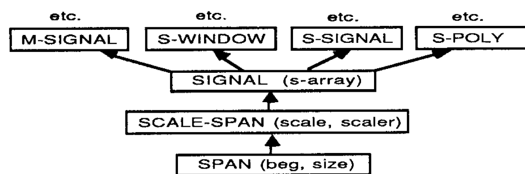

Fig.1. A part of the QuickSig object inheritance hierarchy.

SPAN AND INTERVAL PROCESSING

Span is a low level object class which provides a foundation for the signal object system. A *span* object has two primary integer-valued properties: *beg* (the first index included) and *size* (the number of index positions included in the span). Some related secondary properties are *end* (= beg+size, not included) and *stop* (= *end*-1, is included). Span objects are convenient for index range computations in signal processing operations and they can be created by the form *(make-span beg end)*, e.g.

(setq spanx (make-span -3 20))

where setq assignes the symbol spanx with a new span object. There are access functions with the names *beg*, *size*, *end* and *stop* that can be used to read or change the properties of an object *spanx*, e.g.,

(end spanx) returns the end index of spanx

(setf (size spanx) 120) sets the size of spanx to be 120.

An important part of span processing consists of set-theoretical span computations with *intersection-span* (see Fig. 2), *union-span*, *correlation-span* and *convolve-span*, e.g.

(convolve-span sp1 sp2 [to-span])

where *sp1* and *sp2* are objects inherited from *span*. *To-span* is optional and is created if not given. The function returns the span that results when convolving signals with spans *sp1* and *sp2*.

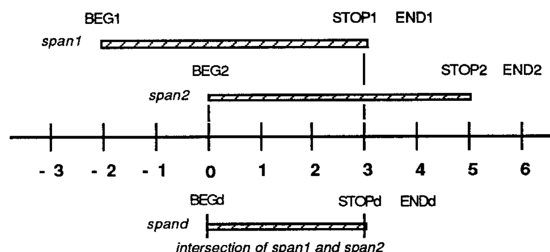

Fig 2. Processing of the intersection-span of two spans

The *scale-span* object class inherits all propreties and method functions of span and adds the properties *scale* and *scaler*. *Scale* keeps a symbol to denote a scale like time, frequency, position, etc. *Scaler* is a real-valued number to relate index values to continuous scale points (e.g. 1/sample-frequency for time scale signals). Secondary properties *beg-point*, *end-point*, *stop-point* and *scale-size* correspond to the index properties on the real-valued *scale* and can be accessed by method functions like

(setf (stop-point scale-spanx) 0.5)

sets the stop index so that the stop-point will be 0.5 rounded to the nearest index. There are method functions to create and copy *scale-spans* and to check the scale-compatibility of two scale-span-inherited objects.

Interval is still another kind of range object that has the primary properties *beg-point*, *end-point* and *scale* with the corresponding access functions. Intervals are not related to index numbers in any way. They can be manipulated e.g. by method functions *union-interval*, *intersection-interval*, *scale-span-to-interval* and *interval-to-scale-span*. The need for intervals and scale-spans as separate objects arises from the difference between discrete indexes and real-valued points as well as from the roundoff error when converting between them.

SIGNAL OBJECTS

Signal, the main object class of the QuickSig system is inherited from scale-span and includes a new property *s-array* (sample array) to keep the samples for the defined span range. By default the signal samples outside this range are considered to have the value 0.0 if scale is defined (NIL if scale is NIL) even if these default values are not stored explicitly.

Signals in QuickSig, contrary to SPLICE, are not immutable. This violates the pure functionality and mathematical elegance of signals as functions but introduces more practicality because a signal can be changed and reused as many times as is needed. In many cases the result of operating on a signal can be directed back into the same signal object. All properties of a signal may change. An important feature is the ability to dynamically change the span of a signal, explicitly or implicitly, as a result of a signal processing operation.

Basic Functions for Signal Manipulation

The QuickSig system has several functions for the generation of signals. One of the most powerful forms is by the syntax:

make 'signal &key *function funscale span scale scaler*,

where the arguments after &key are optional keyword arguments. A typical example of signal generation is:

```
(setq sigx
  (make 'signal :function #'(lambda (x) (sin (* 2 pi 1000 x)))
                :span (make-scale-span -1.0 1.0)
                :scaler (/ 1.0 16000)))
```

This returns a sinusoidal time signal (*time* is the default for *scale*) with a sampling frequency of 16 kHz, a span range from -1.0 to 1.0 (seconds) and a frequency equal to 1 kHz. If the keyword *:funscale* is used instead of *:function*, the domain $0 \leq x < 1$ of the function will be mapped to correspond to the specified span.

There are functions for copying a signal (*make-copy*), signal "editing" by functions *cut-signal* (cut and return a part), *insert-at-point*, *s-reverse* (to reverse the signal samples within a span), *s-concat* (concatenation of signals) and *s-sort* (sorting of samples), changing the *span* explicitly (*span-adjust*), shifting (*shift*, *scale-shift*), changing the sampling frequency (*up-sample*, *down-sample* by an integer ratio) and testing properties (*real-p* and *compatibility* of two signals by *scale* and *scaler*). The access to individual samples is by the functions *at* and *at-point*, e.g.

(at-point sigx 0.5) returns the sample at time 0.5 seconds,

(setf (at sigx 100) 1.0) assigns the sample value 1.0 to the index position 100. If a value outside the span is requested the default (0.0) is returned. If a sample is stored outside the existing span, the *span* and *s-array* are automatically adjusted to include the new sample.

Functions for inquiring scalar-valued properties of signals are e.g. *max-min* (returns the max and min values within a span), *abs-max* (= max of (abs max) and (abs min)), *sum*, *mean*, *sqr-sum*, etc., all of them over an optional span (default is the total signal).

Array-Oriented Signal Processing

A large part of signal processing is carried out in a way that can be called *array-processing*. It is advantageous due to its high speed of loop-like operations. A fundamental part of QuickSig is devoted to array-oriented DSP although it is not the only approach (see block diagram approach below).

Most of the basic functions for signal manipulation mentioned above were actually performed by array-oriented processing, where a loop is run over a specified span to return the desired scalar or signal value. An important class of array processing consists of arithmetic operations, especially those used in linear signal processing, eg.,

(add sig1 sig2 :to sig3 :out-span (make-span 0 10)).

QuickSig supports a set of generic signal functions that may take scalar or signal-valued arguments. Among them are *add*, *sub*, *mul*, *div*, *square-root*, trigonometric functions, etc. This set is easily extendable to include any function that is needed. As a generalization the function *s-call* can be used to apply any Lisp function to signals or scalars by the syntax:

s-call *funct obj1 [obj2]* &key *to out-span type-check mix-mode*

where *funct* is any Lisp-function that acts as a point operator, *obj1* and optional *obj2* are signals or scalar numbers, *to* is the resulting signal (which may also be *obj1* or *obj2* if non-scalars), *out-span* is the optional output-span (default depends on the inputs and type of *funct*), *type-check* controls if compatibility is checked and *mix-mode* determines if the result is written or added to *to*.

Signal Windowing

Signal windowing in QuickSig is carried out by calling a generic window function by the form:

window *sigx wndx* &key ....

where *sigx* is the signal to be windowed, *wndx* is the window object to be applied and the optional keyword arguments include *window-point* or *window-span* to specify the index or scale position of the window, *window-out-point* to indicate the index or scale position of the resulting signal (*to*), etc. The windowing operation can also be imbedded in some other signal processing forms like FFT and correlation as a pre-operation by using a *:prewindow* keyword syntax.

There are two main types of windowing objects available. *s-window* is a signal-like object which is in a numerically sampled form having a fixed length. *f-window* is more flexible because it is represented as a function that can be scaled by length at the time of application to a signal. The window function itself must be specified in the domain $0 \leq x < 1$. The most common types of windows are predefined in QuickSig as well as functions for the generation of new window types.

Convolution, Correlation, and Transforms

Convolution, correlation, and several different types of transforms are useful elements in array-based signal processing. QuickSig uses efficient methods to realize these operations. Different method functions exist to calculate the Fourier transform of a signal depending upon its properties (e.g. real or complex valued). Run-time cost analysis is performed to determine whether it is faster to use a direct method or a transform (along with optimum blocking size given a set of available transforms) to calculate a convolution or a correlation. The user has the option to avoid the slight overhead induced by cost-analysis by being able to apply functions directly if signal properties are fixed and known a priori. Temporary signal variables are frequently needed with these operations and therefore a *resource pool* of signals is used reducing the need for garbage collection.

FILTER OBJECTS, FILTERING AND LPC

Filtering is one of the most important areas in signal processing. It is perhaps most clearly characterized by traditional ways of thinking with attention especially being payed to numerical computations. 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 posseses 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 tried 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-poly* and *poly-ratio*), which can be seen in Fig. 3.

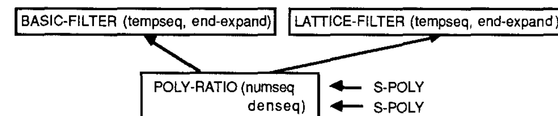

Fig. 3. Inheritance hierarchy of filter objects

The class of polynomials (*s-poly*) 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 (poles and zeros) 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.

BLOCK-DIAGRAM COMPILATION AND GRAPHICAL EDITING OF DSP ALGORITHMS

There is another approach to computation in object-oriented signal processing that will be added to the QuickSig environment. It includes the use of a graphical interface to edit block and flow diagrams that will compile into efficient Lisp code for later execution. This part of the system is described here only briefly because of the preliminary nature of the realization.

There are separate *block object classes* for all basic DSP operations like *constant block, unit delay, adder, multiplier, generalized function block*, etc. The blocks can be wired to form diagrams which can further be named and defined as new classes of *composite blocks*. Other basic objects are *nodes* that are divided into *input ports, output ports* and *wire nodes*. Wires between nodes are also objects but they are used only for the user interface, not for computational 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 interfacing. Each computation block also contains a set of method functions (local functions) as an interface to the computational environment.

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 instances of the class. The main idea is to attach a compiled function object to each input and output port of a computable block. This means that a computation step (index or time step) can be activated from any input (input driven) or output (output driven) and propagated through the connected part of a block diagram. A useful feature of the system to be utilized in the future is that instead of generating compiled code for the Lisp machine it is possible also to use other target machines, processors or languages, especially the floating-point signal processor chips that will be available soon.

A graphic editor will be an essential part of the system for the creation of DSP algorithms in the form of flow and block diagrams. Thus all the QuickSig computation block objects, signal objects, etc., have corresponding presentation objects which govern the graphical presentations. The presentation objects are linked to the corresponding computation objects. The graphic editor is used interactively by the mouse, menus and keyboard.

OTHER FEATURES OF THE QuickSig SYSTEM

Object-oriented programming allows for the systematic extension of a complex DSP system. In this section we mention some of the other major features of the system.

An important part of any DSP tool is the graphics interface. QuickSig supports several classes of display objects that can be used easily. For instance the generic function form

(**draw** objx [&key options ...])

is able to take signal-inherited objects (*objx*) and draw them in several ways. There are displays and layouts available for drawing combinations of DSP objects as seen in Fig. 4. The graphics interface will be enhanced using mouse-sensitive presentation objects and it will be integrated into the graphics editor of the block diagram compiler.

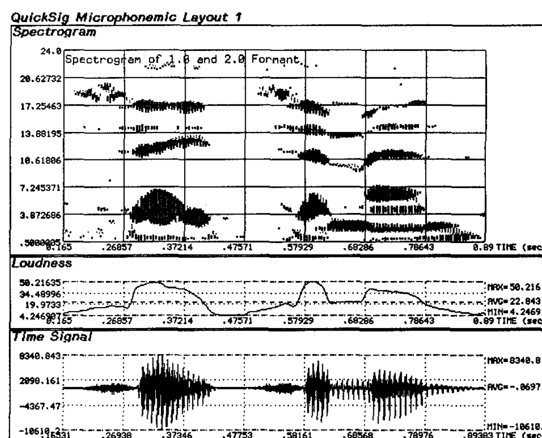

Fig. 4. An example of a QuickSig display: auditory formant spectrogram, loudness function and the corresponding speech signal time waveform.

QuickSig objects can form *signal databases*. In the current implementation this serves primarily as a file storage mechanism for the objects and a way of naming the objects apart from the Lisp symbol name space. The objects can use a flexible set of *relation-mixins*; special features to represent various kinds of relations between the objects. The file storage of these relations is not yet supported.

There is also an experimental formalism to analyze and represent signals by events and event structures. This allows for symbolic and rule-based processing of signals (see this proceedings [9]). QuickSig includes many application-specific features as well, especially for speech analysis, synthesis, recognition and auditory modeling studies.

ACKNOWLEDGEMENTS

This work is supported by the Academy of Finland. The authors are grateful to all those who have contributed to the ideas and development of the QuickSig system.

REFERENCES

[1] Goldberg A., Robson D., Smalltalk-80. The Language and its Implementation. Addison-Wesley 1983.

[2] New Flavors documentation in: Symbolics Common Lisp Language (2A). Symbolics Inc., Cambridge, Mass., 1987.

[3] Kopec G., The Representation of Discrete-Time Signals and Systems in Programs. MIT PhD Thesis, Cambridge 1980.

[4] Kopec G., The Integrated Signal Processing System ISP. IEEE Transact. ASSP, vol. ASSP-32, No. 4, August 1984.

[5] Kopec G., The Signal Representation Language SRL. IEEE Transact. ASSP, vol. ASSP-33, No. 4, August 1985.

[6] Myers C., Signal Representations for Symbolic and Numerical Processing. MIT Technical Report No. 521, August 1986.

[7] Dove W., Knowledge-Based Pitch Detection. MIT Technical Report No. 518, June 1986.

[8] Milios E., Signal Processing and Interpretation using Multilevel Signal Abstractions. MIT Technical Report No. 516, June 1986.

[9] Altosaar T., Karjalainen M., Event-Based Multiple-Resolution Analysis of Speech Signals. In this Proceedings.