QuickSig - AN OBJECT-ORIENTED SIGNAL PROCESSING ENVIRONMENT

Matti Karjalainen, Toomas Altosaar, Paavo Alku
Helsinki University of Technology
Otukaari 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 machine. 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.

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 Milton [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 -∞ and +∞. Signals are created by objects called systems that are like function generators.

1682
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.,

\[
(+ 1 2) \rightarrow 3
\]

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

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

whatever the internal representations of the signals \texttt{sig1} and \texttt{sig2} may be. This generic function \texttt{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 \texttt{span} and \texttt{scale-span} which describe integer- or real-valued intervals. Based on them comes the object class \texttt{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

\texttt{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: \texttt{beg} (the first index included) and \texttt{size} (the number of index positions included in the span). Some related secondary properties are \texttt{end (= beg+size, not included)} and \texttt{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 \texttt{(make-span beg end, e.g.)}

\[
\texttt{(setq span (make-span -3 20))}
\]

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

\[
\texttt{(end span)} \quad \text{returns the end index of span}
\]

\[
\texttt{(setf (size spans) 120)} \quad \text{sets the size of spans to be 120.}
\]

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

\[
\texttt{(convolve-span sp1 sp2 [to-span])}
\]

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

\texttt{Fig. 2. Processing of the intersection-span of two spans}

The \texttt{scale-span} object class inherits all properties and method functions of \texttt{span} and adds the properties \texttt{scale} and \texttt{scaler}. \texttt{Scale} keeps a symbol to denote a scale like time, frequency, position, etc. \texttt{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 \texttt{beg-point}, \texttt{end-point}, \texttt{stop-point} and \texttt{scale-size} correspond to the index properties on the real-valued \texttt{scale} and can be accessed by method functions like

\[
\texttt{(setf (stop-point scale-spans) 0.5)}
\]

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

\texttt{Interval} is still another kind of range object that has the primary properties \texttt{beg-point}, \texttt{end-point} and \texttt{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 \texttt{union-interval}, \texttt{intersection-interval}, \texttt{scale-span-to-interval} and \texttt{interval-to-scale-span}. The need for intervals and scale-spans as separate objects arises from the difference between discrete index and real-valued points as well as from the roundoff error when converting between them.

SIGNAL OBJECTS

\texttt{Signal}, the main object class of the QuickSig system is inherited from \texttt{scale-span} and includes a new property \texttt{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 in many times as it 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 ≤ 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, 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, 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 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 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.

**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 possesses internally a wealth of knowledge.

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

```lisp
(draw obj &key options ...)
```

is able to take signal-inherited objects (obj) 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.

**REFERENCES**


