Microprocessing and Microprogramming 32 (1991) 541–548
North-Holland

An Applicative Real-Time Language for DSP-Programming Supporting Asynchronous Data-Flow Concepts

A. Knoll and M. Freericks
Technische Universitaet Berlin
Sekretariat FR 2-2
W-1000 Berlin 10
Germany

An applicative language designed to express parallelism and data-flow in signal processing systems is presented. The language may be used for the specification of networks consisting of any number of processing entities. It is equally well suited for the simulation of networks both at the data flow level and at the algorithmic level. The language constructs have been selected so as to to obtain concise descriptions of typical signal processing tasks. The paper discusses several key aspects of the language design; some simple examples illustrating the power of the language are included.

1 INTRODUCTION

Applicative (functional) programming languages have several advantages over standard imperative languages: The abstraction from machine and algorithmic details is much higher, therefore a program specification is normally considerably closer to the problem to be solved. As a consequence, programs written in an applicative programming style are normally much shorter than their imperative counterparts, they are nicer to read, easier to comprehend, simpler to write and less error-prone. In recent years, there has been a number of attempts to design applicative languages for signal processing applications [1;2], most notably Silage [3], a comparatively small and simple language used primarily in the realm of VLSI design. Applicative languages may be used stand-alone or as "natural" target languages of graphic signal processing design systems such as Gabriel [4] or CADiSP [5] because processing blocks in these systems map directly to function applications of applicative languages.

The programming language ALDiSP [6] is a general-purpose applicative language primarily designed for DSP algorithm and network specification. It may be used both for the simulation of complex asynchronous networks and the generation of code for typical synchronous signal processing tasks implemented on standard or dedicated signal processors. The expressive power of the language is comparable to that of Scheme [7] or ML [8;9], and it can be applied to problem domains other than DSP programming. In the sequel, however, we will focus on the language constructs relevant for DSP applications. The major design goals to make the language useful for DSP were:

- Time and process management must allow for the handling of both synchronous and asynchronous events
- The programmer must be able to handle infinite streams of data objects
- A rich set of (parameterized) numerical types should be defined to accommodate as many DSP architectures as possible
- Powerful operators on complex data structures (most notably arrays) should be available
- The language should be able to model numerical properties of common DSP architectures in a bit-by-bit manner
- Exceptions (both numerical and algorithmical) should be handled easily
- There should be constructs permitting the division of programs into modules

2 LANGUAGE FEATURES

General Program Structure

An ALDiSP program models a signal flow graph, i.e. a net of *signals* connecting *nodes*. Nodes are specified by user-defined or pre-defined functions; signals are modelled by *streams* and *pipes*.

A program may consist of an arbitrary number of *modules* and a *net description*. The latter describes the top-level flow of data and the connection of signals to I/O-devices.

Signal processing may be *input-driven* or *output-driven*. In an input-driven ("call-by-availability") environment, a node remains dormant until there are enough input data available to activate the node; on activation the node consumes a number of input tokens and pushes a number of output tokens into the net. In an output-driven ("Call-by-need") network output devices "pull" data tokens out of the network. When a node receives a request to produce an output token, it propagates the request to its input. Obviously, this propagation works only if there are functions at the end of the propagation path that produce output tokens without prior input, i.e. functions acting as *signal generators*.

In ALDiSP, signals are modeled as *infinite lists* of data tokens. The programming language offers two different kinds of lists: *Streams*, which are a well-known data structure in functional languages [10], model output-driven signals. *Pipes*, on the other hand, model input-driven signals. The elements of a stream are produced by a stream-generator, which is simply an ordinary function supplying the elements of a stream upon request, one at a time. These elements typically represent successive samples of an algorithmically definable signal. At any given time, only a finite set of stream elements is known while the remaining infinite list of stream elements remains to be determined.

The following ALDiSP program illustrates the usage of streams by modeling a tapped shift register via a stream called "RandomBits". It can be used as a noise signal generator. The generator produces an infinite sequence of pseudo-random bits. The initial seed (a large cardinal number) is divided into a stream of bits. The stream is tapped by a function GenBitStream, which generates new pseudo-random bits by XORing a number of bits on the stream.

Fig. 1 shows the circuit diagram of the noise generator; fig. 2 shows how the algorithm works in terms of a data flow model. The program implementing this network is shown in fig. 3, where ">>" denotes a right shift of bits, "&" is the logical AND of bits, and "::" is an operator that puts a token at the head of a stream.

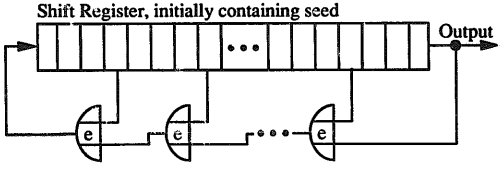

Fig. 1: Circuit diagram of the noise generator

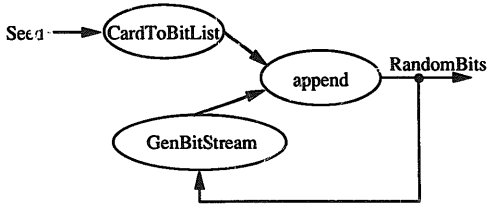

Fig. 2: Data-flow model of the noise generator

```
\ ---- Seed and length of shift register

     Seed = 745608675123538549606958473849506847389;
     ShiftRegLength = 100;


\ ---- Function dividing a number into a stream of single bits

     func CardToBitList (n, Index) =
       if Index < 0 then Null
         else ((n >> Index) & 1) :: CardToBitList (n, Index-1)
       endif;


\ ---- Vector holding the indices of the bits to be XORed, may
       have any number of Elements < ShiftRegLength

     XORBitVect = [4, 17, 39, 99];


\ ---- Function XORing the numbers as indicated by
       XORBitVect in a stream
\ ---- Note: select is a predefined function selecting the nth
       element of a list and the predefined function reduce
       applies a binary function to a vector or a list

     func GenRandomBit (list) =
       let
         func Extract (n : Card) = select (list, n)
       in
         reduce ('bitXOR', Extract (XORBitVect))
       endlet


\ ---- Function applying the random number generator
       successively to the shift register and then shifting it

     func GenBitStream (stream) =
       GenRandomBit (stream)::GenBitStream (tail(stream))


\ ---- Netlist describing the topography of the network

     net
       RandomBits =
       append (CardToBitList (Seed, ShiftRegLength)
                delay (GenBitStream (RandomBits)))
     in
       StreamToPort (AnOutputPort, RandomBits)
     endnet
```

Fig. 3: Program implementing the noise generator

The function Extract, which is based on the predefined function select, returns the n^{th} element of a list; the first element having the index 0. If it is called with the index n being an array, the automapping mechanism built into ALDiSP will return an array of selected objects. An example call Extract ([1, 3, 2]) applied to a list (10, 20, 30, 40, ...) will give an array of the form [20, 40, 30]. The auto-mapping mechanism in ALDiSP works as follows: If a function is defined for numbers but applied to a stream (or an array), it is automatically mapped to each element of the stream (or the array). The results are collected into a new stream (or array). The predefined function append appends a list or a stream to a stream. The delay operator delays the evaluation of its argument until its value is needed. This operator is implicitly contained in the stream-building operator "::" causing it to have a "call-by-need" semantics.

The topography of the network, i.e. the flow of streams between nodes containing processing functions is described in the *net*-part of the program. The program shown is complete, i.e. no additional memory management, variable assignments, etc., is necessary. The program is driven by the predefined function StreamToPort, which writes stream elements to an output port whenever the port can accept data.

Consider the stream processing system of fig. 4: A signal generator produces a stream of scalar elements. A functional block "ConstructVect" collects this sequence of scalar elements into blocks or vectors of, say, 256 elements. Each of these vectors is transformed into the frequency domain by a Fourier transformer. The transformed vectors (now of complex type) on the stream are filtered in the frequency domain by multiplying the amplitude of each vector component by a corresponding attenuating coefficient and leaving the phase unchanged. The result is then transformed back into the time domain and consumed by further processing entities. Assuming the functions

```
SigGen () = ...                  \ -- Generate signal
FFT (aVector) = ...              \ -- input: Real vector,
                                      Output:Two real vectors
iFFT (twoVectors) = ...
AttnCoeffs = [c1, c2, .... ]     \ -- Vect. of 256 spectral coeff.
```

have been defined and collected in a module called FreqDomainFilt. Then, the complete ALDiSP program implementing the system according to fig. 4 looks as shown in fig. 5.

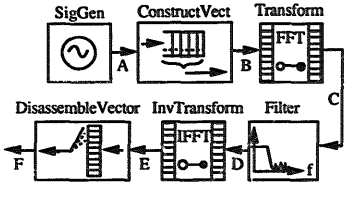

Fig. 4: Network for filtering blocks of a scalar signal

```
Import SigGen, FFT, IFFT, AttnCoeffs from FreqDomainFilt

\ ---- Note that VectorizeStream and ListifyVectors are
       predefined functions

Net
      A = SigGen ();
      B = VectorizeStream (A, 256);
      (Ampl, Phase) = FFT (B);
      D = Ampl * AttnCoeffs;          \ -- Do the filtering by
                                          multiplying the
                                          vectors elementwise
      E = IFFT (D, Phase);
      F = ListifyVectors (E)          \ -- Now we have a
                                          stream of scalars
in
      F
Endnet
```

Fig.5: ALDiSP program for the network of Fig. 4

Note that FFT returns two separate vectors for amplitude and phase which are sent onto two different streams Ampl and Phase. If these streams are kept separate, the manipulation of only the amplitude vector is very easy. Every ALDiSP function may generate multiple return values obviating the need to "sequentialize" compound results before sending them onto a stream. The automatic mapping of functions on finite data structures is employed by the program line doing the filter: The operator "*" is applied to all elements of the vectors Ampl and AttenuatingCoeffs yielding a new scalar vector. In general, if a function is not defined to manipulate an array, but is applied to one, it is applied automatically to all its elements. This holds for user-defined functions as well as for predefined functions. A comprehensive set of functions manipulating arrays as complete data units has been defined (see [6]); the need to manipulate single elements of arrays will hardly ever arise.

This example clearly shows how well the applicative approach of programming is intuitively compatible with the DSP-engineer's block-diagram oriented way of thinking. The automatic mapping of functions on stream elements and particularly the automatic mapping of functions on arrays makes the process of translating block diagrams into programs extremely simple. The programmer need not be concerned with breaking down complex data structures and subsequently manipulate their elements before reassembling them again. Instead, he will manipulate data objects as a whole and the mapping process is done implicitly by the compiler.

Like other modern applicative languages, ALDiSP is polymorphic, i.e. functions may accept parameters of different types. The compiler detects type inconsistencies by analyzing the type of operations applied to a given function parameter (see the following example). The evaluation strategy is call-by-value, for functions generating streams this may be changed to call-by-need by the delay operator (which is implicitly contained in the "::" operator used for assembling lists or streams). Like functions, operators may be overloaded to accept different types of operands. Operators may also be defined to be used in pre-, post-, or infix-notation, whichever is more convenient. Operator precedence may also be specified. ALDiSP permits recursion and higher order functions, i.e. functions that have other functions as arguments and/or return other functions as results. Streams may be defined recursively and they may be delayed by an arbitrary number of elements. Therefore the concept of stream processing lends itself ideally to the implementation of filter networks. The example IIR-filter of fig. 6 may be realized directly by the ALDiSP function shown in fig. 7.

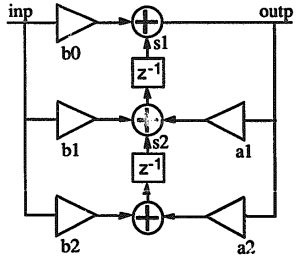

Fig. 6: IIR filter network

```
func IIR (b0,b1,b2,a1,a2: number) (inp: stream) =
let
      outp = delay (b0*inp + s1)
      s1 = 0 :: (b1*inp + a1*outp + s2)
      s2 = 0 :: (b2*inp + a2*outp)
in
      outp
endlet
```

Fig. 7: Function implementing the Network of Fig. 6

Here, the stream outp results from the addition of the streams inp (each element weighted by b0) and s1. Stream s2 is the concatenation of the element "0" and the weighted sum of all the other streams. If necessary, delaying functions may be defined (recursively) that delay streams by an arbitrary number of elements. Note that the IIR fil-

ter function will filter integer values of any type and real values as well because the operators "+", "*" and "::" are defined for all of these data types. A shorter program implementing this filter is hardly conceivable!

Asynchronous communication and I/O

Demand-driven (output-driven) streams are adequate in synchronous applications where the permanent availability of stream elements can be guaranteed. However, this concept of communication is not suitable in situations where it cannot be specified when (if ever) the next data item becomes available (asynchronous case). With streams it would be possible to continuously generate "not available" elements if there is no item but this would be extremely inefficient; it would in fact be like constructing a "busy-waiting" loop in imperative programming languages. To cope with this problem we have introduced the concept of pipes into ALDiSP. Pipes are syntactically similar to streams, but instead of being demand-driven they are data-driven. A pipe can be viewed as a buffer accepting and storing data elements as they become available over time. A consumer may make use of data held in pipes in exactly the same way as with streams; by reading from the pipe, the consumer successively empties the buffer. If there is no data available, the consumer is suspended. On pipes a predicate isAvailable is defined, which becomes true once there is at least one element in the pipe.

In ALDiSP, synchronicity is considered a special case of asynchronicity. Since ALDiSP is an applicative language, values cannot "change". Once an object is created it cannot be destroyed any more. Therefore, all I/O is handled through (virtually) infinite data structures instead of program state. Hence, the distinction between synchronicity and asynchronicity is only important when it comes to input/output. There are two kinds of I/O-devices: *ports* and *registers*. An object of type register maps to a hardware register or memory location, i.e. it can be read or written to at any time. Since its value can change at any time, the exact timing of register accesses is important. *Ports*, on the other hand, are asynchronous devices: They may generate and accept tokens at arbitrary points in time, as soon as they become available. A process trying to access a port that has no data available is *suspended*. It is re-activated automatically once data become available.

Time management and the suspension construct

In ALDiSP there is only a single language construct covering both synchronous and asynchronous timing of actions, the *suspension* construct :

```
suspend expr1 until expr2 within time1, time2
```

When a suspension is evaluated, i.e. called by a function, an anonymous process is created. This process is hibernated upon its creation. It remains dormant until expr2 becomes true. Once this happens, the process is guaranteed to be activated *after* the period of time specified by time1 is over and *before* the time specified by time2 has elapsed. After activation, the process evaluates expr1 and then terminates. A simple example illustrates the usage of this construct:

```
suspend OpenValve() until OverPressure within 0 ms, 0.5 ms
```

Here, a safety valve controlled by an interrupt OverPressure opens no later than 0.5 milliseconds after an overpressure is signaled. Using suspend, macros covering all other relevant cases of scheduling are readily composed:

```
\ ---- Synchronous delay
     expr after time ≡ suspend expr until true within time,
     time

\ ---- Asynchronous, action is taken immediately
     expr1 when expr2 ≡ suspend expr1 until expr2 within
     0 sec, 0 sec

\ ---- Asynchronous with timeout (SystemClock() is a built-in
     function returning the current time)
     when (expr1, expr2, timeoutPeriod) ≡
     !at
       StartTime = SystemClock()
     in
       suspend expr1
       until      (expr2 or SystemClock() - StartTime >
                  timeoutPeriod)
       within 0 sec, 0 sec
     endlet

\ ---- Synchronous, even when duration of action varies
     expr equidistant duration ≡
     let
       StartTime = SystemClock()
     in
       proc tmp() =
       suspend seq expr; tmp() endseq
       until (SystemClock() - StartTime) mod duration == 0
       within 0 ms, 0 ms
     endlet
```

The times may be dynamic; however, if they are known at compile-time, a static schedule may be produced. Pipes are implemented using suspensions: The last available element of a pipe is a suspension waiting for more input. Thus, when a pipe is accessed, the accessing process is suspended until data become available.

Using suspend and the isAvailable predicate, processing of asynchronous pipes is very elegant. The ubiquitous merging of two pipes (where any

of the two may or may not have data available) takes on the following form:

```
proc merge (p1, p2) =
suspend
      if isAvailable(p1) then
        head(p1) :: merge(p2,p1)
      else
        head(p2) :: merge(p1,p2)
      endif
until
      isAvailable(p1) or isAvailable(p2)
within 0sec, 0sec
```

This is an easy-to-understand counterpart of the ALT-construct used in Occam [11] for processing asynchronous data flowing through channels at different rates. Using this construct, interrupt handlers responding asynchronously to a condition becoming true are also written easily.

The ALDiSP Type System

The type system of ALDiSP is based on predicates: An arbitrary set of values may constitute a type. For example, a type comprising all multiples of 3 may be defined as follows:

```
func mult3p(x) = If isInt (x) then x mod 3 == 0 else false endif;
type mult3 = mult3p
```

Mult3p is a predicate testing whether its argument is a multiple of 3; mult3 is a type defined by this predicate.

In most cases, however, such complex definitions are unnecessary. A rich set of frequently used types has been defined, many of which are parameterized. Type classes are:

- Base types, necessary to construct a complete type system
- Atomic types: Unstructured objects and numeric types
- Arrays: n-dimensional collections of objects of the same type
- User-defined abstract types (Records, variant records)
- Machine types: Registers, Ports, Interrupts

Examples of atomic types are parameterizable numeric types: nBitInteger (n), nBitCardinal (n), FixInt (n,m), ShortReal, Real, Longreal, etc. All commonly used operations are available. For one-dimensional arrays (vectors), a great number of functions has been defined, some examples are listed below:

- Selective Update:
  UpdVector ([1,2,3], 1, 5) → [1,5,3]
- Reduction:
  Reduce ("-", [1,2,3,4]) → ((1 - 2) - 3) - 4
- Create Subvector:
  SubVector ([1,2,3], 0, 2) → [1,2]
- Compose Vector:
  CmpsVector ([1,2], [3]) → [1,2,3]

Because of the automatic mapping facility, all functions defined on atomic types can be applied to the contents of arrays, too:

- [1,2,3] + 10 → [11,12,13]

Similar functions exist for two dimensional arrays (matrices), moreover functions are predefined that select rows or columns of matrices, extract diagonals, and reduce matrices.

Exception Handling

Expressions or functions may be guarded for errors that may occur during their evaluation. If an error occurs, a predefined or user-defined exception-handler (a special kind of function) is called automatically. In general, a guarded expression looks as follows:

```
guard expr
on    ExceptionDefinition
.
.
on    ExceptionDefinition
endguard
```

When an error occurs during the evaluation of expr, an exception is called. After the evaluation of the exception the evaluation of expr is not continued; instead, the value defined by the exception-handler is returned as the result of the entire guarded expression. Continuation within the expression is sometimes desirable and therefore possible, too. Examples:

```
\ ---- DivisionByZero is a predefined exception name

func reciprocalPlus3Vers1 (x) =
      guard (1/x) + 3
      on DivisionByZero () = 42
      endguard

func reciprocalPlus3Vers2 (x) =
      guard (1/x) + 3
      on DivisionByZero () = continue(42)
      endguard
```

In the first example, the result of the function when called with 0 is 42, as defined in the exception handler; in the second example the expression causing the error will continue to be evaluated, resulting in a return value of 45.

Exceptions are the basis for actions to be taken on overflows: Whenever an overflow occurs, a predefined exception handler is called. Overflow-reactions are user-definable; the most commonly

used are predefined: Signaling, Ignoring, Realing, Complexing, Saturated, Wrapping. Exceptions are also used to define rounding modes with real numbers. Predefined rounding modes are RoundToZero, RoundToPlus, RoundToMinus, RoundToEven, RoundToOdd.

3 CONCLUSIONS

We have presented a modern applicative language featuring automatic mapping of functions on infinite and finite data, overloadable operators for complex data structures, modularization and time management. Both synchronous and asynchronous data streams of different rates may be defined, which makes interprocess communication in multi-rate-systems simple. Direct machine access and hardware-interrupt handling are possible. An extremely flexible type system, an elaborate exception handling mechanism and a large number of predefined functions operating on structured data as a whole make the language suitable for complex DSP applications. Using this language, programs may be written at very high levels of abstraction, in many cases equivalent to the mathematical description of the problem. Expressing algorithms in an applicative language directs much of the work normally done by the programmer to the compiler. As a consequence, transforming applicative programs into code running on standard processor architectures is a much more complicated task than compiling an imperative program. On the other hand, parallelizing high-level applicative programs is potentially easier than trying to re-parallelize imperative programs written in von-Neumann languages. As compiler technology advances, the difference in execution time between the languages will diminish but the advantage of the compiler being able to match and optimize applicative specifications to parallel architectures will remain.

A simulation program realizing most of the features of ALDiSP has been developed, which is rather slow. Current work centers on transformation techniques to stepwise simplify ALDiSP programs to reduce the run-time requirements for ALDiSP programs. The long-term goal is to generate optimized code directly executable on digital signal processors.

4 REFERENCES

[1] P. Caspi, N. Halbwachs, D. Pilaud, J.A. Plaice
Lustre, a Declarative Language for Programming Synchronous Systems
Proc. 14th ACM Symposium on Principles of Programming languages, Munich, 1987

[2] A. Benveniste, P. Bournai, T. Gautier, P. LeGuernic
Signal: A Dataflow Oriented language for Signal Processing
INRIA, Rapport No. 378, March 1985

[3] European Development Center
Silage Compiler Reference Manual
Brussels, 1989

[4] E.A. Lee
Programmable DSP Architectures: Part II
IEEE-ASSP Magazine, Vol. 6, No. 1, Jan. 89

[5] A. Knoll, R. Nieberle
CADiSP - A Graphical Compiler for the Programming of DSP in a completely symbolic way
Proc. IEEE-ICASSP 90, Albuquerque, 1990

[6] M. Freericks, A. Knoll
AI.DiSP - Eine applikative Programmiersprache für Anwendungen in der digitalen Signalverarbeitung
Technical Report, Techn. Univ. Berlin, FB 20 No. 90-9

[7] Rees, J. et al.
Revised^3 Report on the Algorithmic Language Scheme
SIGPLAN Notices Vol. 21, No. 12, Dec. 1986

[8] Milner, R.
A Proposal for Standard ML
1984 ACM Symp. on Lisp and Functional Programming

[9] MacQueen, D.
Modules for Standard ML
1984 ACM Symp. on Lisp and Functional Programming

[10] P. Pepper, G. Egger
The Opal Project
Internal Memo, TU Berlin, 1987

[11] INMOS Ltd.
Occam 2 Reference Manual
Prentice Hall, 1988