SIGNAL PROCESSING USING MHDL

Mark Kahrs

Dept. of Electrical and Computer Engineering
Rutgers University, P.O. Box 909
Piscataway, NJ 08855-0909
kahrs@ece.rutgers.edu

ABSTRACT

MHDL is a new language created for the behavioral description of state-of-the-art microwave hardware. A microwave communication system includes both continuous time and discrete time processing subsystems. MHDL includes a number of novel features; some of these features are of direct use in the specification of both continuous time and discrete time signal processing. MHDL includes data types `complex` and `z` that find direct use in signal processing applications. This report describes the use of MHDL in the description of signal processing algorithms.

1. INTRODUCTION

MHDL (the MIMIC Hardware Description Language) is a new language created for the behavioral description of state-of-the-art microwave hardware. A microwave communication system includes both continuous time and discrete time processing subsystems. MHDL includes a number of novel features that will be described shortly; several of these features are of direct use in the specification of both continuous time and discrete time signal processing.

Initially, an examination of the difference between a description language and an algorithmic language will be detailed. Next, a brief look at some past DSP languages will be included. In the subsequent section, we examine the specification of continuous time processing and then discrete time processing. Next, we consider how to specify a mixed-mode system, such as a A/D and D/A converter. Finally, we conclude with a report on the current status of MHDL and information about available systems.

1.1. Description vs. Algorithmic languages

Description languages describe *behavior*, not algorithms. In other words, a description language says what the function of the circuit should be and *not* how to perform or calculate the function. MHDL is just such a language. It is meant to describe the behavior of various subsystems in a microwave circuit but not the actual detailed simulation of each subsystem.

This work was partially supported the ARPA tri-service MHDL effort, monitored by ARL DAAL01-93-K-3370.

1.2. Languages of the past

Since the latter 1960s, a number of languages have been designed for DSP applications. BLODI [1] was perhaps the first block diagram language to be created. CLASP [2], another block diagram language, was created for use in a very high level silicon compiler. In the recent past, Kopec's SRL [3] (based on his thesis [4]) used the Lisp Machine environment for the creation of streams and array operators as part of a larger system (ISP [5]). More recently, a group at INRIA used the notion of a centralized clock as a theoretical underpinning for a DSP language called "SIGNAL" [6][7]. Silage [8][9] is another language developed for silicon compilation of DSP algorithms. It is an applicative language with a built in delay oprator. It has avoids explicit state variables by eliminating variable assignments (Silage has equations but not assignments).

Block diagram languages are often too high level; sometimes one wants to specify algorithms, not blocks (one can specify an algorithm inside a block but that defeats the purpose). Often a language offers a single formalism to force the user into using one particular model of DSP computation. This may not be what the user wants, for example, if the user's application doesn't fit easily within the model (adaptive filters often present problems because of feedback; the language may or may not make feedback easy to specify). All of the DSP languages mentioned lack the ability to specify the higher level design constraints this is where a description language really counts. For example, one would like to make a high level specification of a filter and have the rest of the parameters of the filter calculated automatically.

0-7803-2431-5/95 $4.00

1.3. MHDL features

MHDL was designed to specify the behavior of large microwave communication systems; in order to specify all the levels in such a system, the language was designed to be extremely flexible and extensible. Many features of MHDL were borrowed from the new programming language Haskell [10]. Haskell has been used experimentally for the description of DSP algorithms [11].

The first method of extending MHDL is the type system. MHDL types are *polymorphic*, that is, a function can accept more than one type for an argument. Type usage in MHDL is statically checked (i.e., at compile-time). The type system was derived Haskell's "type classes". Type classes are an extension of the normal type system and resembles the object programming notion of method declaration. The declaration of the `Complex` type class is shown below:

```
class Complex (a, b) {
  mkPolar     :: b -> b -> a;
  mkRectang   :: b -> b -> a;
  realPart    :: a -> b;
  imagPart    :: a -> b;
  cmag        :: a -> b;
  cphase      :: a -> b;
} where { Real (a), Real (b) } ;
```

This declares the class of complex numbers (with a real and imaginary part) and the associated functions (`a -> b` denotes a function with single argument `a` returning a value `b`. In this case, both the input and output values are real. The `Complex` type class can be used by an `instance`, for example, to form a rectange from complex points.

The MHDL standard library includes the standard types `Int` and `Float`. It also includes `Complex` (a tuple of `Reals`, shown above) in Cartesian space and also `Polar`, the polar coordinate representation. It can also specify a `Ratio`, i.e., rational numbers as well as signals. Signals have two domains: continuous signals in `f`, with continuous Laplace and Fourier transforms and also discrete signals in `f`, with Z-transforms. Periodic signals can be either continuous or discrete and use the discrete Fourier Transform (via the FFT). MHDL also features physical types, i.e., types with measurable units (these are denoted within single quotes). A prime example is frequency, which is declared

```
dimension frequency = 'time^(-1)';
unit hertz of frequency = 'second^(-1)'
  variations Hz;
```

where `frequency` is a dimension of time, as shown in the declaration above. Conversion is done automatically between compatible units, as in degrees and radians:

```
unit radian of plane_angle
  variations rad, rads, radians;
unit degree of plane_angle conversion
  floatApproxRational((pi :: double) /
    180, 1.0E-10) radian
  variations degrees, deg;
```

MHDL is a 'first class' language, so functions can be values and returned from functions just like any other value. MHDL also includes "pattern matching" as a fundamental part of the language. Pattern matching is used in numerous ways including searching data structures, doing `case` selection and iteration. Constraints are also available as a mechanism to verify an assertion about any expression; constraint failure is reported to the user immediately. MHDL has many other features but we shall concentrate on the features most useful in the specification of signal processing.

2. CONTINUOUS TIME PROCESSING

Consider a simple low pass filter:

```
components
  LP :: lowpass_filter
  definitions
    f_pass = 700 'Hz';
    f_stop = 1000 'Hz';
    ripple = 0.1 'dB';
  end LP;
end components;
```

Here, the definitions serve as method for specifying the behavior but *not* the implementation of the filter. In particular, the filter could be implemented using in either continuous *or* discrete time. The next step is to refine the implementation of the filter. For example, we could specify the behavior as follows:

```
structure frequency_response
  configuration typical
    for lowpass_filter use
      if (ripple > 1 'dB') then
        Chebychev
        else Butterworth;
  end typical;
end frequency_response;
```

Here, the specification calls for the system to use a Chebychev filter if the ripple parameter is greater than 1 dB otherwise, use a Butterworth. The definition of the filter will be done inside another model:

```
model Butterworth
  structure continuous
    cmag(H(s)) = 1/(1 + (s/s_c)^(2 * N));
    N = ceiling(log(10^(A_stop/10 -1)/
        10^A_pass/10 -1) /
```

```
        2 * log (f_stop / f_pass));
  end continuous;
end Butterworth;
```

First, the use of = denotes an equation, not an assignment. The binding between the parameter names inside the filter model (s, s_c, N and the amplitude and frequency parameters) with the specification in the model is done explicitly by the MHDL interpreter.

The expression of the Butterworth polynomial can follow any number of paths: first, one can specify the polynomial in equational form as a function of s, i.e.,

```
H(s) = K / ((s + a[0])^2*(s + a[1])^2)
```

Another possibility is to express the polynomial as a function of poles and zeros, i.e., as a function that takes two tuples:

```
H(s) = K * Zeros([])/
  Poles([(2.0*PI*440, -0.65*J),
         (2.0*PI*-400, -0.80*J)])
```

Here, `Poles` and `Zeroz` are functions that accept a list of complex points and form products $\prod_i^M (s + s_i)$ where M is the length of the list and each complex point is s_i. MHDL also includes a number of continuous time operators, e.g., integration and differentiation. But these operators are just for specification; they do not say *how* the operator is simulated. Using such an operator, the specification of a sample and hold is easy:

```
output.voltage =
  integral(t)(t, t+1/Fs) C;
```

3. DISCRETE TIME PROCESSING

MHDL also includes the complex z domain. So, an MHDL user can specify a digital filter by just giving its polynomial in z. Of course, continuous time filters in s and discrete time filters in z can *not* be directly interconnected because MHDL will find a type error when the discrete time and continuous time variables interact. Below is a digital second order section

```
model Butterworth
  structure discrete
    H(z) = Hi(z, N) where
      Hi(z, j) =
        if (j == 0) then 1 else
         Hi(z, j-1) *
         ((G[j] * (1 + 1/z)^2) /
        (1 +
          a[1,j]*1/z +
          a[2,j]*1/z*1/z));
  end discrete;
end Butterworth;
```

Here, the IIR nature of the filter is exposed by the step down recursion. Note the use of the different structure name to differentiate its behavior from the continuous version shown above (but the parameter N is the same).

MHDL features "lazy evaluation", i.e., functions are not called until needed. This mechanism can be used to create *streams* a la SRL. The key relevant feature of laziness is the ability to construct infinite sequence of samples and *not* evaluate the sequence until each sample is needed. Interpolation and decimation can be easily implemented using lazy evaluation: `interpolation(S, N)` takes a discrete signal S and inserts an $N-1$ zeros; `decimation(S, N)` removes $N-1$ samples by either advancing the time pointer in the sequence or by truncating the list. What's important to realize is that because of the lazy evaluation, the entire vector does *not* need to be computed; rather, the insertion and deletion of samples happens *by need.*

4. MIXED MODE

Push comes to shove at the interface of continuous and discrete time systems. Here, any system specification language must be able to interconnect both parts (and simultaneously, any simulator must be able to simulate both parts). Because of its flexible type structure, MHDL can represent both parts and furthermore prohibits explicit interconnection of continuous and discrete time components. How then, can one specify their interconnection? The answer is *coercion.* Consider the analog to digital converter: we can break this down into two parts, sampling and quantization.

```
model analog_digital_converter
  structure topology
    connections input, output;
    output.out =
      Quantizer(Sampler(input));
  end topology;
end analog_digital_converter;
```

Here, `Sampler` is a function from a continuous `input` and continuous time that returns a continuous voltage and a `discrete` time. The `Quantizer` function converts this to a `discrete` voltage value. Note that the quantized output can be made into a new, separate type (i.e. an instance of the `Ratio` type. This will insure that use of quantized signals from different quantizers can not be confused. However, each of these types must share the same type class so they can be used in the same functions, i.e., so they can all use the same digital filters.

Here, we demonstrate the use of coercion in a DAC to coerce a discrete time input to a physical value.

```
coercion ToV :: DiscTimeDim (a) -> Phys (a);
DiscTimeDim (p) = \(t) -> p;
```

```
model digital_analog_converter
  structure electrical
    connections input, output;
    output.voltage = Filter(Hold(
      ToV(input.in)));
    H(f) = 1/f_sample *
        exp(-j * PI * f_d) *
        (sin(PI * f_d) / PI * f_d)
          where { f_d = f/f_s };
  end electrical;
end digital_analog_converter;
```

`ToV` is the coercion function; its type declaration is declared first, the function definition of the conversion function `DiscTimeDim` follows. `DiscTimeDim` converts the discrete time signal input into a continuous physical output. `Hold` is the output sample and hold and `Filter` is the anti-aliasing filter whose response is described by `H(f)`.

5. CONCLUSION

MHDL is a different kind of Hardware Description Language because of its flexibility. This malleability is due to its type definitions and facilities for specifying hierarchical systems. MHDL can be used for the specification of both continuous and sampled signal processing systems. Furthermore, it can also specify mixed systems, such as A/D and D/A converters, so that signal processing systems can be specified from continuous input to discrete time processing to continuous output.

MHDL is on its way to becoming a new IEEE Standard through the SCC-30 committee. A fully public domain implementation is available and is distributed without fee.

6. REFERENCES

[1] B. J. Karafin. A new block diagram compiler for simulation of sampled-data systems. In *Fall Joint Computer Conference*, pages 55–61. AFIPS, 1965.

[2] M. Kahrs. Silicon compilation of a very high level signal processing specification language. In P. R. Cappello, editor, *VLSI Signal Processing*, pages 228–238. IEEE Press, 1984.

[3] G. E. Kopec. The signal representation language SRL. In *IEEE Intl. Conference on Acoustics, Speech and Signal Processing*, pages 1168–1171, 1983.

[4] G. E. Kopec. *The representation of Discrete-time signals and systems in programs.* PhD thesis, MIT, MAY 1980.

[5] G. E. Kopec. The integrated signal processing system ISP. Technical Report Rapport No. 644, Fairchild Laboratory for AI Research, June 1983.

[6] T. Gautier P. Le Guernic, A. Nenveniste. Signal: Un language pure le traitement du signal. Technical Report Rapport No. 206, INRIA, May 1983.

[7] M. Le Borgne P. Le Guernic, T. Gautier and C. Le Maire. Programming real-time applications with signal. *Proc. IEEE*, 79:1321–1336, 1991.

[8] Paul Hilfinger. A high-level language and silicon compiler for DSP. In *Proc. Custom Integrated Circuits Conference*, pages 213–216, May 1985.

[9] D. Genin, P. Hilfinger, J. Rabaey, C. Sheers, and and H. DeMan. DSP specification using the Silage language. In *IEEE Intl. Conference on Acoustics, Speech and Signal Processing*, pages 1057–1060, April 1990.

[10] P. Wadler P. Hudak, S. L. Payton-Jones. Report on the programming language Haskell, a non-strict purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27, May 1992.

[11] David M. Goblirsch. Digital Signal Processing in Haskell. In Philip A. Wilsey and David Rhodes, editors, *Intl. Conference on Simulation and Hardware Description Languages*, pages 17–22, January 1994.