A Functional Specification Notation for Co-Design of Mixed Analog-Digital Systems

Alex Doboli
State University of New York at Stony Brook
Electrical and Computer Engineering Department
Stony Brook, NY, 11794-2350, USA
adoboli@ece.sunysb.edu

Ranga Vemuri
University of Cincinnati
ECECS Department
Cincinnati, OH, 45221-0030, USA
Ranga.Vemuri@uc.edu

Abstract

This paper discusses aBlox - a specification notation for high-level synthesis of mixed-signal systems. aBlox addresses three important aspects of mixed-signal system specification: (1) description of functionality and (2) performance issues and (3) expression of analog-digital interactions. The semantics of aBlox embeds concepts and rules of a **functional computational model**, *and uses a* declarative *style to denote performance elements. The paper shows some mixed-signal specifications that we developed in aBlox. Finally, we describe a high-level analog synthesis experiment that used aBlox specifications as inputs.*

1. Introduction

A large variety of modern applications are based on mixed analog-digital circuits that are fabricated on the same silicon chip as *systems on chip* (SoC) [13]. Such circuits provide information processing and communications capabilities to consumer electronics, industrial automation, retail automation and medical markets. For example, analog circuits interface a mixed-signal system to the external world by sensing, receiving, amplifying and filtering continuous-time analog signals. Typical analog circuits include low-noise amplifiers, filters, mixers, oscillators etc. Then, analog-digital converters transform, through a digitization process, the analog signals into digital signals so that signal-to-noise and distortion ratios are kept within acceptable ranges. Finally, discrete-time digital signals are processed using general-purpose processors, DSP-s, ASIC-s, FPGA-s etc. Hence, because of their inherent complexity, *high-level design automation methodologies* and *techniques* are highly demanded for managing the design cycle of mixed-signal systems [13].

High-level mixed-signal synthesis (HMS) takes as input an abstract high-level specification of the system to be designed, and automatically produces analog and digital hardware that optimizes a large set of constraints i.e. area, power, speed, precision etc [7] [8] [13]. Successive tasks are performed during HMS including (1) analog-digital partitioning, (2) architecture generation, (3) performance model generation and (4) parameter optimization [8] [13]. High-level specifications describe the behavior (functionality) of a mixed-signal system without distinguishing the analog and digital parts or providing any hardware details. Donnay *et al* [9], among others, motivates that there are no well established specification languages for HMS even though specification is compulsory for any HMS methodology. Existing languages i.e. VHDL-AMS [2], Verilog-A [1], MAST [10] etc are all simulation oriented. There are difficulties in adapting their semantics for automated synthesis [6]. This strongly motivates research on synthesis-oriented specification for mixed analog-digital systems.

This paper presents a *specification notation* for co-design of mixed analog-digital systems. The notation, called aBlox, follows a *functional descriptions style* [12] for expressing mixed-signal systems. aBlox constructs address three aspects: (1) description of functionality and (2) performance, and (3) expression of analog-digital interactions. Following concrete aBlox elements target these elements:

1. aBlox specification "philosophy" is to explicitly describe signal processing and flows in the analog and digital domains. This is important for having *similar* description styles for the two domains so that analog-digital trade-offs i.e. analog-digital partitioning can be easily explored in the methodology. The challenge is to define uniform and sound models and specification notation for analog, digital, continuous and discrete-time signals and computations, and their interactions.
2. aBlox constructs encourage a hierarchical and modular description of system through higher-order functions

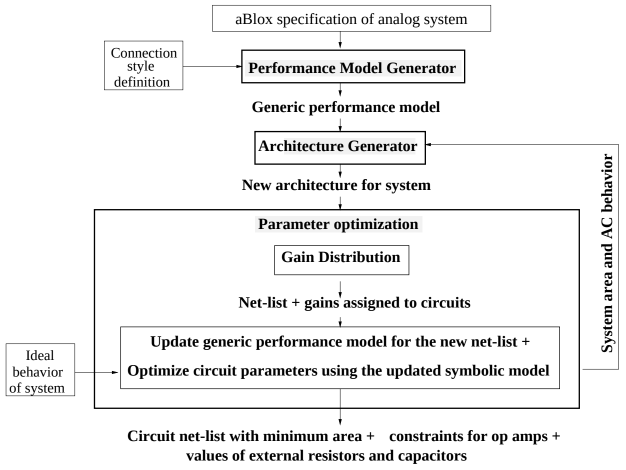
aBlox specification of analog system
Connection style definition
Performance Model Generator
Generic performance model
Architecture Generator
New architecture for system
Parameter optimization
Gain Distribution
Net-list + gains assigned to circuits
Update generic performance model for the new net-list +
Optimize circuit parameters using the updated symbolic model
Ideal behavior of system
System area and AC behavior
Circuit net-list with minimum area + constraints for op amps + values of external resistors and capacitors

Figure 1. **High-level mixed-signal synthesis**

(HOF) [12]. HOF-s express structural patterns in a system. Patterns enforce a certain interconnection structure between modules without specifying their nature. Concrete building blocks are passed as parameters for every HOF call. HOF-s are very useful as system hierarchy and regularity can be effectively exploited in improving synthesis quality [7] [8].

3. aBlox provides a coherent notation for linking performance constraints and models to the constituting modules of a program. As a result, a large variety of performances such as speed, area, DC, AC and transient behavior, noise coupling, power consumptions etc can be expressed in a uniform manner without introducing dedicated keywords for each of them. This is important as components of a mixed-signal system have heterogeneous performances requirements. For example, the analog part of a telephone set includes a receiver and a transmitter with different noise constraints [7].
4. aBlox notation offers a well defined mechanism for interfacing analog and digital domains.

aBlox notation was successfully used for specifying various analog applications including telecommunication systems, filters, and A/D converters. To motivate the usefulness of aBlox, this paper discusses a high-level synthesis example that used aBlox specifications as inputs.

This paper has the following structure. Section 2 offers insight on mixed-signal synthesis and enumerates requirements for system specification at a high level. Section 3 discusses the main aBlox constructs. Section 4 concentrates on performance model description in aBlox. Section 5 shows a synthesis experiment, and finally, conclusions are provided.

2. Synthesis and Specification of Mixed-Signal Systems

Figure 1 depicts the mixed-signal synthesis flow that we target [7] [8]:

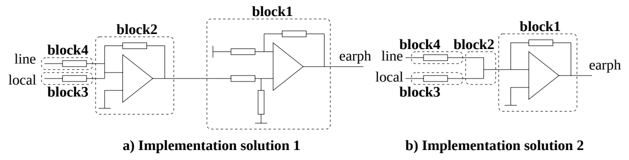

Figure 2. **Architectures for telephone receiver**

- The **Performance Model Generator** produces *generic* mathematical expressions that formulate how system performance parameters depend on the parameters of the blocks in the system. Then, symbolic performance models for all architectures generated during synthesis result by updating this generic model.
- The **Architecture Generator** creates different implementations for a specification. Specification functionality can be achieved by interconnecting basic building blocks i.e. op amps, resistors, capacitors, and not necessarily only library circuits i.e. adders, integrators etc. Figures 2(a) and 2(b) illustrate two distinct architectures for a telephone receiver module [7].
- Area, AC and transient behavior of each architecture are determined by the **parameter optimization** module, and used to guide the Architecture Generator. It finds sizes for external resistors and capacitors and bounds for op amp parameters i.e. input and output impedance, gain and dominant pole so that total area is minimized and the resulting AC and transient behaviors of a system are within an error margin from the desired behaviors. To guarantee feasible solutions, each free parameter was modeled by a feasibility range for CMOS technology [14] i.e. external resistors are in range $[1, 100]$kΩ, op amp gains in range $[10^3, 10^4]$ etc.

Different specification styles can be used for describing mixed-signal systems i.e. declarative style, imperative style, functional style, object-oriented style etc [12]. Each of the styles can be useful for different synthesis tasks. For example, declarative specifications are popular for describing performance constraints and models for transistor sizing of analog circuits [18]. A declarative specification expresses relationships and constraints among signals or circuit performances i.e. voltages, current, unity-gain-frequency, slew-rate, etc. A declarative specification shows *what* a system does and not *how* it achieves its functionality. Thus, this style does not provide any insight into producing structural implementations (hardware architectures) for the system.

We believe that functional specifications at the *Signal Flow Graph* (SFG) level [17] permit to systematically synthesize optimized mixed-signal implementations. SFGs indicate the system behavior by showing the signal processing and flow. Similar approaches of SFGs specification notations are proposed by Kopec [15] and Lee *et al* [16] for synthesis of digital DSP systems. We present next our concrete arguments for adopting a functional specification style at the level of SFGs:

- SFG-s are similar to algorithmic descriptions for digital synthesis as they explicitly capture signal flow (dependencies) and processing (operations). Keeping similarity between analog and digital specifications is important for re-targetable mixed-signal synthesis.
- The effectiveness of analog synthesis dramatically depends on describing lower-level attributes i.e. frequency, speed, noise. SFG-s are a convenient abstraction-level for linking such attributes to the language constructs of a specification.
- Effective synthesis algorithms can be formulated for SFG-s [7]. SFG blocks suggest the structure of a system. As they represent operations i.e. amplification, integration, summation etc, SFG blocks are easily mappable to the corresponding electronic circuits.

A final requirement for mixed-signal synthesis is that the specification language clearly distinguishes functional from performance elements. Otherwise, erroneous situations occur where synthesized systems emulate the performance aspects i.e. a system that calculates the slew-rate even though slew rate is a performance constraint.

3. aBlox Notation for Synthesis

3.1 Macro Definitions

An aBlox program describes interacting analog and digital domains. These domains can have a hierarchical structure, if a domain is built of sub-systems, stages, components, etc. The *macro* construct is the main notation feature for describing functionality, hierarchy and interfaces. Figure 3 illustrates samples of macro definitions for a two stage 4-th order filter. The figure suggests how macro definitions and macro calls are employed for expressing the hierarchy in a system. A mixed-signal specification must contain a top-most macro (the macro that is not called by other macros). The top-most macro is executed forever.

A macro definition includes following five elements:

- *Domain descriptor* that indicates the domain of the macro. It can be *continuous_time, digital* or none if the domain is not fixed yet. In the latter case, finding the macro domain is subject to analog-digital partitioning.
- *Input* and *output ports* define the interface of a macro with the rest of the specification or with the external environment. Ports of the top-most macro are system ports with the external environment.
- *Generic parameters* are used for indicating the generic elements of a macro i.e. constant values, operators, block identities and performance attributes. Each macro call instantiates concrete values for the generics. Generic parameters are useful for expressing uniformity and hierarchy of macro structures. Linear operators i.e. addition, integration, etc. can also be generics for a macro. The two filter stages in Figure 3 are characterized by different filter constants that are specified as generics in the program. Operators are passed as arguments to macro calls for describing stages built of distinct blocks connected in similar patterns.

```
macro stage
  inputs
    i1;
  outputs
    out;
  generics
    constants a1, a2;
  arch controlable is
    variables
      m, n, p;
      o is array[2];
     o[1] = i1 + m;
     o[2] = a2 * p;
     n = + o;
     p = integ(n);
     m = a1 * integ (p);
    out = integ(p);
  end arch;
end macro;
```

```
macro filter is continuous_time
  inputs
    i is voltage;
  outputs
    o is voltage;
  arch two_stage_filter is
    variables
      v;
      v = stage.controlable(
            i, generics are
               1.7251, -1.9374);
      o = stage.controlable(
            v, generics are
               1.7251, -1.9374);
  end arch;
end macro;
```

Figure 3. **aBlox specification of a filter**

- *Attribute section* is allowed only for macro-s of the continuous-time analog domain. It introduces declarative or equational performance models that are associated with a macro, and then used for parameter optimization during synthesis.
- *Macro body* expresses the functionality described as a set of statements i.e. assignment statements, if statements, macro-calls and that refer to input ports, output ports and local variables.

Semantic rules for domain definitions

Rule 1: Each call to a *continuous_time* macro defines a distinct macro structure having as inputs and outputs the variables referred by the call. If more macro calls or operators take the same variables as inputs then a single structure is generated but its output is linked to all referring places.

This rule is natural as no macro sharing is feasible for distinct signals in a continuous-time analog system.

Rule 2: Inside a macro with a *continuous_time* (*digital*) domain descriptor only macros with *continuous_time* (*digital*) or without any domain descriptor can be called. Inside a macro *without a domain* descriptor following cases are correct: (1) only macros with *continuous_time* or without a domain descriptor are called or (2) only macros with *digital* or without a domain descriptor are called. The top most macro can call both *continuous_time* and *digital* macros.

This semantic rule prohibits developing hierarchal specifications where a macro (excepting the top-most macro) can call other macro-s having both time models. Reason is that transforming analog (digital) functionality and performance constraints into requirements for the opposite domain is a very difficult task. This rule is different from simulation-oriented mixed-signal environments i.e. Ptolemy II [4], where hierarchical compositions of distinct domains are possible.

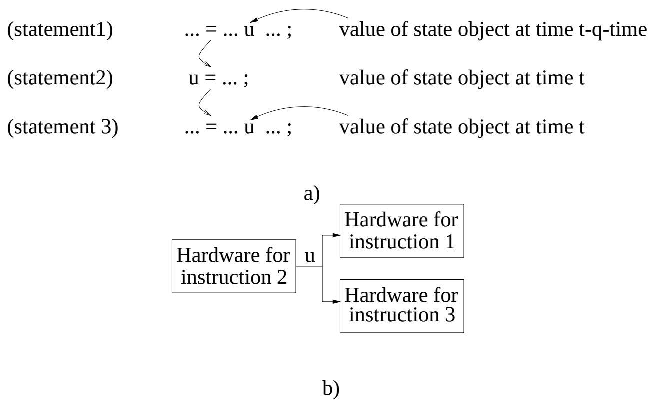
(statement1)
... = ... u ... ;
value of state object at time t-q-time
(statement2)
u = ... ;
value of state object at time t
(statement 3)
... = ... u ... ;
value of state object at time t
a)
Hardware for instruction 2
u
Hardware for instruction 1
Hardware for instruction 3
b)

Figure 4. **Semantics of instruction sequence**

Semantic rule for mapping variables to ports

Rule 3: Ports of the top-most macro are of electrical types such as *voltage*, *current* or *digital* as the interface of the system with its external environment is well defined. Ports can be annotated with attributes i.e. value ranges, input/output impedances, and frequency ranges.

Semantic rule for variable scoping

Rule 4: The scope of variables defined inside a macro is limited to the macro body, only.

This rule guarantees that macro definitions have the meaning of mathematical functions, thus the property of *referential transparency* [12]. This implies that a macro's functionality (meaning) is not influenced by its connections with the other macros.

Semantic rule for describing domain interactions

Rule 5: Interdomain interactions happen through macro-calls, port mappings, and variable/port assignments at the level of the top-most macro. Explicit conversions from *bit* or *bitstring* to *float*, and vice-versa can be performed depending on the variable/port types.

Semantics of data objects, expressions and assignments

Rule 6: All variables of *continuous_time* macros denote memory-less objects. These variables can be of three types: *voltage* - when they only correspond to voltages in implementations, *current* - when they are "realized" as currents, and *unspecified* - when both voltage and current alternatives are acceptable in an implementation.

Rule 7: Variables of *digital* macros are either memory or memory-less objects. Input and output ports of the top-most macro are memory-elements, always. Memory elements are indicated by using the keyword **static** before variable definitions. Digital variables can be of type *bit* or *bitstring*. Bitstring is an array with either static dimension or a dimension is described using generics. However, bitstring dimensions must be computable at compile time.

Rule 8: The two domains have different operators:

- *Continuous_time* macros can include following arithmetic operations: addition, subtraction, multiplication by a constant, and integration. This is a complete operator set for any linear system [17], and it can be implemented with simple electronic circuits [6].
- *Digital macros* might include *arithmetic operators* i.e. addition, subtraction, multiplication, division and *logical operators* i.e. and, or, negation, etc.

Rule 9: An assignment statement is a connection between a name (the left part of an assignment) and the anonymous function definition (λ-expression [12]) introduced by the right part of the statement. All references to assigned objects are actually calls to the lambda expression with the same values for input parameters as in the assignment statement. Memory elements reside only in the top most macro.

Rule 10: Memory-less objects are updated in time zero after their executing the assignment statement.

Rule 11: Update of memory variables happens after executing the last statement of their defining macro.

Unconstrained assignments in a macro violate the referential transparency property as they introduce side-effects [12]. Although side-effects do not pose any problem for digital synthesis, they are difficult to cope with during analog synthesis [6]. Thus, we accept assignment statements in our functional model but we enforce a functional semantics for being consistent with the rest of the mixed-signal model.

Semantic rule for instruction sequence

Rule 12: A variable of a *continuous_time* macro can not be assigned more than once in a sequence of statements.

If a variable were assigned twice or more times in a sequence of statements it means that for a short time same it has more than one value. Continuous-time variables have only a single value in our model (we assumed that each distinct data object has a different name).

Rule 13: Any variable or output port of a *continuous_time* macro that is referred by a statement must appear in the left part of an assignment statement.

Thus, continuous-time objects have a value at any time.

Rule 14: Data dependencies among instructions of *continuous_time* macros describe signal flows among the processing blocks corresponding to the instructions.

For example, consider Figure 4, where object *u* denotes a variable of the analog domain. Figure 4(a) depicts a program fragment, and Figure 4(b) shows how data dependencies among instructions express the corresponding signal flows between the processing macros.

Observation: For a *continuous_time* macro, any sequencing (ordering) of a given set of instructions will produce the same block structure (flow of data is unique).

Semantic rules for if statement

If statements denote a conditional behavior of a system with multiple modes of behavior. For example, a variable-gain block has multiple modes of behavior fixed by its distinct gains. Following are requirements for *if* statements of *continuous_time* macros.

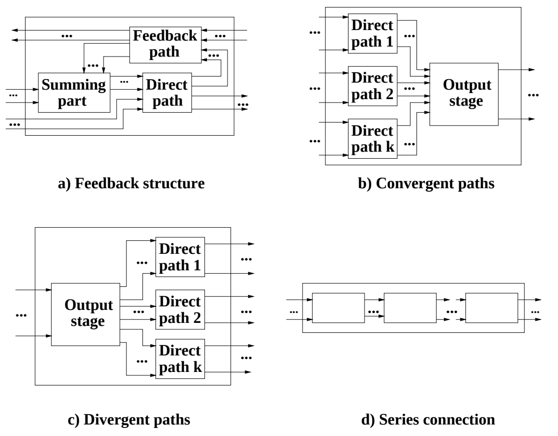

Figure 5. **Block structure**

```
feedback_structure is
  inputs ...
  outputs ...
  summing part is
    ...
  end summing part;
  direct path is
    ...
  end direct path;
  feedback path is
    ...
  end feedback path;
end feedback_structure;
```

a) feedback_structure construct

```
convergent_paths is
  inputs ...
  outputs ...
  direct path 1 is
    ...
  end direct path 1;
  direct path k is
    ...
  end direct path k;
  output stage is
    ...
  end output stage;
end convergent_paths;
```

b) convergent_paths construct

```
divergent_paths is
  inputs ...
  outputs ...
  input stage is
    ...
  end input stage;
  direct path 1 is
    ...
  end direct path 1;
  direct path k is
    ...
  end direct path k;
end divergent_paths;
```

c) divergent_paths construct

```
series_blocks is
  inputs ...
  outputs ...
  ...
  aBlox instructions
    for indicating
   block connections;
  ...
end series_blocks;
```

d) series_blocks construct

Figure 6. **aBlox instructions for block structure**

Rule 15: If a variable of a *continuous_time* macro is assigned by one *if*-branch then it has to be assigned by the other branch, also.

This is a consequence of the life-time rule for analog objects, considered to be permanently a-life. If an object were updated by only one of the branches then, the object will not have a value when the opposite branch is executed.

Rule 16: An analog domain object assigned inside an *if* statement cannot be assigned outside the *if* statement, also.

This is a consequence of Rule 12 for instruction sequencing. The rule also accommodates well a functional specification style where all object assignments are inside the same scope (the scope of the *if* statement in this case).

Rule 17: For the analog domain, conditions of *if* statements refer only to digital input ports of the macro.

There are two reasons for this rule:

- For mixed-signal applications, functioning modes of the analog domain are selected by signals coming from the outside of the continuous-time domain [13].
- To avoid repeated ”switching” of *if* statements at each ”unit” of time. The semantics of digital objects prohibits repeated switching of the same instruction.

3.2 Higher-order functions

System hierarchy and regularity are very important for effective mixed-signal synthesis [7] [8]. Hierarchy and regularity make system specifications simpler and more readable, and they simplify synthesis tasks i.e. performance model generation and parameter optimization. aBlox macro definitions and macro calls can describe hierarchical specifications. For example, the two stage filter in Figure 3 is described hierarchically. Both stages are in controllable form, and they contain blocks with similar kind of functionality.

Similar block structures can occur in a system so that these structures involve different blocks. Such situations introduce structural regularity that can be exploited for synthesis. aBlox notation permits definition of higher-order functions (HOF) to allow full re-use of structural regularities. HOF-s are macro-s that have other macro-s as their generic parameters. Then, structural regularities are expressed as interconnections of generic blocks.

Rule 18: A higher-order function is a macro definition that includes in its *generics* section *signatures* for all generic blocks involved in expressing the HOF body structure. A signature enumerates the number and type of inputs and outputs of a generic block without indicating its functionality. Each call to an HOF replicates the regular block structure of the HOF body, while replacing the generic blocks by the actual parameters of the call.

Special HOF-s called *feedback_structure*, *convergent_paths*, *divergent_paths* and *series_blocks* were introduced to increase the readability of aBlox programs by providing dedicated constructs for very popular block structures. Figure 5 illustrates these block structures. Figure 6 depicts the syntax of the four aBlox constructs. These constructs are not orthogonal as their behavior can be achieved with the already existing aBlox instructions.

4. Description of performance models

In the process of exploring different notations for mixed-signal synthesis, we found that a declarative description style can be very useful in the context of functional specifications. Following reasons motivate our conclusion:

- Macro-s can define structures with heterogeneous design/performance constraints. These constraints can be expressed in a declarative style, and annotated to the macros. In Figure 3, the two stages of the filter can have different noise and bandwidth constraints.
- Application-specific performance models are required for synthesis. An ”ideal” mixed-signal synthesis tool would automatically infer all required performance

models. We already automated linear performance model generation. Nevertheless, there is currently no solution for automated generation of non-linear performance models. To overcome this limitation, aBlox notation permits explicit definition, in a declarative style, of all missing performance models.

We stress that declarations do not express system functionality, thus they are not mapped to hardware. They are thought as performance requirements and models for macro implementations. In our synthesis methodology, they are useful for parameter optimization.

aBlox notation has a flexible mechanism of performance model/ constraints definition based on the principle that a language must offer the possibility of describing new entities based on primitive constructs. This avoids an explosion of dedicated keywords for the many performance elements describing analog systems i.e. rise-time, fall-time, settle-time, slew-rate, sensitivity, unity-gain frequency etc [14].

Rule 19: Declarative descriptions are expressed using four constructs: (1) primitive constructs, (2) predicate definitions, (3) attributes definitions and (4) model definitions. Declarative constructions can be global or local to aBlox macros. Global declarations are defined using an *attribute_package* construct. Then, global declarations are made visible to a macro by *importing* its definitions in the attribute section of the macro. Local declarations are defined using the *attributes* construct. Finally, attributes can refer to generic elements that are instantiated by macro-calls.

For example, it is not necessary to redefine slew-rate for all macros in a specification. Slew-rate can be described in an *attribute_package* section, and then imported in all macros that require slew-rate definitions.

Rule 20: *Primitive constructs* include

1. **Signal characteristics** such as voltage, current, phase and frequency are denoted using the *dot* construct. The subsequent example refers to a dot construct.
2. Following **predicates** are defined for signals: *min* for indicating the minimum value of a signal, *max* for the maximum signal value, *current* for the momentary value of a signal, and *final* for the final signal value (value at time infinite). Using predicate *in*, it can be tested that a signal value pertains to a given range.
3. **Time aspects**: Keywords *StartTime* and *EndTime* denote time moments for start and end of execution. Construct *Time.(event at i)* indicates the time moment when the *i*-th occurrence of event *event* happens. The happening of the event is indicated by predicate *event* being true. Construct *a.voltage(event at i)* denotes the voltage of signal *a* at the *i*-th occurrence of event *event*. Similar constructs exist in aBlox for currents, phase and frequency.
4. **Frequency aspects**: Construct *Frequency.(event at i)* indicates the frequency for the *i*-th occurrence of event *event*. Keyword *DC* denotes a frequency of 0 Hz.

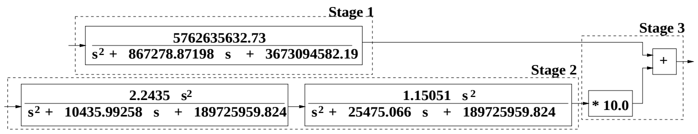
Stage 1
$\frac{5762635632.73}{s^2 + 867278.87198\ s + 3673094582.19}$
Stage 2
$\frac{2.2435\ s^2}{s^2 + 10435.99258\ s + 189725959.824}$
$\frac{1.15051\ s^2}{s^2 + 25475.066\ s + 189725959.824}$
Stage 3
+
* 10.0

Figure 7. **Block structure of optimized filter**

For example, the construct *v.voltage* denotes the voltage facet of signal *v*. The settle-time of a circuit is the time moment for which the value of its output signal stays in a given range. The condition that the voltage facet of signal *a* is within a 2% error margin from its final output value is defined as the aBlox predicate

*a.voltage in [0.98 * final (a.voltage), 1.02 * final (a.voltage)]*.

The 3-db bandwidth of a system is defined in aBlox as

define *Bandwidth = Frequency. ((output.voltage - output.voltage (DC) < 3dB) at 1)*

Rule 21: *Predicates* are formed using (1) arithmetic operators i.e. $+, -, *, /$, (2) relational operators i.e. $<, <=, >, >=, <>, ==$, and (3) the derivative operator *derivate* for indicating sensitivities or rates of change over time i.e. slew-rate. Predicates refer to primitive constructs or attribute definitions.

Rule 22: An *attribute definition* associates a name with a predicate.

An example is the previous definition of attribute *Bandwidth*.

Rule 23: *Model definitions* introduce a set of equations that are simultaneously solved during synthesis for obtaining the performance values.

Model definitions are useful to indicate behavioral performance models. For example, we described in aBlox the behavioral model of a PLL system as indicated by Vassiliou *et al* [19]:

*derivate (phase (Vi.voltage), Time) == 2 * Pi * frequency (Vi.voltage);*
*derivate (Vc.voltage, Time) == 1 / C2 * Ipeff.current - 1/(R*C2) * Vc.voltage + 1/(RC2) * Vx.voltage;*
*derivate (Vx.voltage, Time) == 1/(R * C2) * Vc.voltage - 1/(R * C) * Vx.voltage;*
*derivate (phase (Vj.voltage), Time) == 2 * Pi * nd * (Fo + ko * Vc.voltage);*

Model definitions are simulation oriented. aBlox does not state how model definitions are solved. This concept is similar to *simultaneous* statements [2] in VHDL-AMS.

5. Case Study

This section discusses a case study that arguments the feasibility of aBlox notation for conducting high-level analog synthesis. The considered application is the optimized filter of the Eartalk system [11]. Figure 7 presents the block

```
macro filter is continous_time
    inputs v_in is voltage with range 0-1.0 V;
    outputs v_out is voltage with range 0-1.5 V
        with impedance 280.0 Ohms;
    attributes
        bandwidth is range 10000-80000 Hz;
    arch optimized_filter is
        variables s11, s12, s13, s14, s15, s21, s22, s23,
        s24, s25, s31, s32, s33, s34, s35, v_out1, v_out2,
        v_out3;
        -- Stage3: filter output
        v_out = v_out3 + 10.0 * v_out1 + v_out2;
        v_out1 = s15 + 0.0023413149353 * s12 + 2.2435 * v_in;
        s15 = 0.000425650190865144 * s11;
        s11 = integ (s12);
        s12 = integ (s13);
        s13 = s14 - 0.01043599258 * s12;
        s14 = v_in - 0.000189725959824 * s11;
        -- Stage2: fourth order filter
        v_out2 = s25 + 0.038480341944 * s22 + 1.51051 * v_in;
        s25 = 0.00028658295957375 * s21;
        s21 = integ (s22);
        s22 = integ (s23);
        s23 = s24 - 0.025475066 * s22;
        s24 = v_in - 0.000189725959824 * s21;
        -- Stage 1: second order filter
        v_out3 = 0.00576263563273 * s31;
        s31 = integ (s32);
        s32 = integ (s33);
        s33 = s34 - 0.08627087198 * s32;
        s34 = v_in - 0.00367309458219 * s31;
    end arch;
end macro;
```

Figure 8. **aBlox specification of the optimized filter**

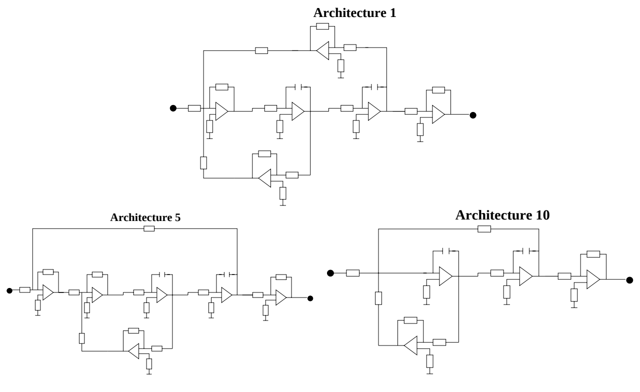

Figure 9. **Architecture samples for filter stages**

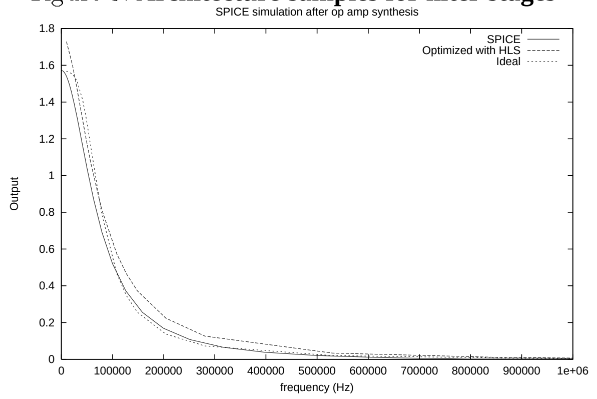

Figure 10. **System and transistor level simulation of** ***Stage1***

structure of the filter that consists of three parts [11]: (1) *Stage 1* is a low frequency band filter for transmitting a portion of the spoken signal, (2) *Stage 2* is a high-frequency band filter for transmitting a second portion of the spoken signal, and (3) *Stage 3* combines the sound signals from the two portions (filters). We discuss the aBlox specification of the filter, and then present our simulation results for the filter design at the system, transistor and layout levels.

Figure 8 shows the aBlox program for the optimized filter. Signal v_in is an input voltage with its values limited to the range 0-1.0 V. Signal v_out is an output voltage with its values limited to the range 0-1.5 V, and driving an impedance of 280 Ohms. The attributes sections defines constraints for the filter: noise should be less than 80dB and filter bandwidth is 10-80 KHz. The macro body expresses the three stages of the filter. Each stage is described as a set of instructions involving addition, subtraction, multiplication with constants and integration operations. This defines the operation kinds of the building blocks in the analog architectures. The signal flow among blocks is a resultant of the data dependencies introduced by the instructions. Note that the instruction sequence did not have any influence on the structure of the signal flow.

Figure 9 depicts three of the 16 distinct architectures, automatically produced for *Stage 1*. Similar architectures were created for *Stage 2* and *Stage 3*.

For each architecture, automated performance optimization (PO) ended the high-level analog synthesis step. Optimization criteria were that the error of real AC behavior is within 50% from ideal AC behavior, and that filter area is minimized. The resultant of PO step was fixing constraints for op amp gains, unity-gain-frequency (UGF), output and input impedances and finding values for external resistors and capacitors. We decided to use *Architecture-10* for *Stage 1* as it had the smallest area, and its error was larger only by an insignificant amount as compared to *Architecture-1*, which provided the smallest output error. Similarly, we used *Architecture-5* for the first block of *Stage 2* as it had the smallest area, and its output error was acceptably large. Finally, we considered *Architecture-1* for the second block of *Stage 2* as it was the only one with a small output error.

The next step was circuit synthesis. Op amp constraint for gain, dominant pole, input and output impedance (found during PO) were input to a circuit synthesis tool [5] that sized op amp transistors. SPICE files for synthesized op amps were used to complete the filter description at the transistor level. Each of the SPICE models was simulated and resulting plots for *Stage 1* are shown in Figure 10. Similar plots were obtained for *Stage 2*. Simulation results show a good similitude between ideal AC behavior, predicted AC behavior after analog HLS and SPICE simulation after op amp synthesis.

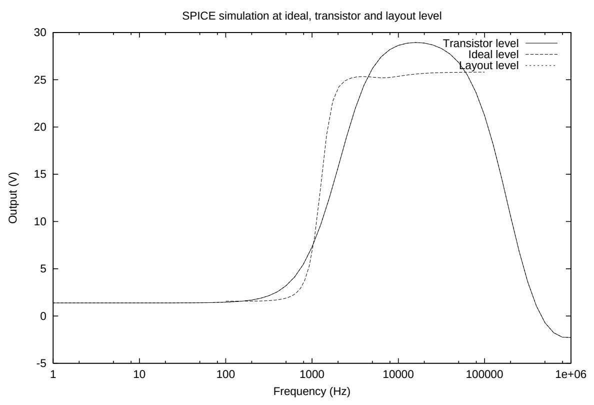

Figure 11. **SPICE simulation of optimized filter at transistor and layout levels**

The last step was layout synthesis. The SPICE description of the filter was given as input to KOAN/ANAGRAM placement and routing tools [3]. Parasitic capacitances were extracted from the produced layout and then simulated with SPICE. SPICE simulations are provided in Figures 11. Simulation results motivate that AC behavior of the filter after layout generation satisfies imposed performance constraints. Also, we concluded that for the optimized filter there is a good resemblance between AC behaviors evaluated at the system, transistor and layout levels.

6. Conclusions

This paper discusses specification issues for synthesis of mixed-signal and analog systems by defining the aBlox specification notation. aBlox provides constructs for expressing system functionality and structure, interactions among the analog and digital domains and performance models and constraints. The soundness of the notation semantics was achieved by basing it on a computational model for mixed-signal systems. The analog component of aBlox already serves as a specification notation for our existing top-down synthesis methodology. The described research is also important because it enables identifications of cases when functionality can be moved across analog and digital domains so that the semantics of a system is the same. Finally, some of the aBlox rules can be applied as guidelines for VHDL-AMS or Verilog-A specifications.

References

[1] "Verilog-A Language Reference Manual - Analog Extensions to Verilog HDL Version 1.0", IEEE, 1996.

[2] "IEEE Standard VHDL Language Reference Manual", IEEE Std.1076.1.

[3] J. Cohn *et al*, "KOAN/ANAGRAM II: New Tools for Device-Level Analog Placement and Routing", *IEEE JSSC*, Vol. 26, No. 3, March 1991.

[4] J. Davis *et al*, "Heterogeneous Concurrent Modeling and Design in Java", Technical Memorandum UCB/ERL M01/12, UC Berkeley, 2001.

[5] N. Dhanwada *et al*, "Hierarchical Constraint Transformation using Directed Interval Search for Analog System Synthesis", *Proc. of DATE*, 1999.

[6] A. Doboli *et al*, "A VHDL-AMS Compiler and Architecture Generator for Behavioral Synthesis of Analog Systems", *Proceedings of DATE'99*, 1999, pp.338-345.

[7] A. Doboli *et al*, "Behavioral Synthesis of Analog Systems using Two-Layered Design Space Exploration", *Proc. of the 36th DAC*, 1999, pp.951-957.

[8] A. Doboli, "Specification and Design-Space Exploration for High-Level Synthesis of Analog and Mixed-Signal Systems", Ph.D. Thesis, University of Cincinnati, 2000.

[9] S. Donnay *et al*, "Using Top-Down CAD Tools for Mixed Analog/Digital ASICs", *Analog Integrated Circuits and Signal Processing*, Kluwer, 1996, pp.101-117.

[10] P. Duran, "A Practical Guide to Analog Behavioral Modeling for IC System Design", 1998.

[11] J. Franks *et al*, "Ear Based Hearing Protector/ Communication System", United States Patent, Patent Number 5,426,719, June 20 1995.

[12] C. Ghezzi, M. Jazayeri, "Programming Language Concepts", John Wiley & Sons, 1998.

[13] G. Gielen, R. Rutenbar, "Computer-Aided Design of Analog and Mixed-Signal Integrated Circuits", *Proceedings of the IEEE*, December 2000, pp. 1825-1852.

[14] R. Gregorian, G. Temes, "Analog MOS Integrated Circuits for Signal Processing", John Wiley & Sons, 1986.

[15] G. Kopec, "Signal Representations for Numerical Processing", in *Symbolic and Knowledge-Based Signal Processing*, Prentice Hall, 1992.

[16] E. Lee *et al*, "GABRIEL: A Design Environment for DSP", *IEEE Trans. on Acoustics, Speech, Signal Processing*, ASSP-37, vol. 37, no. 11, 1989, pp. 1751-1762.

[17] K. Ogata, "Modern Control Engineering", Prentice-Hall, 1990.

[18] K. Swings, W. Sansen, "Ariadne, a Constraint-based Approach to Computer-aided Synthesis and Modeling of Analog Integrated Circuits", *Analog Integrated Circuits and Signal Processing Journal*, Kluwer, May 1993, pp.197-215.

[19] I. Vassiliou *et al*, "A Video Driver System Designed Using Top-Down, Constraint-Driven Methodology", *Proc. of ICCAD*, pp.463-468, 1996.