Towards a Specification Notation for High-Level Synthesis of Mixed-Signal and Analog Systems*

Alex Doboli, Ranga Vemuri
Digital Design Environments Laboratory, Department of ECECS
University of Cincinnati, Cincinnati, OH 45221-0030
Email: {adoboli, ranga}@ececs.uc.edu

August 15, 2000

Abstract

This paper discusses aBlox - a specification notation that we defined for automated synthesis of mixed-signal systems. aBlox addresses two important aspects of mixed-signal system specification: (1) description of functionality and performance issues and (2) expression of analog-digital interactions. The semantics of aBlox embeds concepts and rules of a computational model that we developed for mixed-signal systems. Finally, the paper shows some mixed-signal specifications that we developed in aBlox.

1 Introduction

Different specification styles can be used for describing mixed-signal and analog systems i.e. declarative style, imperative style, functional style, object-oriented style etc [10]. Each of the styles is useful for different synthesis scenarios or synthesis tasks. For example, declarative specifications are very popular for describing performance constraints and performance models for parameter optimization of analog circuits [19]. A declarative specification shows *what* a system does and not *how* it achieves its functionality. A declarative specification expresses relationships and constraints among objects that describe signals or circuit performances i.e. voltages, current, unity-gain-frequency, slew-rate, etc. Declarative specifications perform well if the implementation (structure) of a mixed-signal system is known and the goal is parameter optimization. Nevertheless, a declarative style does not provide any clues for producing structural implementations (hardware architectures) for a general system. Hence, identifying a proper specification style is one of the tasks when defining a synthesis-oriented specification notation.

A requirement for a specification language for mixed-signal synthesis is that the language must clearly distinguish functional and performance elements in a specification. Otherwise, erroneous situations might occur where systems are synthesized to emulate performance aspects. Finally, another need is to preserve a similar specification style for both the analog and digital parts of a mixed-signal system. If not, there will be a dramatic difference between the descriptions of the two domains. Hence, specification styles must be converted in order to perform analog/digital trade-off explorations during mixed-signal synthesis.

We feel that the first step in systematically synthesizing optimized mixed-signal implementations is to start from functional specifications at the *Signal Flow Graph* (SFG) level [17]. SFGs indicate the system behavior by showing the signal processing and flow. Similar approaches for basing a specification language on SFGs are proposed by Kopec [14] [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.
- Effective synthesis algorithms can be formulated for SFG-s [5]. SFG blocks suggest the structure of a system and they are easily mappable to electronic circuits as they represent operations i.e. amplification, integration, summing, etc.
- 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.

This paper describes aBlox - a specification notation for automated synthesis of mixed-signal and analog systems. aBlox notation permits functional descriptions of mixed-signal systems at the level of SFGs. aBlox constructs address two key

*This work was sponsored by the USAF, Air Force Research Laboratories, Wright Patterson Air Force Base under contract number F33615-96-C-1911

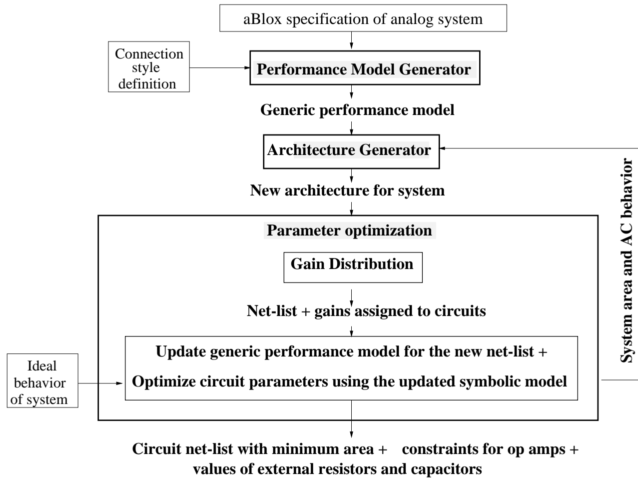

Figure 1: Behavioral analog synthesis flow

aspects: (1) description of functionality and performance issues and (2) expression of analog-digital interactions. Following concrete aBlox elements target the two general goals:

1. The aBlox "philosophy" is to explicitly describe signal processing and flows. This is important for having a similar description style for both analog and digital domains so that analog-digital trade-off exploration can be easily performed.

2. aBlox constructs encourage a hierarchical and modular description of systems. This is useful for increasing the effectiveness of both specification and synthesis tasks [5].

3. aBlox provides a notation for linking performance constraints or performance models to the constituting modules of a program. This is important as components of a mixed-signal system tend to have very heterogeneous performances. For example, the analog part of a telephone set [20] includes two modules, a receiver and a transmitter, with different noise constraints.

4. aBlox provides a well defined interface mechanism for describing analog-digital interactions.

The definition of aBlox was motivated by the absence of any feasible specification notation for mixed-signal synthesis. Existing languages i.e. VHDL-AMS [3], Verilog-A [2], MAST [8] are all simulation oriented. There are difficulties in adapting there semantics for synthesis [4].

The paper has the following structure. Section 2 presents the main tasks that are performed by our analog synthesis methodology. Section 3 discusses the aBlox constructs for specifying system functionality and Section 4 those for expressing higher-order functions. Section 5 concentrates on performance model description in aBlox. Finally, conclusions will be provided.

```
macro receiver is continuous_time
  inputs
    line is voltage
      with range 0-1.0 V;
    local is voltage
      with range 0-1.0 V;
  outputs
    earph is voltage
      with range 0-1.5 V
      with impedance 280.0 Ohms;
  attributes
    noise <= 80 dB;
    bandwidth is range 300-3600 Hz;
  arch receiver is
    output = (100.0 * line + local) * -1.78;
  end arch;
end macro;
```

Figure 2: Specification of the telephone receiver system

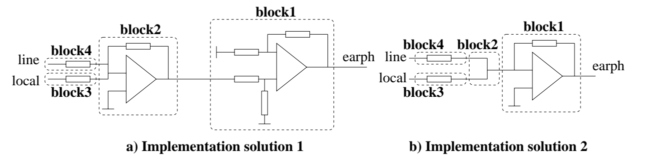

Figure 3: Net-list samples for receiver module

2 Synthesis Approach

We illustrate the successive synthesis steps by referring to a much simplified version of the receiver module of a telephone set [20]. The receiver provides an audible output signal to the earphone of the telephone set. It amplifies with different gains incoming signals from the calling part (signal *line*) and locally produced by its own microphone amplifier (signal *local*). The specification imposes that port signals *line*, *local* and *earph* are voltages and that their value ranges are $[0.0, 1.0]$V for inputs and $[0.0, 1.5]$V for the output. The output load is 280Ω. Our synthesis method assumes that specifications express *how* continuous-time analog behavior results as signal flow and processing. Signal-flow graphs (SFG) accommodate well this description style and they are specified as *aBlox* programs in our environment. Figure 2 depicts the aBlox program for the receiver module.

The considered analog synthesis methodology is depicted in Figure 1. The **performance model generator** [6] produces a generic *computational tree* that describes how system parameters depend on parameters of the blocks composing the system. A computational tree referred in the paper as *Analog Performance Tree* (APT) is an uninterpreted variant of the closed-form symbolic expressions produced by traditional methods [11]. Symbolic models for all net-lists explored during synthesis result by updating the generic model in a very short time (linear with the number of blocks).

The **architecture generator** [7] 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 3a and 3b illustrate two distinct implementations for the receiver module obtained by our method.

Area and AC behavior of each net-list are determined by net-list **parameter optimization** [6] and used to guide the net-list generator. Parameter optimization step first updates the implementation dependent part of the generic performance model corresponding to the net-list to be optimized. Next, 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 frequency behavior of a system is within an error margin from the desired behavior. To guarantee that feasible solutions result, each free parameter was modeled by a feasibility range for CMOS technology [12] i.e. external resistors are in range $[1, 100]\text{k}\Omega$, op amp gains in range $[10^3, 10^4]$ etc.

3 aBlox Notation for Specifying Mixed-Signal Systems for Synthesis

An aBlox program describes interacting analog and digital domains. These domains can have a hierarchical structure as a domain is built of parts, stages, components, etc. The *macro* construct is the main notation feature for describing functionality, hierarchy and interface of a system, sub-system or block. Samples of macro definitions are illustrated in Figure 4 for a two stage 4-th order filter. The latter figure suggests how macro definitions and macro calls are employed for expressing hierarchy description for a system. A mixed-signal specification must contain a top-most macro (the macro that is not called by other macros). The semantics of the top-most macro is that it executes 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 trade-off exploration.
- *Input* and *output ports*: ports indicate 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 parameter* 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 4 are characterized by different filter constants that are specified as generics in the program. Thus operators can be passed as arguments to macro calls for describing stages built of distinct blocks but connected in similar patterns.
- *Attribute section* is currently allowed only for macro-s of the continuous-time analog domain. It allows description of declarative or equational performance models that can be associated with a macro and then used for synthesis.
- *Macro body* expresses 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.

```
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 4: aBlox specification for a filter

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 macro 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. The second part of the rule refers to situations where outputs of the same operators or macro calls are used as inputs multiple places (i.e. operation *integ (p)* in macro *stage* in Figure 4).

Rule 2: Inside a macro with a *continuous_time* domain descriptor only macros with *continuous_time* or without any domain descriptor can be called. Inside a macro with a *digital* domain descriptor only macros with *digital* or without any domain descriptor can be called.

Rule 3: Inside a macro without a domain descriptor following three 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. (3) A top most macro without a domain descriptor can call macros with both *continuous_time* and *digital* domain descriptors.

These semantic rules prohibit developing a hierarchy of subsystems in which a subsystem (excepting the top-most macro)

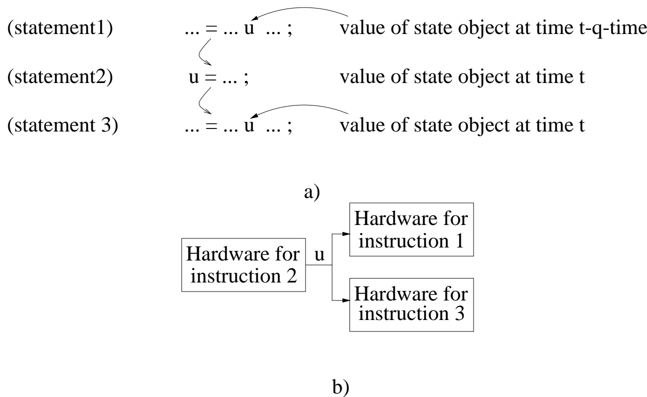

Figure 5: Semantics of instruction sequence

includes both time models. Hence, a synthesis tool can easily identify the parts meant for analog and digital realizations.

Semantic rule for mapping variables to ports

Rule 4: Ports of the top-most macro have types *voltage*, *current* or *digital* as the interface with the environment is assumed to be well defined. These ports can be annotated with attributes i.e. value ranges, impedances etc using the *with* construct (see Figure 2).

Semantic rule for variable scoping

Rule 5: The scope of variables defined inside a macro is limited to the macro body.

Semantic rule 5 is important to guarantee that macro definitions have the meaning of functions, thus the property of *referential transparency* [10]. Hence, a macro's functionality (meaning) is not influenced by its connections with other macros.

Semantic rule for describing domain interactions

Rule 6: 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 are performed depending on variable/port types.

Data objects, expressions and assignment instruction

We defined two types of data objects depending on the time intervals they are capable of preserving their values (their life-time). Data objects can be with or without memory. Memory-less objects preserve their value for one q-time (a very small time interval) if they are in *continuous_time* macros or one clock cycle if they pertain to *digital* macros. Memory elements can store their values for more than one q-time or one clock cycle. This classification is due to implementation specifics for objects i.e. as wires, latches, flip-flops and is a link between the abstract specification and a physical implementation.

Semantic rules for aBlox objects:

Rule 7: All variables of *continuous_time* macros denote memory-less objects. 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 8: Variables in *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 9: Expression operators are different for the two domains:

- The *continuous_time* macros include the following arithmetic operations: addition, subtraction, multiplication by a constant and integration. This is a complete operator set for a linear system [22] and it can be implemented with simple electronic circuits [9]. For smoothening the specification task we introduced unary add and minus operators (i.e. in statement $n = + o$ in Figure 4). Its argument is an array variable. The operator returns the sum of all array elements. The operator is useful for describing multi-entry additions.
- *Digital macros* include *arithmetic operators* i.e. addition, subtraction, multiplication, division and *logical operators* i.e. and, or, negation, etc.

Semantic rules for assignment statement:

Unconstrained assignment statements violate the functional character of a specification as they introduce side-effects. Although side-effects do not pose any problem for digital synthesis they might be difficult for analog synthesis. 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.

Rule 10: An assignment statement is viewed as a connection between a name (the left part of an assignment) and an anonymous function definition (λ-expression [10]) defined 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 11: Memory-less objects are updated immediately after their corresponding assignment statement is executed.

Rule 12: Updating of memory variables implicitly happens after executing the last statement of its defining macro.

Semantic rule for instruction sequence:

Rule 13: It is not allowed to assign a variable of a *continuous_time* macro 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 the same $q - time$ it has more than one value. Variables can have only a single value in our model (we assumed that each distinct data object has a different name).

Rule 14: Any variable or output port of a *continuous_time* macro that is referred by a statement must also appear in the left part of an assignment statement.

This guarantees that a continuous-time data object has a value at any *q*-time.

Rule 15: Semantics of data dependencies among instructions of *continuous_time* macros describes signal flows among processing blocks.

For example, consider the situation in Figure 5, where object *u* denotes a variable of the analog domain. Figure 5(a) depicts a program fragment and Figure 5(b) shows how data dependencies among instructions express signal flows between the processing macros.

Observation: If previous semantic rules hold for the *continuous_time* macro then any sequencing (ordering) of a given set of instructions produces the same block structure.

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 and its behavior can be described with *if* statements. There are no special semantic rules for the digital domain but there are some requirements for *if* statements of *continuous_time* macros.

Semantic rules for if statement:

Rule 16: 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 object values, which are 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. This contradicts the assumption that any analog object is permanently a-life.

Rule 17: An object of the analog domain that is assigned inside an *if* statement cannot be also assigned outside the *if* statement.

This is a particular case of Rule 13 for instruction sequence.

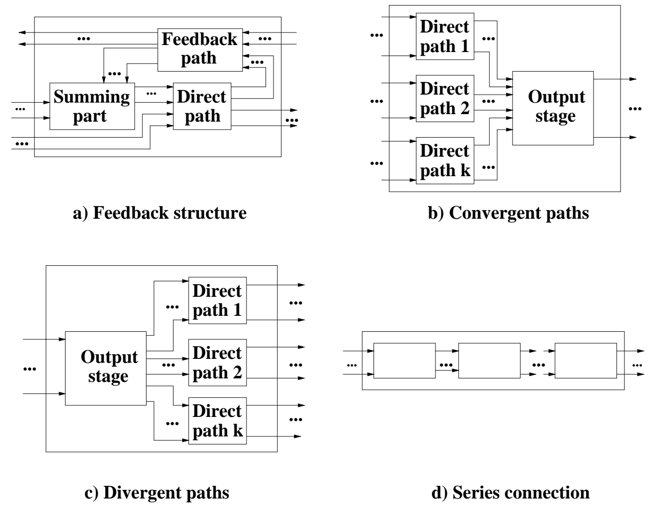

Figure 6: Block structure

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

There are two reasons for this rule:

- For addressed applications, the modes of functioning of the analog domain are selected by signals coming from outside the continuous-time domain. This is reasonable as we assume that any non-linear functionality i.e. comparing two analog signals is outside the system.
- To avoid repeated "switching" of *if* statement executions due to changes in each *q*-time. The semantics for updating digital objects prohibits repeated switchings.

4 Higher-order functions

An important aspect for mixed-signal synthesis is the possibility to use hierarchical specifications. Hierarchy helps not only in abstracting design elements irrelevant to a particular synthesis task but also in approaching synthesis complexity. Another opportunity for addressing synthesis complexity is to identify and exploit any uniform (similar) parts in a system. Thus, hierarchy and uniformity are important for making system specification easier and more readable and synthesis tasks i.e. performance model generation more efficient.

Macro definitions and macro calls permit a hierarchical specification of a system. For example, a two stage filter is described in Figure 4. Both stages are in controllable form

```
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 7: aBlox instructions for expressing block structure

and they contain blocks performing similar kind of functionality. Macros are a convenient way for describing identical parts of a system. Moreover, it can happen that similar block structures occur in a system but involving different blocks. Such situations introduce structural uniformity that can be exploited for synthesis [6].

aBlox notation permits definition of higher-order functions to allow full re-use of structural uniformities. Higher-order functions are macro-s that have other macro-s among their parameters. Structural uniformities are expressed as interconnections of generic blocks. Such macros are instantiated for different actual aBlox blocks or macro-s. The only restriction for higher-order function definitions is that is must be possible to produce the global block structure of a system at compile time.

Rule 19: No direct or indirect recursions are allowed for any macro-calls including higher-order macro calls.

Rule 20: A higher-order function is described in aBlox language by indicating in the *generics* part the signature of the generic macro structure. Signature is defined by enumerating the type of inputs and outputs for the generic structure so that it can be verified when calling the higher-order macro.

A special category of higher order functions are aBlox constructs *feedback_structure*, *convergent_paths*, *divergent_paths* and *series_blocks*. Figure 6 illustrates the corresponding block structures. These constructs are not orthogonal as their behavior can be achieved with the already existing aBlox instructions. They were introduced to increase the readability of aBlox programs. Figure 7 depicts the syntax of the four aBlox constructs.

5 Description of performance models

In the process of exploring different notations for specifying mixed-signal systems for synthesis, we found that a declarative description style is still needed. Following reasons motivate our conclusion:

- Macro-s can express structures with heterogeneous design-performance constraints. In Figure 2, the transmitter and the receiver module of the telephone set have different noise and bandwidth constraints.
- Explicit description of performance models of a system could be required for synthesis. An ideal mixed-signal synthesis tool would have the ability of automatically inferring all performance models needed for synthesis. 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 of 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 has a flexible mechanism of attribute 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 many possible performance elements i.e. raise-time, fall-time, settle-time, slew-rate, sensitivity, unity-gain frequency etc.

We extended the formulation of Rosenberger *et al* [18] for system-level performances by defining a flexible notation.

Rule 21: The notation for declarative descriptions includes four types of constructs:

- Primitive constructs
- Predicate definitions
- Attributes definitions
- Model definitions

Declarative constructions can be global or local to aBlox macros. Global declarations are defined using an *attribute_package* construct. 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. 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 22: The *primitive constructs* for declarative specifications are:

1. **Signal characteristics** such as voltage, current, phase and frequency can be denoted using the *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.

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 23: *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 in time i.e. slew-rate. Predicates refer to primitive constructs or attribute definitions.

Rule 24: An *attribute definition* associates a name with a predicate.

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

Rule 25: *Model definitions* introduce a set of equations that are simultaneously approached (solved) during synthesis for obtaining performance values.

Model definitions have a simulation character and their need is a consequence of the current knowledge on developing automated CAD tools. aBlox does not explicitly define how model definitions are solved but linear or non-linear solvers are possible options. The concept of this mechanism is similar to *simultaneous* statements [3] in VHDL-AMS. 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* [21]:

*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);*

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 [5]. The research described in the paper is also important because it identifies situations for which functionality can be moved across the analog and digital domains so that semantics of the resulting systems is the same.

References

[1] "IEEE Standard VHDL Language Reference Manual (Integrated with VHDL-AMS changes)", IEEE Std.1076.1.

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

[3] "IEEE Standard VHDL Language Reference Manual (Integrated with VHDL-AMS changes)", IEEE Std.1076.1.

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

[5] A. Doboli, A. Nunez-Aldana, N. Dhanwada, S. Ganesan, R. Vemuri, "Behavioral Synthesis of Analog Systems using Two-Layered Design Space Exploration", *Proc. of the 36th DAC*, 1999, pp.951-957.

[6] A. Doboli, “Specification and Design-Space Exploration for High-Level Synthesis of Analog and Mixed-Signal Systems”, *PhD Dissertation*, University of Cincinnati, 2000.

[7] A. Doboli, N. Dhanwada, R. Vemuri, “A Heuristic Technique for System-Level Architecture Generation from Signal-Flow Graph Representations of Analog Systems”, *Proc. of ISCAS’2000*, Geneva.

[8] P. Duran, “A Practical Guide to Analog Behavioral Modeling for IC System Design”, 1998.

[9] S. Franco, “Design with Operational Amplifiers and Analog Integrated Circuits”, McGraw Hill, 1998.

[10] C. Ghezzi, M. Jazayeri, “Programming Language Concepts”, John Wiley & Sons, 1987.

[11] G. Gielen, W. Sansen, “Symbolic Analysis for Automated Design of Analog Integrated Circuits”, Kluwer, 1991.

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

[13] B. J. Hosticka, W. Brockherde, R. Klinke, R. Kokozinski, “Design Methodology for Analog Monolithic Circuits”, *IEEE Transactions on Circuits and Systems - I: Fundamental Theory and Applications”*, Vol.41, No.5, May 1994, pp.387-394.

[14] G. Kopec, “The Signal Representation Language SRL”, *IEEE Transactions on Acoustics, Speech and Signal Processing*, vol. ASSP-33, No.4, August 1985, pp.921-932.

[15] G. Kopec, “Signal Representations for Numerical Processing”, in *Symbolic and Knowledge-Based Signal Processing*, editors: A. Oppenheim, S. Hamid Nawab, Prentice Hall, 1992.

[16] E. Lee, W.-H. Ho, E. Goei, J. Bier, S. Bhattacharyya, “GABRIEL: A Design Environment for DSP”, *IEEE Transactions 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] R. Rosenberger, S. A. Huss, “A Systems Theoretic Approach to Behavioral Modeling and Simulation of Analog Functional Blocks”, *Proceedings of DATE*, 1998, pp. 721-728.

[19] 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.

[20] J. Trontelj, L. Trontelj, G. Shenton, “Analog Digital ASIC Design”, McGraw-Hill Book Company, 1989.

[21] I. Vassiliou, H. Chang, A. Demir, E. Charbon, P. Miliozzi, A. Sangiovanni-Vincentelli, “A Video Driver System Designed Using Top-Down, Constraint-Driven Methodology”, *Proc. of ICCAD*, IEEE CS Press, pp.463-468, 1996.

[22] L. Zadeh, C. Desoer, “Linear System Theory”, McGraw Hill, 1963.