Overview of Analog-VHDL Requirements with Contrasts to Other Languages

By Kevin Nolan

1.0 What is an AHDL?

An Analog Hardware Description Language (AHDL) allows the *direct* specification of both the *behavior* and *structure* of analog systems. These analog systems are not simply confined to electrical circuits. An AHDL should support any system that can be described as a set of nonlinear ordinary differential, algebriac, and boolean equations. This also includes thermal, magnetic, rotational, etc. in any combination. The ability to describe systems incorporating mixed technologies is essential to an AHDL. Consider the following definition:

> An Analog Hardware Description Language is a formal descriptive paradigm which allows the direct specification of both the *behavior* and *structure* of both analog and digital systems.

The specification of *behavior* in an AHDL is given by specifying the equations that govern the behavior of the hardware element being described. For electrical systems, these equations usually relate currents to voltages as a function of time. It is important to note that in analog systems, time is represented as a continuum, and currents and voltages are represented as continuous. Digital simulators represent both time and signal levels as discrete values. An AHDL supports both domains in order to effectively simulate all aspects of today's designs.

The *structure* of an analog system in an AHDL is described both in terms of its inherent hierarchy and the interconnectivity of its constituent analog elements.

1.1 AHDL Requirements Checklist

One of the most basic requirements of an AHDL is that it model the physical conservation laws which are imposed by nature on physical systems. Examples of such conservation laws is the conservation of energy law which states that energy can neither be created or destroyed; it can only change its form. Electrical systems have energy conservation laws for charge, current, voltage, etc. An AHDL preserves these properties by allowing the equations which relate one conserved quantity to another to be implicitly solved as part of the underlying paradigm.

Specifically, the top-down design of today's circuits and systems places the following requirements on an AHDL:

❏ Physical conservation laws	Components that obey physical laws (normal electrical elements), and components that do not obey physical laws (control blocks) must both be supported.
❏ Technology independence	Components of any technology (electrical, mechanical, thermal, optical, fluid, etc.) must be supported, together with *any* mixture of these technologies within a component (electromechanical, electrothermal, etc.)
❏ Continuous & discrete signals	The AHDL must support components with continuous time and signals (analog blocks), components with discrete time and boolean signals (digital blocks), and components with discrete time and continuous signals (sampled-data system blocks), or any mixture of any of these inside any block.

- ❏ Fully hierarchical

  The AHDL should support full hierarchy in the system, a mix of technologies and concepts throughout the hierarchy, and be able to communicate information between blocks within the hierarchy.

- ❏ Full modeling capabilities

  A complete range of modeling capabilities must be supported, including:

  - ❏ any number of simultaneous, nonlinear algebraic and differential equations with any interactions between them (full behavioral modeling)
  - ❏ the ability to modify the characteristics of a block as a function of time or a condition in that block or any other block
  - ❏ extendable elements (variable number of pins, parameters, etc.)
  - ❏ the ability to use models written in standard programming languages
  - ❏ the ability to mix analog and digital signals and capabilities in any model
  - ❏ the ability to mix any technologies in any model
  - ❏ statistical dependencies for all parameters for all components of any technology
  - ❏ the specification of behavior in the frequency domain
  - ❏ the ability to assign noise properties to any physical component
  - ❏ either port-based or terminal-based descriptions
  - ❏ the ability to specify initial conditions
  - ❏ computation of small-signal parameters for any component of any technology

- ❏ Compatibility

  The AHDL must be able to tie to existing technology simulators (e.g., digital simulators), achieve the same results as existing analog simulators, and have the ability to incorporate all emerging AHDL standards.

- ❏ Ease of use

  The AHDL must support the ability for ANY user (experienced and novice) to create models and provide libraries of existing models, including:

  - ❏ electrical
  - ❏ analog
  - ❏ digital
  - ❏ sampled-data systems
  - ❏ interface models between these
  - ❏ mechanical
  - ❏ electromechanical
  - ❏ thermal
  - ❏ electrothermal

1.2 How AHDLs Differ from Behavioral Languages

Any computer language can be called a behavioral language. FORTRAN and C, for example, are used in SPICE to describe the behavior of transistors. C++ is used in MIDAS to describe the behavior of sampled-data systems. The problem with using a computer language to describe analog systems is that there is no way to *directly* specify, as mathematical equations, the behavior of an element.

In SPICE3, for example, approximately 300 lines of C code are needed to model a junction diode with parasitic capacitances. Using an AHDL, however, approximately 6 equations are needed to model the equivalent behavior. These 6 equations directly specify the physical behavior of the diode. No manual interpretation is needed to transform them to a behavioral language like C. The 6 equations are:

```
id     = v(p,m)/res          # current through series resistance
qd     = v(p,m)*jcap         # charge storage
i(p)  += id                  # contribute current to + terminal
i(pi) += d_by_dt(qd) - id    # contribute change in charge
i(n)  -= idi + d_by_dt(qd)   # contribute neg current to - terminal
```

Exmaple 1: Diode Equations in MAST

In AHDLs, the underlying semantics of the language and its syntax are designed to succinctly and accurately represent the structure and behavior of analog systems.

Most computer languages are *procedural* in nature. Behavior is described within them as a series of steps to be performed to calculate a given result from a given set of inputs. AHDLs, on the other hand, have a procedural aspect but are *declarative* in nature. The procedural part is typically used to validate or manipulate parameters for a particular model. The declarative aspects are used to declare the behavior of the devices as a set of equations.

1.3 How AHDLs Differ from Macromodeling

Macromodeling is defined here as the process of creating models by using pre-existing primitives to construct a more complex structure. Macromodeling is usually identified with SPICE. A macromodel, for example, would use current sources and transistors to model a differential amplifier. The problem is, however, circuit behavior must be described as a network of pre-existing primitives. The artificial transformation of behavior to primitive structure obfuscates the expression of the behavior, and introduces simulation inefficiencies. An AHDL, on the other hand, allows complex behavior to be expressed directly in simple mathematical terms.

Quite often it is necessary to develop a model or set of models in order to perform computer simulations of a circuit or system. A model of a physical device may be required so that relatively detailed phenomena of a circuit can be studied. A model of a larger circuit or system may be required in other circumstances. The level of detail in a system model may not have to be as high as that of a device model, but as part of a simulation of a larger system may still be quite useful.

Implementing a new model in a SPICE-like simulation program [1][2] is a tedious and time-consuming task. This is the case if the model is to be implemented in the SPICE program as a primitive or simply constructed as a macromodel from other SPICE primitives. This task is much easier if a more powerful technology is used.

The following example addresses an article [3] (*previously published in Circuits & Devices*) to illustrate that simulation and modeling using an AHDL is the preferred approach for modeling and simulating a broad range of circuits and systems in many applications. This article presented a modeling problem that is indicative of the level of effort required to “shoehorn” the model into SPICE. It contained a detailed description of a method to solve the following simultaneous nonlinear differential equations using SPICE. The solution achieved using the MAST® AHDL is then presented and described.The differential equations are:

$$\dot{x} = y\,(1 - z)$$

$$\dot{y} = x\,(15 + z) - 2\frac{y}{t}$$

$$\ddot{z} = z - 2\frac{\dot{z}}{t} - 2\,(x^2 + y^2)$$

The initial conditions are given as:

$$x = 1.8$$

$$y = 0$$

$$z = 7$$

$$\dot{z} = 0$$

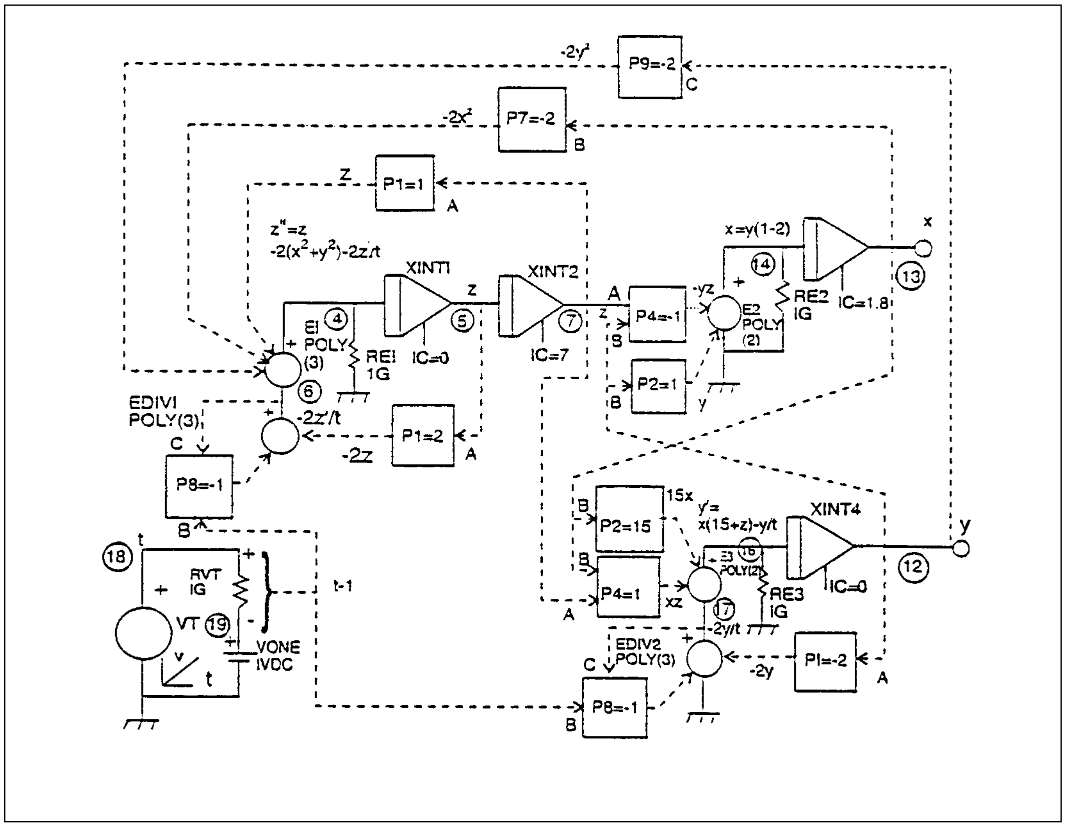


Fig. 1: SPICE2 network to simulate the example differential equations.

The diagram in Fig. 1, taken from [3], shows the SPICE2 network used to simulate the previous set of equations.

There is, however, a more straightforward way to solve this problem. Although the diagram of the analog computer realization in [3] is quite impressive, it doesn't “look” much like differential equations. In an ADHL, the problem is posed to the simulator in much the same way as it was posed to the engineer. In fact, the description looks almost identical to the original problem statement. It almost seems like cheating.

Missing are artifacts like the 1000 MΩ resistors needed to satisfy SPICE's restriction of floating nodes. The following model, written in the AHDL MAST®, would simulate the example differential equations:

```
make d_by_dt(x) = y*(1-z)
make d_by_dt(y) = x*(15+z) - 2*y/t
make d_by_dt(zprime) = z - 2*zprime/t - 2*(x**2 + y**2)
make d_by_dt(z) = zprime

control_section {
   initial_condition(x,1.8)
   initial_condition(y,0)
   initial_condition(z,7)
   initial_condition(zprime,0)
}
```

Exmaple 2: SPICE'S Differential Equations in MAST

All of the semiconductor models in SPICE are nonlinear differential equation-based. With SPICE, the differential equations must be implemented in the SPICE source code or by using techniques similar to those shown in Fig. 1. The task of implementing the differential equations in the SPICE source code is difficult. The resulting models cannot be transferred and shared without transferring the entire SPICE code. This is one of the primary reasons why much effort has been put into modeling with macromodel (*such as the SPICE differential equation solving techniques described here*). Macromodeling is a fine technique, but limited. The problem with SPICE macromodeling is that there are insufficient fundamental building blocks with which to work. This is why a description language, such as an AHDL, is essential for modeling efforts to move forward. Technical journals are full of excellent modeling work. A standard AHDL will make models more readily available to interested engineers.

1.4 How AHDLs Differ from Existing HDLs

An AHDL differs from existing HDLs in a variety of ways. It is reasonable to say that an ADHL represents a technology which is qualitatively different from technologies in use in HDLs today. In this section, a prototypical AHDL is compared with SPICE, VHDL, CSSL, and MAST®. Except for MAST®, each comparison reveals significant contrasts.

1.4.1 AHDLs vs. SPICE

On of the most fundamental aspects of an AHDL is its distinct separation from the simulator. Writing a SPICE model requires detailed understanding of the operation of the simulator and, once written, actually becomes part of the simulator (i.e., it is linked in). This makes writing a SPICE model more like extending the simulator rather than describing model behavior.

A SPICE deck cannot really be considered an AHDL. SPICE decks can only describe the structure of an analog system, not its behavior. One could argue that by allowing SPICE models to be parameterized, which they are, behavior is being described by those parameters. But this argument really doesn't hold up. Most SPICEs do not allow equations to be given for parameters, which is a requirement for behavioral modeling. Even the SPICE implementations that do allow expressions as parameters restrict the flexibility of their application. Such parameters can only modify the behavior of controlled sources. One example of this limitation is that there is no way to model an inductor in SPICE.

1.4.2 AHDLs vs. VHDL

VHDL is a hardware description language. As such, it provides a model of hardware behavior, a model of time, and a model of structure[4]. Each of these models must be expanded to allow for the specification of analog behavior. VHDL was originally designed to model systems whose behavior can accurately be predicted by analyzing the propagation of discrete quantities across a directed network in a discrete amount of time. Each of these transactions is called an *event*. The state of the system at time *t* is a deterministic function of the discrete events which precede *t*[5].

1.4.2.1 Digital Behavior vs. Analog Behavior

The simulation of analog (electrical) systems, on the other hand, requires that the system, and its constituent elements, be modeled not as the side-effects of discrete events, but as a system of non-linear ordinary differential and algebraic equations. The system must obey certain conservation laws; for electrical systems these are Kirchoff's Current Law (KCL) and Kirchoff's Voltage Law (KVL)[6]. KCL states that the sum of all currents entering each node is exactly equal to the sum of the currents leaving that node. KVL holds that the sum of all voltages around any loop in the circuit must be exactly zero. Simulation of such a system, therefore, requires that all analog elements be allowed to change their state continuously as a function of time.

It is useful to compare how each of the models of behavior, time, and structure currently supported by VHDL apply to the simulation of analog systems. The tables below represent these contrasts. It should be noted that although the tables often present the modeling characteristics of the two domains as being diametrically opposed, they are, in fact, orthogonal. Indeed, every digital system can be shown to be simulatable using analog techniques, the only trade-off being a significant degradation of simulation efficiency, and an over-supply of data.

VHDL	Analog
Directed functional transformation of input signals to output signals.	No directed input to output relationships; "inputs" and "outputs" affect each other simultaneously.
Modeled as a discrete system.	Modeled as a continuous system.
Elements are modeled as decoupled processes communicating via signals.	Elements are tightly coupled, each mutually influencing each other's state simultaneously.
Behavior of an element is modeled algorithmically as a process consisting of sequentially-ordered operations.	Behavior of an element is modeled declaratively as an unordered set of equations describing relationships and contributions.

Table 1: Behavioral Aspects

Clearly, only the behavioral characteristics which are actually being modeled can be observable in a simulated system. A logic simulation of a digital circuit can give detailed information on logic states and timing information, but it can say nothing about circuit loading, power consumption, or the effects that self-heating and parasitic capacitances have on transistor switching delays. For instance, a three-input NAND gate, modeled electrically using bipolar TTL, would typically be a circuit consisting of 8 transistors, 3 diodes, and 6 resistors[7] (see Fig. 2). Additionally, each transistor would be modeled[1] internally as a sub-circuit consisting of a current source, 4 diodes, 3 resistors, and 5 nonlinear capacitors[8] (see Fig. 3).

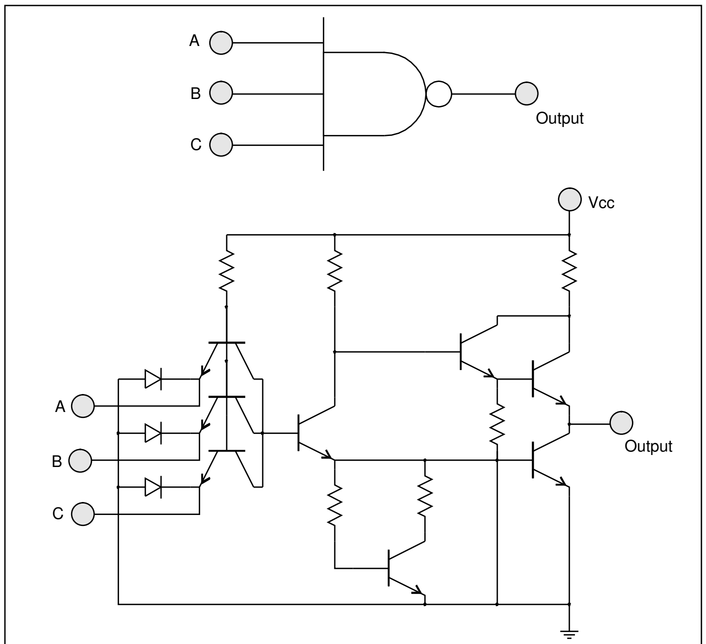


Fig. 2: Three-Input TTL Nand Gate

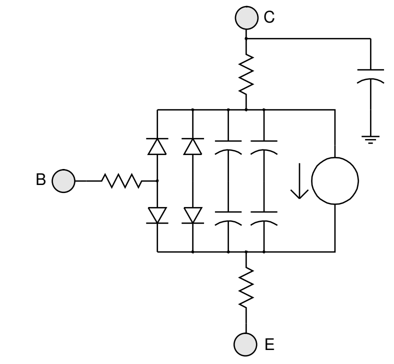


Fig. 3: Bipolar Transistor (Gummel-Poon Model)

A three-input NAND gate modeled at this level exhibits much more complex behavior than the same gate simulated with logic states. An essentially one line boolean expression becomes a system of fifteen equations with fifteen uknown when described in analog terms. This increase in the volume of observable information, though, comes at the expense of model complexity and simulation efficiency.

Consider the circuit and timing diagrams below (figures 4 and 5). In figure 5, the upper half of the diagram represents the values input and output signals of three-input NAND gate when modeled using discrete

1. There are essentially two archtypical descriptions proposed for the bipolar junction transistor (BJT). The first, proposed in 1954 by Ebers and Moll[9], remained the industry standard until 1970. An improved version based on Ebers-Moll was advanced by Gummel and Poon[10] in 1970. They improved on the Ebers-Moll(EM) model by adding additional parasitic effects and associated paramaters. Notable simulators such as SPICE use the Gummel-Poon (GP) model for BJTs, however, reduced accuracy simulations can be achieved by omitting some of the GP paramaters whereupon the model behaves like the older EM model.

events. The lower portion of the graph displays the voltage of the output signal for the analog version of the NAND gate when driven with the same inputs. Notice the glitches occurring at 300 and 500 nanoseconds. These glitches would were not detected by the digital simulator, but become readily apparent from the comensurate analog simulation. This gives an example of the kind of critical information which is computed as the result of an analog simulation which is not available using event-driven simulation techniques. Often the thousand-fold increase in simulation time required for analog is necessary to resolve the fine-grained behavior of analog components.

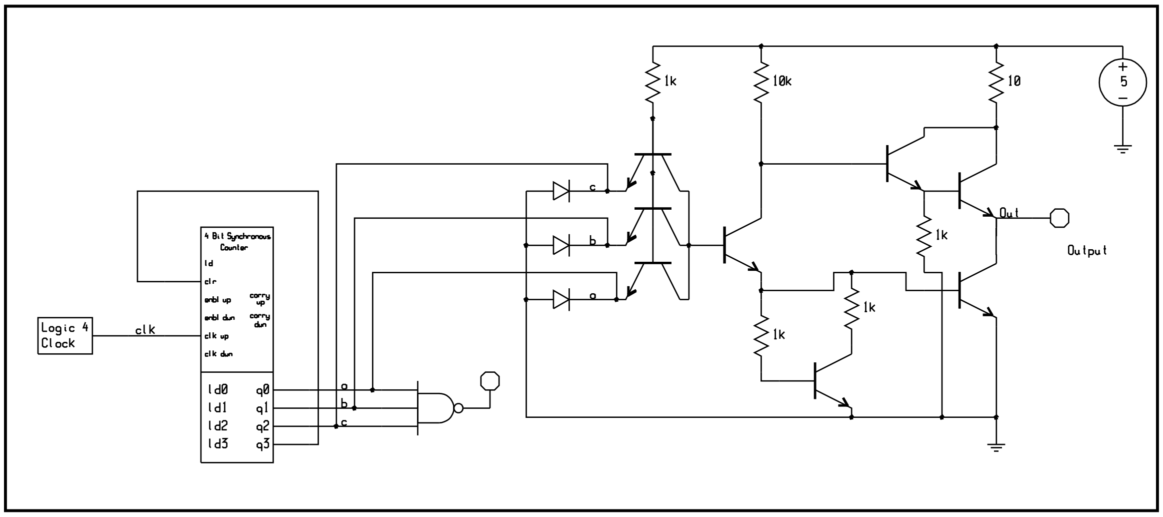


Fig. 4: Circuit containing logic NAND and analog NAND

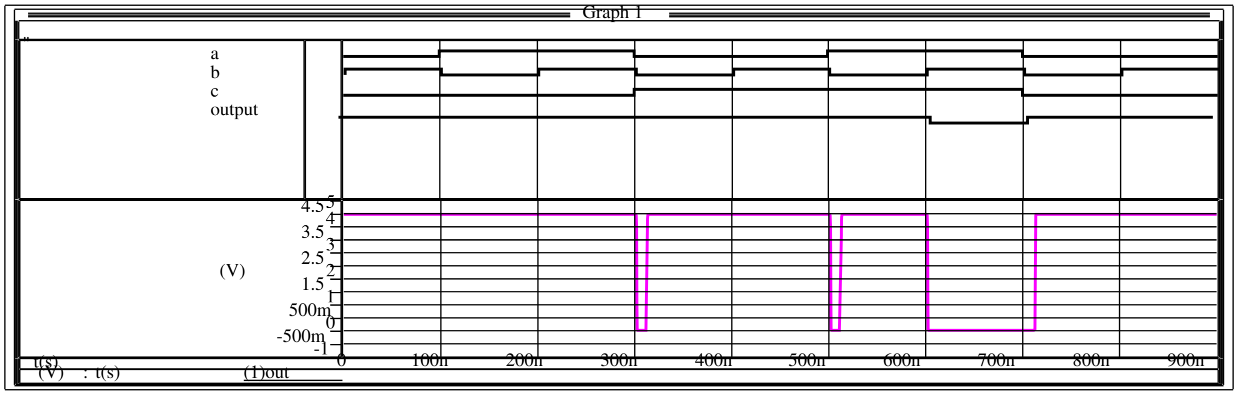


Fig. 5: Timing diagram of three-input NAND gate

1.4.2.2 Digital Time vs. Analog Time

The VHDL standard clearly defines the meaning of simulation time, as well as the sequence of events and actions during a simulation cycle. The intent was to ensure that simulations are deterministic[2]. When simulated, a given circuit should behave exactly the same way, every time, and using any vendor's simulator. Furthermore, the state of the system is, by definition, knowable and observable for any time *t*, provided *t* is an integer multiple of the minimum simulation time interval. This is a precise definition, and, in practice, sufficient.

VHDL	Analog
Time represented as discrete intervals.	Time represented as a continuum.
Each time step is divided into two stages: a stimulus stage then a response stage (cause-effect).	No distinction between stimulus and response; both happen simultaneously and instantly.
Minimum time interval is one femto-second.	No static minimum interval is available; it is a function of the precision of the underlying floating point behavior.

Table 2: Temporal Aspects

With analog systems, however, the situation is different. The state of an analog system can be known precisely for any time *t* by solving the set of nonlinear differential and algebraic equations describing the system's transient behavior. For certain small systems this can be done analytically, providing an exact solution. For most non-trivial or real-world systems, however, numerical methods are employed to give an approximation of the solution[6][11][12]. In general, these methods operate by descretizing the time continuum into the necessary number of time points to accurately compute the behavior of interest. For each new time point, the system of equations is linearized, and then iteratively solved until the desired accuracy is attained.

There are several consequences of simulating transient behavior using numerical approximation. One is that the transient behavior of the circuit is only computed at specific time points. Behavior between those points can be approximated through interpolation. (Even though the computed solution is an approximation, the computed solution is guaranteed to have a *relative error* to the exact solution which is less than the *local truncation error* specified for the simulation, provided the previous solutions are exact.)[3] The problem is, of course, that the previous solutions are themselves approximations and therefore not exact. The result of this compounding of local truncation errors is known as *global truncation error*. Unfortunately, there is no way to predict or measure the amount of global truncation error for a given local truncation error other than changing the local truncation error, re-simulating (re-integrating), and comparing the results with a previous simulation. Comparing the results of successive simulations with decreasing local truncation errors is a process known as *calibration*.

Some circuits may yield a system of equations which is numerically unstable. Numerical instability will cause the numerical error component of each approximation to increase, rather than decrease. When this happens, it is usually impossible for the simulation algorithms to converge on a solution which maintains the local truncation error criteria.

Additionally, the choice of simulation techniques places certain limitations on models. SPICE, for instance, requires the first derivative of a model, with respect to time, to be continuous. In order to guarantee convergence, *relaxation*-based simulators require that a capacitor be connected between every node and ground. Certain numerical methods are more appropriate for *loosely-coupled*[4] MOS devices, while others

2. If floating point calculations are used to trigger events, the function is deterministic only if the same floating point format is used from one simulation to the next.

3. In other words, the underlying mathematics guarantee that each step in the approximation is within a specified error tolerance, however, there is no way to know how these errors will accumulate over an entire simulation run.

work better for circuits using *tightly-coupled* bipolar devices. Choosing the best simulation method for a given circuit topology and technology is still an art, not a science.

1.4.2.3 Digital Structure vs. Analog Structure

The structural aspects between digital and analog systems share the most similarity. Both can express hierarchy. Both are represented as a network of interconnected components. VHDL networks, however, are directed, while analog networks are not. Connecting two analog elements together represents a mutually-shared contribution to each other's behavior. Analog networks are thus tightly coupled, while VHDL networks are highly decoupled. The connection of digital elements via signals can be thought of as an assembly line, while the connection between analog elements is more like a marriage.

VHDL	Analog
Elements are connected via unidirectional ports.	Elements are connected via directionless terminals.

Table 3: Structural Aspects

1.4.2.4 Analog-Digital Interfacing

The question then arises: *How do digital elements communicate with analog elements within a mixed-mode environment?* One proven method[13][14] allows analog elements to generate events in response to a change in their internal state, usually the crossing of a threshold. Digital elements which are sensitive to the generated event are, in turn, activated and affected. Analog elements can similarly be sensitive to events, whether generated by analog elements or digital elements, causing event-driven behavior modification of the analog system. In this paradigm, the burden of analog event generation and event sensitivity rests within the analog elements.

A proposed alternate scenario would be to allow digital elements to read analog quantities at specific times. These values could then be tested, and appropriate action taken. This scenario has a two undesirable consequences:

1. It would force unnecessary synchronization with the analog state, impacting simulation performance with no gain in timing accuracy.
2. Analog values, which have dual characteristics (voltage and current), are incompatible with multi-valued logic states. Analog thresholding would still be necessary, but this would then occur within digital blocks, introducing mixed-mode behavior in an otherwise pure-digital section.

1.4.2.5 The Simulation Cycle

The following description is taken from the IEEE 1076 restandardization draft of 1992.Changes to this text to support analog are shown <u>underlined</u>.

The execution of a model consists of an initialization phase followed by the repetitive execution of process statements in the description of that model. Each such repetition is said to be a *simulation cycle.* In each

4. Loosely-coupled devices are devices in which the inputs do not have dependent relationships on the outputs, such as MOS transistors. Tightly-coupled devices have mutual dependencies between inputs and outputs, as in the case of bipolar junction transistors (BJTs).

cycle, the values of all signals in the description are computed. If as a result of this computation an event occurs on a given signal, process statements that are sensitive to that signal will resume and will be executed as part of the simulation cycle.

At the beginning of initialization, current time, T_c, is assumed to be 0 ns.

The initialization phase consists of the following steps:

1. The values of all node voltages and branch currents are set to those calculated by dc analysis except for those node voltages and branch currents which have been given an initial condition..
2. If the value of any analog system variable has crossed a treshold upon which a process is sensitive, execute the process. Any changes to digital signals become the values to be used for step 3.
3. The driving value and the effective value of each explicitly declared signal are computed, and the current value of the signal is set to the effective value. This value is assumed to have been the value of the signal for an infinite length of time prior to the start of simulation.
4. The value of each implicit signal of the form S'Stable(T) or S'Quiet(T) is set to True. The value of each implicit signal of the form S'Delayed(T) is set to the initial value of its prefix, S.
5. The value of each implicit GUARD signal is set to the result of evaluating the corresponding guard expression.
6. Each process in the model is executed until it suspends.
7. If no analog values are sensitive to signal which changed as a result of steps 1 through 6, the system has become quiescent therefore proceed to step 8. Otherwise, determine of the ocillation count has been exceeded. If it has report ocillation and exit. Otherwise loop back to step 2.
8. The time of the next simulation cycle (which, in this case is the first simulation cycle), T_n, is calculated according to the rules of step 7 of the simulation cycle, below.

A simulation cycle consists of the following steps:

1. The current time, T_c is set equal to T_n. Simulation is complete when T_n = TIME'HIGH and there are no active drivers or process resumptions at T_n.
2. Each active explicit signal in the model is updated. (Events may occur on signals as a result.)
3. Each implicit signal in the model is updated. (Events may occur on signals as a result.)
4. For each process P, if P is currently sensitive to a signal S, and an event has occurred on S in this simulation cycle, then P resumes.
5. Each non-postponed process that has resumed in the current simulation cycle is executed until it suspends.
6. The analog system is evaluated at time Tn...
7. The time of the next simulation cycle, T_n, is determined by setting it to the earliest of:
   a. TIME'HIGH,
   b. the next time at which a driver becomes active, or
   c. the next time at which a process resumes.

   If $T_n = T_c$, then the next simulation cycle (if any) will be a delta cycle.
8. If the next simulation cycle will be a delta cycle, the remainder of this step is skipped. Otherwise, each postponed process which has resumed but has not been executed since its last resumption is

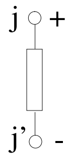
executed until it suspends, then T_n is recalculated according to the rules of step 6. It is an error if the execution of any postponed process causes a delta cycle to occur immediately after the current simulation cycle.

1.4.2.6 Formulation of Network Equations

There are essentially two methods used to formuate network equations: tableau form and modified nodal form. Tableau form is mainly of theoretical importance. It generates large matrices even for very small problems. They are very sparse, and therefore sparse solvers are a necessity. Unfortunately, since the matrices do not have regular structures, the renumbering and preproccessing is complicated. Modified nodal formautions are much more conmpact and can be solved without relying on sparse techniques even in the case of moderate-size networks.

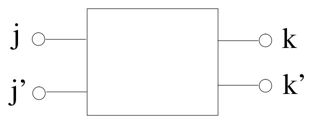
In the case of modified nodal form, each idelized component has an associated stamp. The stamp represents a matrix of the energy conservation equations which describe the behavior of the particular element. The following list defines the stamps for the fundemental elements of electrical circuits. The canonical form of the stamp derived using single-graph modified normal form.

Generic two-terminal element



current flows through j to j'

Generic four-terminal element



current flows through j to j' and through k to k'.

Current Source

$$\begin{array}{c} j \\ j' \end{array} \begin{bmatrix} \boldsymbol{J} \\ -J \end{bmatrix}$$

where J is the amount of current through j->j'. Equations:

$I_j = J$

$I_{j'} = -J$

Voltage Source

$$\begin{array}{r} j \\ j' \\ m+1 \end{array} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 1 & -1 & 0 \end{bmatrix} \times \begin{bmatrix} 0 \\ 0 \\ E \end{bmatrix}$$

where E is the amount of voltage across j and j'. Equations:

$V_j - V_{j'} = E$

$I_j = I$

$I_{j'} = -I$

Admittance

$$\begin{array}{c} j \\ j' \end{array} \begin{bmatrix} y & -y \\ -y & y \end{bmatrix}$$

where y is the admittance of the node (admittance is one over the impedence) between j and j'. Equations:

$I_j = y(V_j - V_{j'})$

$I_{j'} = -y(V_{j-}V_{j'})$

Impedence

$$\begin{array}{r} j \\ j' \\ m+1 \end{array} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 1 & -1 & -z \end{bmatrix}$$

where z is the impedence between the nodes j and j'. Equations:

$V_j - V_{j'} - zI = 0$

$I_j - = I_{j'} = I$

Capacitor

$$\begin{array}{c} j \\ j' \end{array} \begin{bmatrix} sC & -sC \\ -sC & sC \end{bmatrix}$$

where sC is the time derivative of the charge stored in the capacitor. Equations:

$I_j = sC$

$I_{j'} = -sC$

Inductor

$$\begin{array}{c} j \\ j' \\ m+1 \end{array} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ 1 & -1 & -sL \end{bmatrix}$$

where sL is the time derivative of the field flux of the inductor. Equations:

$I_j = I$

$I_{j'} = -I$

$V_j - V_{j'} = sL$

Voltage Controlled Current Source

$$\begin{array}{c} k \\ k' \end{array} \begin{bmatrix} g & -g \\ -g & g \end{bmatrix}$$

where g is the function relating the voltage across j and j' to the current through k->k'. Equations:

$I_j = g(V_j - V_{j'})$

$I_{j'} = g(V_j - V_{j'})$

Voltage Controlled Voltage Source

$$\begin{array}{c} j \\ j' \\ k \\ k' \\ m+1 \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 \\ u & -u & 1 & -1 & 0 \end{bmatrix}$$

where u is the function relating the voltage across j and j' to the voltage across k and k'. Equations:

$-uV_j + uV_{j'} + V_k - V_{k'} = 0$

$-V_{k'} = 0$

$I_k = I$

$I_{k'} = -I$

Current Controlled Voltage Source

$$\begin{array}{c} j \\ j' \\ k \\ k' \\ m+1 \\ m+2 \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & -r & 0 \end{bmatrix}$$

where r is the function relating the current through j->j' to the voltage across k and k'. Equations:

$V_j - V_{j'} = 0$

$V_k - V_{k'} - rI_1 = 0$

$I_j = -I_{j'} = I_1$

$I_k = -I_k = I_2$

Current Controlled Current Source

$$\begin{array}{c} j \\ j' \\ k \\ k' \\ m+1 \end{array} \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & a \\ 0 & 0 & 0 & 0 & -a \\ 1 & -1 & 0 & 0 & 0 \end{bmatrix}$$

where a is the function relating currrent through j->j' to the current through k->k'. Equations:

$V_j - V_{j'} = 0$

$I_j = -I_{j'} = I$

$I_k = -I_{k'} = aI$

When a circuit element like the ones above is encountered in a network, the stamp for that element is placed within the system matrix in a location which is determined by the circuit topology.

1.4.2.7 Simulator-specific extensions

To be specified.

1.4.3 AHDLs vs. CSSL

The acronym CSSL stands for Continuous System Simulation Language[15], and, as the name implies, is only useful for describing continuous systems. CSSL uses control-system (signal-flow) semantics to define the interactions between elements of the system. Control systems are an abstraction of analog systems. Control systems are not required to implicitly obey any physical conservation laws like KVL and KCL. The relationships between conserved properties like voltage and current can be expressed in CSSL, but they must be done so explicitly. One advantage of an AHDL over CSSL is that these physical relationships are handled implicitly, reducing the complexity of model to just the essential equations. The other advantage is, obviously, the ability to model *discontinuous* systems with an AHDL

Discontinuities are common in electrical systems, or in physical systems in general, for that matter. Typically, an electronic component, such as a transistor, will have several "regions of operation". The behavior of the transistor within each region is determined by unique set of equations for that region. When the device crosses from one region to another this is said to be a *discontinuity.* Since this is such a common phenomenon in nature, it seems unreasonably restrictive to disallow the expression of this kind of behavior. That is why an AHDL does not. It makes the implementation of the underlying simulator much more difficult, but it a least allows the description of "real-world" behavior.

The following is an example of a CSSL program which models a lunar landing:

```
PROGRAM Lunar Landing Maneuver
   INITIAL
      r=1738.0E3, c2=4.925E12, f1=36350.0,
      f2=1308.0, c11=0.000277, c12=0.000277,
      h0=59404.0, v0=-2003.0, m0=1038.358,
      tmx=230.0, tdec=43.2, tend=210.0
   END $"of INITIAL"
   DYNAMIC
      DERIVATIVE
         thrust  = (1.0-step(tend))*(f1-(f1-f2)*step(tdec))
         c1      = (1.0-step(tend))*(c11-(c11-c12)*step(tdec))
         h       = integ(v,h0)
         v       = integ(a,v0)
         a       = (1.0/m)*(thrust-m*g)
         m       = integ(mdot, m0)
         mdot    = -c1*abs(thrust)
         g       = c2/(h+r)**2
      END $"of DERIVATIVE"
      termt(t .ge. tmx .or. h .le. 0.90 .or. v .gt. 0.0)
   END $"of DYNAMIC"
END $"of PROGRAM"
```

Exmaple 3: Luner Landing Programs in CSSL

In the example, the simulation is terminated when the boolean expression found in the `termt` statement becomes true, otherwise the simulation would continue indefinitely. This termination strategy works fine for relatively trivial sets of termination conditions, such as exceeding a time limit. However, when a complex system (i.e., more than one element) is simulated, it is generally unwise to specify termination conditions based on time as part of the model. Obviously, if the system is going to simulate as a whole, all of the time limits must agree, which could be difficult in a systems with thousands of elements. Alternatively, the time end-point could be specified as a parameter to the simulator by the user, instead of being declared within a general model.

As given, the lunar landing model would generate a system of three differential equations, five algebraic equations, and eight unknowns. Since the system contains differential equations, it must be integrated to find a solution for any point in time. The initial condition for each integral are given as the second argument to the `integ` function. Notice the variables `thrust` and `c1` each have three states, depending on the value of time. It is interesting to discover that minimal system formed by this example would only have four equations. The necessary variables in the minimal system are `h`, `v`, `a`, and `m`. The rest can be derived by substitution. CSSL does not make this distinction, which results in unnecessarily large systems and correspondingly slower simulations.

It is a requirement of an AHDL to allow the expression of this variable property. If written in the AHDL MAST®, the previous example generates a system of equations which is only four by four (see Newton's laws of motion below):

```
template lander
{
   number  r=1738.0E3, c2=4.925E12, f1=36350.0,
           f2=1308.0, c11=0.000277, c12=0.000277,
           h0=59404.0, v0=-2003.0, m0=1038.358,
           tmx=230.0, tdec=43.2, tend=210.0

   g = c2/(h+r)**2                            # gravity
   mdot = -c1*abs(thrust)                     # lose fuel when firing

   if ( time < tdec ) {                       # pedal to the metal!
      thrust = f1
      c1 = c11
   }
   else if ( time >= tdec & time < tend ) {   # cool our jets...
      thrust = f2
      c1 = c12
   }
   else if ( time >= tend ) {                 # shut'em down...
      thrust = 0.0
      c1 = 0.0
   }
   when ( h <= 0.90 | v > 0.0 )               # bail out if we crash
      halt_simulation(0)                      # or start going up!

   d_by_dt(h)       = v                       # Newton's
   d_by_dt(v)       = a                       #    Laws of
   a                = (1.0/m)*(thrust-m*g)    #       motion...
   d_by_dt(m)       = mdot                    # mass

   control_section {
      initial_condition(h, h0)
      initial_condition(v, v0)
      initial_condition(m, m0)
   }
}
```

Exmaple 4: Lunder Lander in MAST

Notice that the three "regions of operation" for the lander are expressed much more directly by using the if-then-else structures available in MAST®. Also, in MAST® it is possible, but not required, to declare the units of measurement applied to each variable. In the template above, we might have given all of the system variables a units attribute which makes the simulation output more meaningful. Furthermore, the attributing of units to variable allows the simulator to check the dimensional consistency of the variable usage (e.g. meters to meters, velocity to velocity), eliminating some of the most common modeling errors. Consider the following possible declaration of the system variable used in the example above:

```
var meters    h                           # altitude
var mps       v                           # velocity
var mpss      a                           # acceleration
var kg        m                           # spaceship mass

val kgs       mdot                        # rate of weight loss
val newtons   g, thrust, c1               # gravity, ...
```

Exmaple 5: System Variables Declared with Units Attatched

Just as in the CSSL version, discontinuities are introduced into the system at times `tdec`, `tend`, and `tmx`. The model is, however, unnecessarily inefficient. Note the each of the three tests to detect the current region of operation must be performed on *every simulation cycle*. This is necessary although most of the time the region of operation never changes. Instead of checking for this at every time step, digital events can be used to trigger the change. The following model schedules events to trigger these actions. On a mixed-mode simulator it would execute appoximately 10 times faster than the preceding analog-only model would execute. This underscores the necessity of providing event-driven behavior in an analog language.

```
template lander
{
   number  r= 1738.0E3, c2=4.925E12, f1=36350.0,
           f2=1308.0, c11=0.000277, c12=0.000277,
           h0=59404.0, v0=-2003.0, m0=1038.358,
           tmx=230.0, tdec=43.2, tend=210.0

   state nu decid, endid
   state n thrust, c1

   mdot = -c1*abs(thrust)              # get lighter as we burn fuel
   g = c2/(h+r)**2                     # gravitational pull

   when ( time_init ) {
      thrust = f1                      # pedal to the metal!
      c1 = c11
      schedule_event(tdec, decid, 1)   # schedule thrust reduction
      schedule_event(tend, endid, 1)   # schedule thrust shut off
   }
   when ( event_on(decid) ) {          # cool our jets...
      thrust = f2
      c1 = c12
      schedule_next_time(time)         # re-evaluate equations now
   }
   when ( event_on(endid) ) {          # shut'em down...
      thrust = 0.0
      c1 = 0.0
      schedule_next_time(time)         # re-evaluate equations now
   }
   when ( h <= 0.90 | v > 0.0 ) {      # bail out if we crash
      halt_simulation(0)               # or start going up!
   }
   d_by_dt(h)= v                        # Newton's
   d_by_dt(v) = a                       #     Laws of
   a        = (1.0/m)*(thrust-m*g)      #        motion...
   d_by_dt(m)= mdot                     # mass

   control_section {
      initial_condition(h, h0)
      initial_condition(v, v0)
      initial_condition(m, m0)
   }
}
```

Exmaple 6: Mixed-Mode Lunar Lander in MAST

The following graphs were generated were generated by the Saber® simulator using the models given above. Both models were simulated and the resulting waveforms plotted against each other. As is shown by the plots, the waveforms are identical, demonstrating the accuracy of the mixed-mode simulation versus the pure analog simulation.

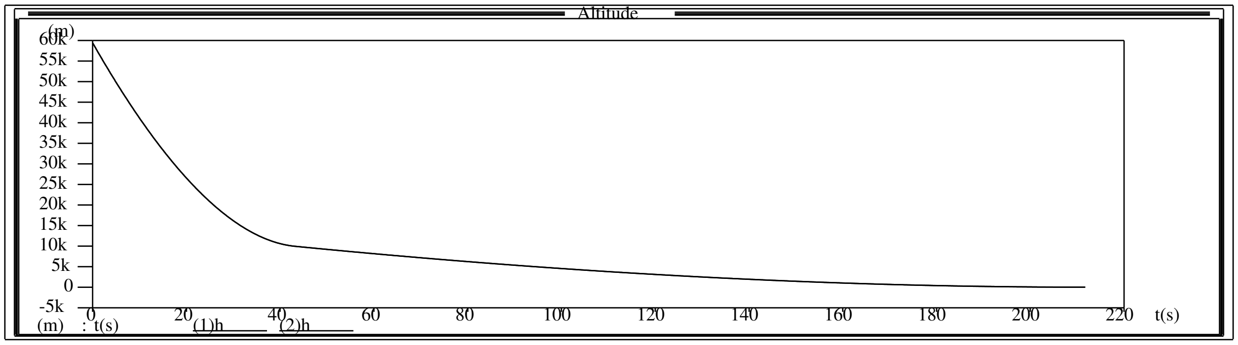


Fig. 6: Plot of Spaceship Altitude

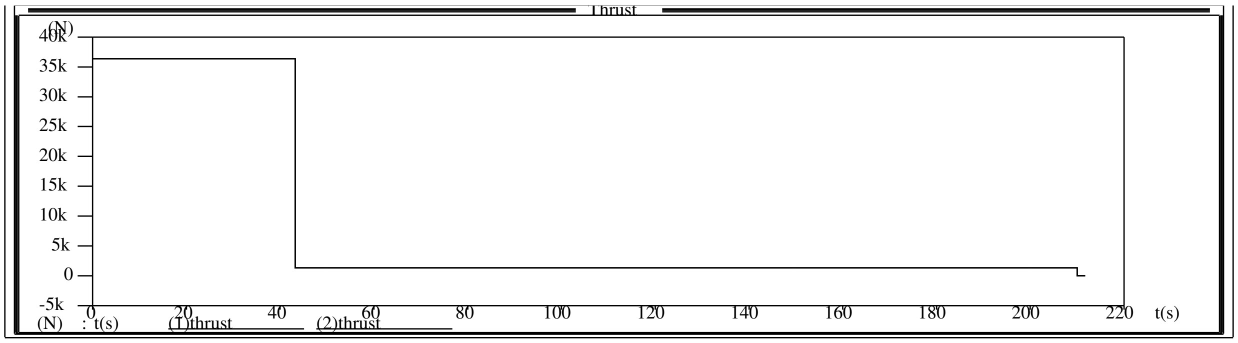


Fig. 7: Plot of Spaceship Thrust

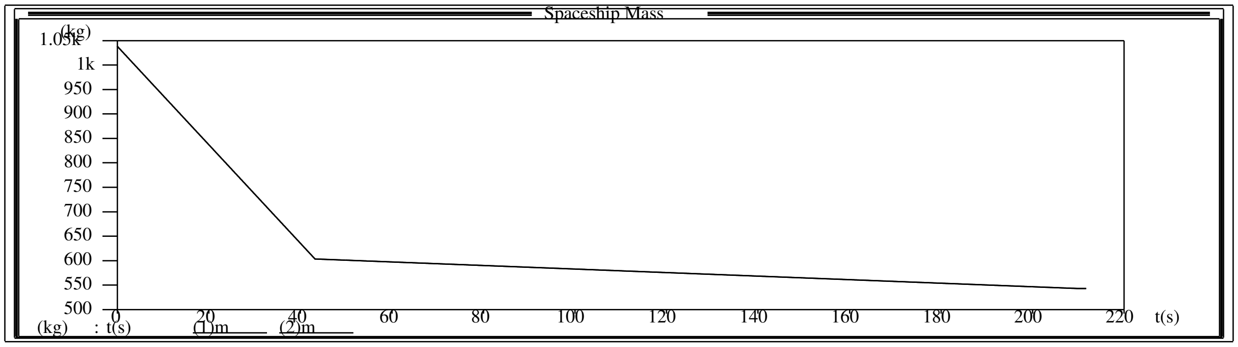


Fig. 8: Plot of Spaceship Mass

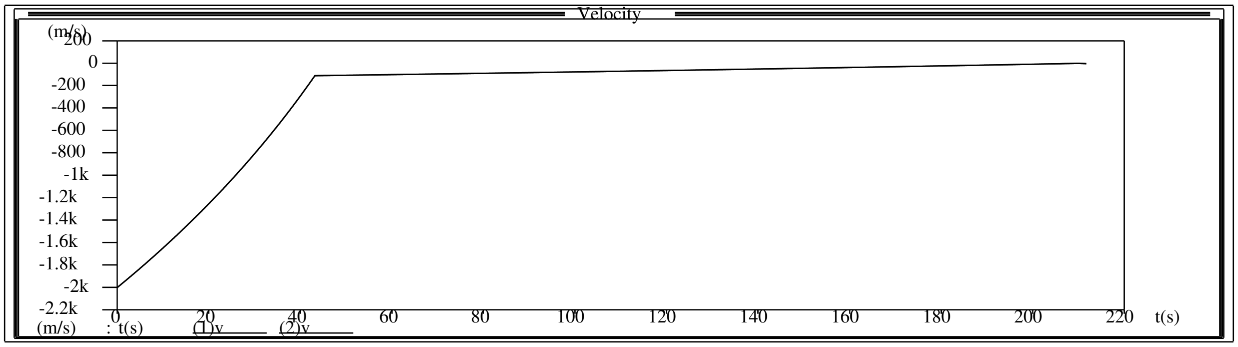


Fig. 9: Plot of SpaceshipVelocity

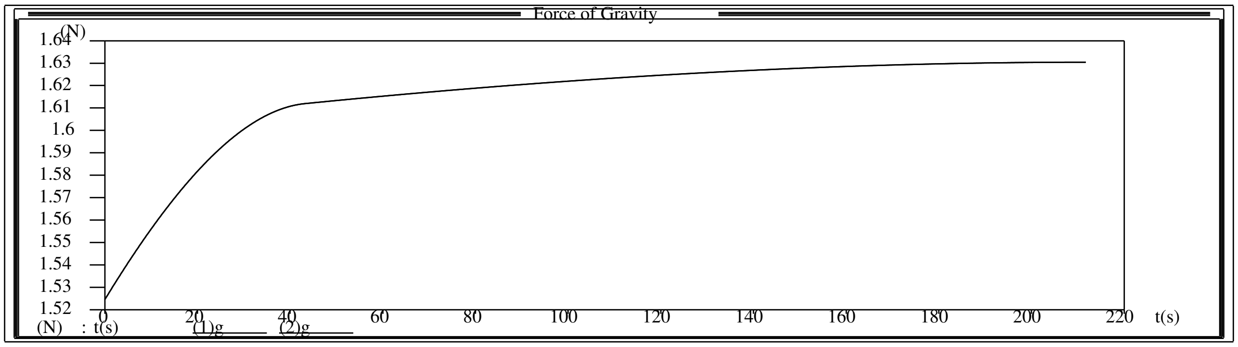


Fig. 10: Plot of the Force of Gravity

Since control systems can be considered an abstraction of analog systems, an ADHL can be used to model them, as is demonstrated in the previous example. An AHDL can be thought of as a true functional superset of control system languages like CSSL.

2.0 A Look at Existing HDLs and How They Evolved

2.1 Evolution of languages

The following diagram (see Fig. 11) represents the applicability and heritage of many popular hardware description languages and/or simulators. It is by no means complete. The placement along the timeline indicates the date which the language or simulator was first in use, or is proposed for release. All languages below 1992 are in active use. The arrows represent lineage.

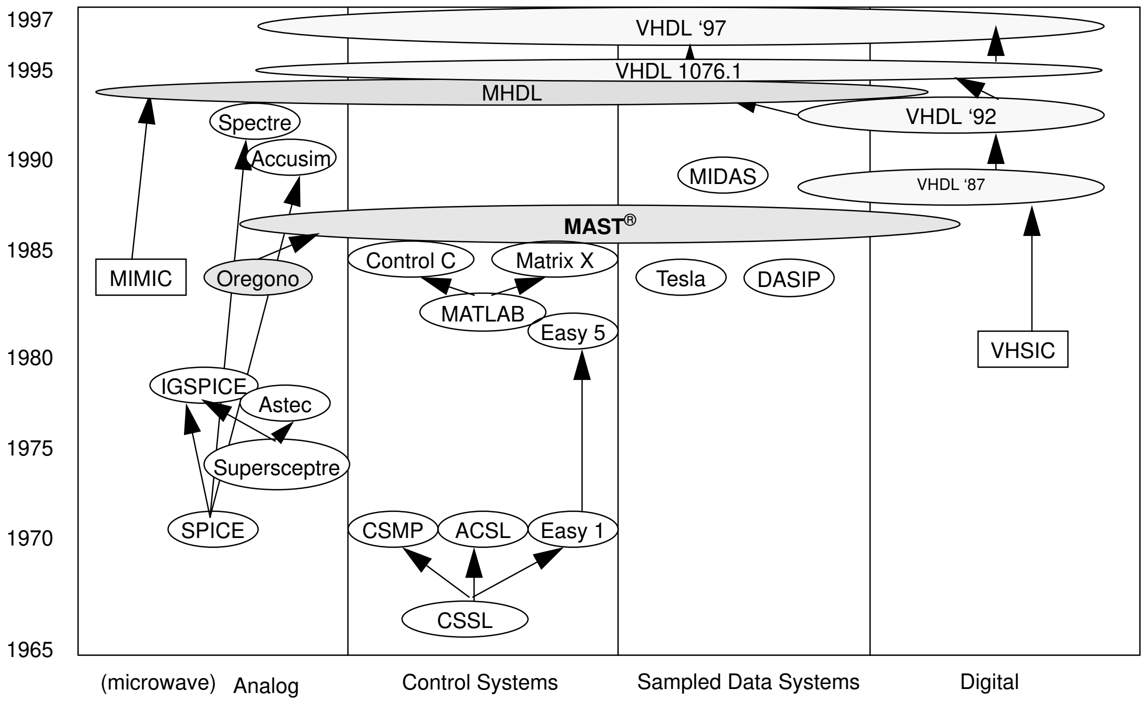


Fig. 11: Evolution of HDLs

What becomes immediately apparent when looking at the above diagram is that the new and emerging hardware description languages are becoming less specialized. In 1986, for instance, MAST spanned the application spectrum of several highly-specialized languages. Between VHDL 1076.1 and MHDL, when they become available (at least 2 years out), the entire application spectrum will be (eventually) covered. This reflects the current trend toward standardization and away from proprietary language that we have witnessed in the past decade, and which continues to dominate future HDL development.

2.2 Sample AHDL models

The following section presents sample models of the most basic elements of electrical circuits: a resistor, a capacitor, an inductor, a diode, and a switch. The transistor, as noted previously, can be modeled as a configuration of the first four elements. The switch is included to give an example of discontinuous behavior. In all cases, only ideal behavior is represented. More comprehensive models would include parasitic capacitances, noise sources, terminal resistances, thermal effects, and the like.

The language constructs used for the VHDL examples are **VERY PRELIMINARY**. The examples are intended to demonstrate required functionality, not the form it will take.

From studying the examples, it becomes apparent that several new constructs have been used. Firstly, analog entities are connected by `PIN`s, not `PORT`s. Pins are the same as terminals in *n*-port networks[16]. All devices must have at least two terminals (pins). The pins have a voltage across them and current flowing through them. These are known as *branch* voltages and *branch* currents, respectively. `PIN`s can only be connected to other `PIN`s, or to the reference node (typically ground), but never to `PORT`s. All of the examples presented below are two-pin devices. Transistors are three-pin or four-pin devices.

Secondly, the constructs `p.v` and `p.i` represent new functionality. This syntax is used to access the *through* or *across* quantities associated with a pin. In this case, and for electrical systems in general, the *through* quantity is current(`i`) and the *across* quantity is voltage(`v`). Further constructs introduced into the language will be explained after each example.

2.2.1 Resistor

The behavior of an ideal resistor is given by Ohm's Law, namely:

$$v = i \bullet r$$

Written in MAST, the model would look like:

```
template resistor p m = r
   electrical p, m
   number r                  # resistance
{
   branch v = v(p,m)         # voltage across pins
   branch i = i(p->m)        # current through capacitor
   v = i * r                 # Ohm's Law
}
```

Exmaple 7: Resistor in MAST

Written in VHDL, the model might be:

```
ENTITY resistor IS
   GENERIC ( r: REAL );
   PIN ( p, m : electrical );
END resistor;

ARCHITECTURE resistor_equ OF resistor IS
BEGIN
   EQUATION
      branch v: voltage(p,m);
      branch i: current(p,m);
   BEGIN
      v = i*r;               -- Ohm's Law
   END EQUATION;
END resistor_equ;
```

Exmaple 8: Resistor in VHDL

In the above example, the three statements comprising the body of the entity are equations specifying relationships, not assignments. The first equation states that the voltage across the resistor is equal to the difference between the voltages (potential) at each pin. The second equation states that the current flowing through the resistor is equal to the current flowing from pin `p` to pin `m`. These first two equations are known as *branch* equations, since they identify branch currents and voltages. The last equation is an application of Ohm's Law; namely, the voltage across a resistor is equal to the current through it, multiplied by its resistance.

2.2.2 Capacitor

The behavior of a linear capacitor is given by the equation:

$$i = C \bullet \frac{dv}{dt}$$

Written in MAST, the model would look like:

```
template capacitor p m = c
   electrical p, m
   number c              # capacitance
{
   branch v = v(p,m)     # voltage across pins
   branch i = i(p->m)    # current through capacitor
   q = c*v               # charge stored in capacitor
   i = d_by_dt(q)        # current is derivative of charge
}
```

Exmaple 9: Capacitor in MAST

Written in VHDL, the model might be:

```
ENTITY capacitor IS
   GENERIC ( c: REAL );
   PIN ( p, m : electrical );
END capacitor;

ARCHITECTURE capacitor_equ OF capacitor IS
BEGIN
   EQUATION
      branch v: voltage(p,m);
      branch i: current(p,m);
      variable q: charge;
   BEGIN
      q = c*v;           -- charge stored in cap
      i = d_by_dt(q);    -- current is derivative of charge
   END EQUATION;
END capacitor_equ;
```

Exmaple 10: Capacitor in VHDL

This example introduces a new function `d_by_dt()`, or time derivative. Using this function, it is possible to specify ordinary differential equations as part of model behavior. Thus, the last equation states that the current through the capacitor is equal to the change in the charge contained within it, with respect to time.

2.2.3 Inductor

The behavior of an ideal inductor is given by the equation:

$$v = L \bullet \frac{di}{dt}$$

Written in MAST, the model would look like:

```
template inductor p m = l
   electrical p, m
   number l              # inductance
{
   branch v = v(p,m)     # voltage across pins
   branch i = i(p->m)    # current through capacitor
   v = d_by_dt(l*i)      # v is time derivative of flux
}
```

Exmaple 11: Indcutor in MAST

Written in VHDL, the model might be:

```
ENTITY inductor IS
   GENERIC ( l: REAL );
   PIN ( p, m : electrical );
END inductor;

ARCHITECTURE inductor_equ OF inductor IS
BEGIN
   EQUATION
      branch v: voltage(p,m);
      branch i: current(p,m);
      variable f: flux;
   BEGIN
      f = l * i;          -- flux is current times inductance
      v = d_by_dt(f);     -- voltage is derivative of flux
   END EQUATION;
END inductor_equ;
```

Exmaple 12: Inductor in VHDL

This example is similar to one given for a capacitor. Notice that the *dependent* and *independent* variables have been exchanged. With the capacitor, the current was dependent upon the voltage. Now, with the inductor, the voltage is dependent on the current. This limits the application of KCL, which requires that currents be dependent on voltages. The equation could be rewritten to express the branch current through the inductor as a function of the integral of the branch voltage across its terminals. Integrals, however, present additional problems when dealing with initial conditions.

2.2.4 Diode

The behavior of a real world diode is given by the equation:

$$i = Is \bullet \left[e^{\frac{v}{vt}} - 1 \right]$$

Written in MAST, the model would look like:

```
template diode p m = vt, is
   electrical p, m
   number vt, is              # diode characteristics
{
   branch v = v(p,m)          # voltage across pins
   branch i = i(p->m)         # current through capacitor
   i = is*(exp(v/vt)-1)       # i is the exponential of v
}
```

Exmaple 13: Diode in MAST

Written in VHDL, the model might be:

```
ENTITY diode IS
   GENERIC ( vt, Is : REAL );
   PIN ( p, m : electrical );
END diode;

ARCHITECTURE diode_equ OF diode IS
BEGIN
   EQUATION
      variable v: voltage(p,m);
      variable i: current(p,m);
   BEGIN
      i = Is*(exp(v/vt)-1);    -- i is exponential of v
   END EQUATION;
END diode_equ;
```

Exmaple 14: Diode in VHDL

This example introduces the `exp()` function, implementing e^x. The diode equation is one example of the many kinds of nonlinear behavior found in electrical systems. All semiconductor devices exhibit nonlinear behavior. Even more so, certain devices cannot have their behavior modeled as a continuous function of time. Their behavior is, therefore, said to be a *discontinuous* function of time. Real world examples include the firing of a spark-plug, or even the turning on or off of a switch. This kind of modeling problem can best be described using the if-then-else constructs already available in VHDL.

2.2.5 Simple Switch

Consider the following example of an ideal single-pole-single-throw (SPST) switch:

```
template switch p m = pos, onres, offres
   electrical p, m
   number pos                 # switch position
   number onres               # on resistance
   number offres              # off resistance
{
   branch v = v(p,m)          # voltage across pins
   branch i = i(p->m)         # current through diode
   if ( pos == 1 )            # if switch closed
      v = i*onres             # then voltage drop is fcn of onres
   else                       # otherwise switch is open
      i = v/offres            # and little current is flowing
}
```

Exmaple 15: Switch in MAST

```
ENTITY switch IS
   GENERIC (  pos : INTEGER;
              onres, offres: REAL
           );
   PIN ( p, m : electrical );
END switch;

ARCHITECTURE switch OF switch IS
BEGIN
   EQUATION
      variable v: voltage(p,m);
      variable i: current(p,m);
   BEGIN
      IF ( pos = 1 ) THEN          -- if switch closed
         v = i*onres;              -- then voltage drop is fcn of onres
      ELSE                         -- otherwise switch is open
         i = v/offres;             -- and little current is flowing
      END IF;
   END EQUATION;
END switch;
```

Exmaple 16: Switch in VHDL

Of course, in this example, the position of the switch is passed in as a GENERIC, and so its effect on the analog system is known at elaboration time. More generally, the position could be a signal, causing the equations to change dynamically during simulation. It should be noted that using if-then-else constructs around equations introduces alternate declarations of the analog system. This is quite different from using if-then-else in a process, which changes the execution path within that process.

Appendix A
References

[1] L. W. Nagel, “SPICE2 - A computer program to simulate semiconductor circuits,” *Electronics Research Laboratory Rep. No. ERLM520*, University of California, Berkeley, 1975.

[2] T. Quarles, “Adding devices to SPICE3,” *Electronics Research Laboratory Rep. No. ERL-M89/47*, University of California, Berkeley, April 1989.

[3] D. B. Herbert, “Simulating differential equations with SPICE2,” *IEEE Circuits and Devices*., vol. 8, no. 1, pp. 11-14, Jan 1992.

[4] Roger Lipsett, Carl Schaefer, Cary Ussery, *VHDL: Hardware Description and Design*, Kluwer Academic Publishers; 1989.

[5] Resve A. Saleh, A. Richard Newton, *Mixed-Mode Simulation*, Kluwer Academic Publishers; 1990.

[6] Jiri Vlach, Kishore Singhal, *Computer Methods for Circuit Analysis and Design*, Van Norstrand Reinhold; 1983.

[7] Charles Belove, *Handbook of Modern Electronics and Electrical Engineering*, John Wiley & Sons; 1986.

[8] I. Getreu, *Modeling the Bipolar Transistor*, Elsevier Scientific Publishing Company, Amsterdam-Oxford-New York; 1978.

[9] J. J. Ebers and J. L. Moll, “Large-Signal Behavior of Junction Transistors”, *Proc. IRE*, Vol 42, pp. 1761-1772, December 1954.

[10] H. K. Gummel and H. C. Poon, “An Integral Charge Control Model of Bipolar Transistors”, *Bell Systems Technical Journal*, Vol. 49, pp. 827-852, May 1970.

[11] N. B. Rabbat, A. L. Sangiovanni-Vencentelli, and H. Y. Hsieh, “A Multilevel Newton Algorithm with Macromodeling and Latency for Analysis of Large-scale Nonlinear Networks in the Time Domain,” *IEEE Transactions on Circuits and Systems*, vol. CAS-26; 1979.

[12] C. W. Gear, “Automatic Multirate Methods for Ordinary Differential Equations,” *Information Processing 80*, International Federation of Information Processing; 1980.

[13] Martin Vlach, “Modeling and Simulation with Saber,” *Proceedings from The Third Annual IEEE ASIC Seminar and Exhibit*; pg. T-11.1, Sept. 1990, Rochester, NY.

[14] H. A. Mantooth, Martin Vlach, “Beyond SPICE with Saber and MAST,” *Proceedings from The IEEE International Symposium on Circuits and Systems*; pp. 77-80, May 10-13, 1992, San Diego, CA.

[15] Donald C. Augustin, Mark S. Fineberg, Bruce B. Johnson, Rober N. Linebarger, F. John Sansom, and Jon C. Strauss, “The SCi Continuous System Simulation Language (CSSL)”, *Simulation*, Vol. 9, pp. 281-303, 1967.

[16] Leon O. Chua, Charles A. Desoer, Ernest S. Kuh, *Linear and Nonlinear Circuits*, Chap. 13, McGraw-Hill; 1987.