

The Probe/Source Approach to Analog Modeling

*Ken Kundert
Designer's Guide Consulting
Los Altos, California
ken@designers-guide.com*

June 2, 2006

1 Introduction

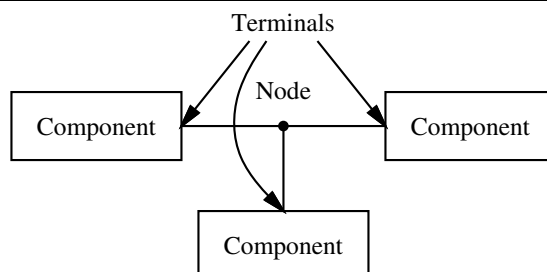
This paper presents a variety of topics that center on how users of AHDLs might formulate conservative and signal-flow component models, and how a simulator might combine the model equations in a way that the resulting set of equations could be solved by conventional numerical techniques. The presentation is largely conceptual. Details of syntax are specifically avoided.

The paper starts by describing the important features of both conservative and signal-flow systems. It is then shown that it is possible to interpret conservative systems as being a generalization of signal-flow systems. This interpretation allows users to arbitrarily combine conservative and signal-flow components without losing any of the benefits of either. It then discusses the issues of multidisciplinary modeling and their impact on tolerances. Finally, an example is presented that demonstrates how the set of equations would be formulated for a mixed conservative/signal-flow system, how uninteresting equations can be eliminated in order to reduce the equations to a reasonable number, and how the tolerances given for user-specified unknowns and equations can be propagated to the additional unknowns and equations needed by the simulator to completely describe the system.

2 Systems

Systems are taken to be a collection of interconnected individual components that are acted upon by a stimulus and produce a response. The components themselves might also be systems, otherwise

Figure 1 — Components connect to nodes through terminals.



they are primitive components. For the purposes of this paper, it is assumed that components do not connect directly to each other, rather they connect to nodes and that more than one component can connect to the same node. Components connect to nodes through terminals as shown in Figure 1.

In order to simulate systems, it is necessary to have a complete description of the system and all of its components. Descriptions of systems are given structurally. That is, the description of a system contains a list of components and how they are interconnected. Descriptions of primitive components are given behaviorally. That is, a mathematical description is given that relates the signals at the terminals.

2.1 Conservative Systems

An important characteristic of conservative systems is that there are two values associated with every node (and hence every terminal) — the potential (also known as the across value, or the voltage in electrical systems) and the flow (the through variable, or the current in electrical systems). The potential of the node is shared with all terminals connected to the node in such a way that all terminals see the same potential. The flow is shared such that flow from all terminals at a node must sum to zero. In this way, the node acts as an infinitesimal point of interconnection in which the potential is the same everywhere on the node and on which no flow can accumulate. Thus, the node embodies Kirchhoff's Potential and Flow Laws (KPL and KFL). When a component connects to a node through a conservative terminal, it may either affect, or be affected by, either the potential at the node, and/or the flow onto the node through the terminal.

With conservative systems it is also useful to define the concept of a branch. A branch is a path of flow between two nodes through a component. Every branch has an associated potential (the potential difference between the two nodes) and flow.

A behavioral description of a conservative component is constructed as a collection of interconnected

branches. The constitutive equations of the component are formulated as to relate the branch potentials and flows. In the probe/source approach, the branch potential or flow is specified as a function of branch potentials and flows. If the branch potential and flow are left unspecified, then the branch acts as a probe. In this case, if the branch flow is used in an expression, the branch potential is forced to zero. Otherwise the branch flow is assumed to be zero and the branch potential is available for use in an expression. Using both the potential and flow of a ‘probe’ branch in an expression is not allowed. Nor is specifying both the branch potential and flow at the same time. (While these last two conditions are not really necessary, they do eliminate conditions that are useless and confusing.)

2.2 Signal-Flow Systems

Unlike conservative systems, signal-flow systems only have one value associated with every node. As a result, a signal-flow terminal must be unidirectional. It may either read the value of the node, or it may specify it. Signal-flow terminals are either considered input terminals if they pass the value of the node into a component, or output terminals if they specify the value of a node.

Signal-flow terminals support a subset of the functionality of conservative terminals. As such, one can always use conservative semantics to represent signal-flow components. There are, however, two important benefits that result from allowing direct description of signal-flow components using signal-flow semantics. First, one only need declare the types of signals that one intends to use. Conversely, one need not declare the types of signals that are not used and therefore for which one would have no basis upon which to make a choice of what the signal type should be. Second, signal-flow semantics require a smaller number of equations and unknowns, and so results in a formulation that is more efficient to simulate.

There are some restrictions that are typically present in signal-flow formulations. For example,

1. Typically, one cannot directly interface signal-flow and conservative components.
2. Typically, signals are potential-like, making it difficult to represent flow-like signals.
3. Typically, components descriptions can only be written in terms of ground-referred signals, making it difficult to write descriptions of components that use floating or differential signals.

2.3 Mixed Conservative/Signal-Flow Systems

When practicing the top-down design style, it is extremely useful to mix conservative and signal-flow components in the same system. Users typically use signal-flow models early in the design

cycle when the system is described in abstract terms, and gradually convert component models to conservative form as the design progresses. Thus, it is important to be able to initially describe a component using a signal-flow model, and later convert it to a conservative model, with a minimum of fuss. It is also important to allow conservative and signal-flow components to be arbitrarily mixed in the same system.

The approach taken is to write component descriptions using conservative semantics, except that terminal and node declarations will only require types for those values that are actually used in the description. Thus, signal-flow terminals will only require the type of one value to be specified (typically the potential, but could alternatively be the flow), whereas conservative terminals would require types for both values (the potential and flow). For example, consider a differential voltage amplifier, a differential current amplifier, and a resistor. The amplifiers are written using signal-flow terminals and the resistor uses conservative terminals. These examples are meant to illustrate conceptual points only, and not suggest a syntax or description format.

voltage amplifier

```

    terminal input potential (V)  $i_+, i_-$ 
    terminal output potential (V)  $o_+, o_-$ 
    parameter real  $A_V$ 
{
     $V(o_+, o_-) \leftarrow A_V V(i_+, i_-)$ 
}

```

In this case, only the voltage on the terminals are declared because only voltage is used in the body of the model. Here i_+ , i_- , o_+ , and o_- are the terminals and $V(o_+, o_-)$ and $V(i_+, i_-)$ access the potential difference between the terminals.

current amplifier

```

    terminal input flow (I)  $i_+, i_-$ 
    terminal output flow (I)  $o_+, o_-$ 
    parameter real  $A_I$ 
{
     $I(o_+, o_-) \leftarrow A_I I(i_+, i_-)$ 
}

```

Here, only current is used in the body of the model, so only current need be declared at the terminals.

resistor

terminal conservative (V, I) a, b
parameter real R
{
 $V(a, b) \leftarrow RI(a, b)$
}

The description of the resistor relates both the voltage and current on the terminals, so both must be declared.

In summary, only those signals types declared on the terminals are accessible in the body of the model. Conversely, only those signals types used in the body need be declared.

This approach provides all of the power of the conservative formulation for both signal-flow and conservative terminals, without forcing types to be declared for unused signals on signal-flow nodes and terminals. In this way, the first benefit of the traditional signal-flow formulation is provided without the restrictions. The second benefit, that of a smaller, more efficient, set of equations to solve, is provided in a manner that is hidden from the user. The simulator begins by treating all terminals as being conservative, which will allow the connection of signal-flow and conservative terminals. This results in additional unnecessary equations for those nodes that only have signal-flow terminals. This situation can be recognized by the simulator and those equations eliminated. Taking this approach, also results in unnecessary equations being eliminated even when the user unnecessarily uses conservative terminals for a component that has been written using signal-flow semantics.

Thus, this approach to allowing mixed conservative/signal-flow descriptions provides the following benefits:

1. Conservative components and signal-flow components can be freely mixed. In addition, signal-flow components can be converted to conservative components, and visa versa, by modifying only the component behavioral description. (Currently VHDL-AMS requires modifying the behavioral description (the architecture), its declaration (the entity), as well as any architectures that instantiate the description, and perhaps their entities and any architectures that instantiate them, etc.)
2. Many of the capabilities of conservative terminals, such as the ability to access flow and the ability to access floating potentials, are available with signal-flow terminals.
3. Unnecessary equations will be eliminated regardless of whether terminals are denoted signal-flow or not.

4. Signal-types only have to be given for potentials and flows if they are accessed in a behavioral description.
5. If nodes and terminals are used only in a structural description (only in instance statements), then no signal-types need be specified.

One of the benefits of this approach to signal-flow descriptions is that it blends naturally with conservative descriptions. Neither the user nor the simulator really needs to distinguish between signal-flow and conservative descriptions.

2.4 Directionality of Signal-Flow Descriptions

The above descriptions used two, somewhat redundant, ways of denoting signal-flow terminals. Direction modes were specified and only one signal type was specified. While specifying the direction of signal-flow terminals is unnecessary with pure signal-flow systems, it is required when mixing signal-flow and conservative components in the same system. Otherwise, it is not always possible to distinguish the inputs and outputs of signal-flow components. It is an interesting facet of signal-flow systems that one need not distinguish the inputs and outputs of, for example, an amplifier, but that is clearly not true with conservative systems. The input of a voltage-controlled voltage-source is fundamentally different from the output.

Using direction modes on terminals to denote signal-flow semantics is not required with the probe/source approach because the inputs and outputs can be determined by the manner in which the signal is accessed. If it is accessed as the target (left-hand side) of a contribution statement, then it is an output. If it is used in an expression, it is an input. Some other formulation methods, such as the one used in VHDL-AMS, do not have this directional nature, which would make the directional modes required. However, even with the probe/source approach, the directional modes might be considered beneficial because they improve the self-documenting nature of the approach and provide a simple sanity check on the correctness of the description.

2.5 Constraints on Signal-Flow Nodes

When signal flow terminals connect to a node there are several situations that must be avoided so as to eliminate obvious conflicts or ambiguities.

1. It is an error to connect more than one signal-flow potential output terminal to the same node.
2. It is an error to connect more than one signal-flow flow input terminal to the same node.

3. It is an error to connect a signal-flow potential output terminal and a signal-flow flow input terminal the same node.
4. It is an error to connect only signal-flow flow output terminals and/or signal-flow potential input terminals to a node.

3 Tolerances

In order to be able to reliably apply iterative solvers (such as Newton's method) to the system of equations internally generated by the simulator to describe the system it is necessary to be able to directly derive tolerances for each unknown and equation. In general, users of the language (the model writers) do not want to specify tolerances in each model for several reasons. First, when writing models, the users want to concentrate on the behavior of the component, and not on the needs of the simulator. Adding tolerances to the quantities and equations used in each model is a distraction and a burden. Second, the user of the simulator usually wants global control of the tolerances. If each tolerance were explicitly specified in each model, then it would be difficult for a user of the simulator to globally loosen or tighten tolerances across the whole system. Finally, tolerances need to vary with the type of system being studied. For example, an absolute tolerance of 1 pA is acceptable when simulating low power electrical circuits, but would be much too tight when simulating utility power distribution grids. If tolerances were placed in the model, the model would not be able to be used in applications where the signal levels are much different from those anticipated by the model writer. In order to apply tolerances in a natural way, they are specified so as to be shared by signals of the same type. To this end, natures and disciplines are defined.

3.1 Natures

Natures are a collection of attributes that describe signal types. They take the form given in the following examples.

```
nature V {
    units "V";
    tolerance 1uV;
}
nature I {
    units "A";
    tolerance 1pA;
}
```

The nature must contain a real number that acts as an absolute tolerance specification. It may also take a string that gives the units for the signals. The nature may also contain additional simulator specific attributes. For example, a simulator may support signal bounds, given in terms of a bound

on the minimum and maximum value of the signal, such that when violated an error or warning message is printed. It may also be desirable to have a nature reference related natures, such as the natures for signals that are the time-integral or time-derivative of the nature being described.

Each node or terminal would reference one or two natures. Signal flow (input and output) terminals would reference one, while conservative terminals would reference two (one each for the potential and the flow).

3.2 Disciplines

The characteristics of nodes and terminals are described using disciplines. Disciplines are a set of either one or two natures as in the following examples.

```
discipline electrical {
    potential V;
    flow I;
}
discipline voltage {
    potential V;
}
discipline current {
    flow I;
}
```

The first represents a discipline for conservative signals, as it declares natures for both the potential and the flow. The subsequent two are disciplines for signal-flow signals, where the first treats the potential as the quantity of interest and the second treats the flow as the interesting quantity.

We also allow the nature attributes to be modified as they are incorporated into a discipline.

```
discipline TTL {
    potential V {
        tolerance 1mV;
    };
    flow I {
        tolerance 1uV;
    };
}
```

The discipline is used in node and terminal declarations as follows:

```
node electrical: e1, e2;
node magnetic: m1, m2;
```

A colon is used when specifying the discipline so as to syntactically distinguish it from the subsequent node or terminal names. It must be distinguished because it is optional and is not a keyword. It is useful to allow it to be optional to allow nodes to simply pass through modules without being forced to fix their discipline.

The name of the nature is used in behavioral models when accessing signals on a node or terminal to specify which signal is desired, the potential or the flow. If a discipline is not specified for a particular node or terminal, then the signals on that node cannot be accessed in a behavioral model, however the node can still be used on instance statements in a structural description.

3.3 Signal Compatibility

All terminals connected to the same node must have disciplines with compatible natures for both the potential and the flow. If terminals have a signal-flow discipline (i.e., only one nature specified) then only the nature given must be compatible with the corresponding nature of the disciplines on the other terminals.

Natures are compatible if they all derive from the same nature declaration, though the nature's attributes may have been modified in the declaration of the disciplines. However, if modified, rules must be given that specify how the differences should be resolved. This is done using the *connect* statement.

```
connect CMOS5V to analog using CMOS5V;
```

Even without inconsistent nature attributes, the connect statement can be used to specify alternate actions. For example,

```
connect CMOS3V to CMOS10V error( "Incompatible supply voltages." );  
connect TTL to CMOS5V using TTL warning( "May need pull-up." );
```

The first connect statement causes the simulator to print an error message when terminals with disciplines *CMOS3V* and *CMOS10V* connect to the same node. The second is similar except a warning message is generated.

3.4 Comparison to SpectreHDL

SpectreHDL supports optional signal type declaration as well as nature propagation and inheritance that are not found in Verilog-A. This allows the users to define generic components that are not

assigned to any particular discipline, and so can be used in any. In most situations this can be considered a luxury. However, in one situation this ability is very important: generic components that are difficult to duplicate. A component description would be difficult to duplicate if it were non-trivial and could not be copied and modified because either it is compiled into the simulator or installed in an encrypted library.

With user-defined components the discipline can be hard-coded because if it is necessary to use the same component in a different discipline, it is a simple matter to copy the description and change the disciplines on the terminal and node declarations. However, with built-in simulator components or encrypted library components this is not possible. For most of these types of components, such as semiconductor models, this is not an issue because the models are simply too specialized to be useful in other disciplines. However, the built-in sources would be useful in any discipline, and can be quite complicated and so quite difficult to replicate.

In Spectre, the discipline for a terminal is specified along with a strength. There are four strengths, *indifferent*, *suggest*, *insist*, and *override*. When a discipline is not specified for a terminal, the terminal is “indifferent” to its discipline — it is generic. When a discipline is specified for a terminal, it can be either suggested or insisted, i.e. specified with strength *suggest* or *insist*. When the terminal is connected to a node, the terminal will propose its discipline to the node. The node will take on that discipline if it is not pre-declared with a stronger discipline and no other terminal with a stronger discipline is connected to the node. In addition, it is possible for the user to manually override the discipline on a node (apply a discipline with strength *override*).

Once this initial discipline resolution has occurred on the nodes, it may happen that there are conflicting disciplines at each end of a single branch, which cannot be allowed. To resolve this issue, the branch is responsible for propagating the stronger discipline from one end of the branch to the other. It may not be possible to resolve the discipline of a particular node because there are conflicting disciplines even after the weaker disciplines are discarded, or perhaps no disciplines were provided. This is an error that must be resolved by the user by overriding the discipline at the node. Finally, once the disciplines are resolved at the nodes, they may be different than those originally specified in a component at a terminal, thus the resolved discipline must be imported, or inherited, into the components (such as onto internal nodes).

Verilog-A chose to abandon the ability to support generic components, However, Spectre continues to support quantity propagation and inheritance internally and so provides a way of overriding the native discipline of built-in components, which allows them to be used in multi-disciplinary applications.

Here is a summary of differences between SpectreHDL and Verilog-A in the area of multidisciplinary modeling.

1. Verilog-A supports the concept of an empty discipline, but only allows its use in a structural context. This section describes how SpectreHDL supports the concept of an empty discipline in a behavioral context as well.
2. Verilog-A eliminated `val()` and `flow()` as access functions (all nodes and terminals used in behavioral models must have declared natures). They are only needed for undisciplined signals, which would no longer be supported.
3. Verilog-A does not provide quantity (discipline) propagation and inheritance.
4. Verilog-A uses disciplines rather than quantities in node and terminal declarations. This allows Verilog-A to distinguish between potential- and flow-based signal-flow models and so can support both. In contrast, if only one quantity is specified on the terminal of a SpectreHDL model, it is not possible to determine if it represents the potential or the flow. Since potential-based signal-flow models are much more common, we assume the quantity represents potential and so cannot support flow-based signal-flow models.
5. Eliminate optional specification of quantities (disciplines) and node/terminal declarations.

3.5 Tolerances for Implicit Formulations

In order for simulators to accurately and reliably simulate the system, there must be a tolerance associated with each unknown and each equation present in the system of nonlinear equations that describe the system. Disciplines, which contain information about tolerances, are used when declaring nodes and terminals. Hence, tolerances are easily associated with the unknowns that are node and branch potentials and flows (tolerances on branches are taken from the nodes they connect to). Node and branch potentials and flows make up most of the unknowns. However, it still remains to be shown how tolerances are associated with equations and those unknowns that are left. How tolerances are associated with equations is covered in this section. How tolerances are associated with the remaining unknowns is covered in the example given later.

When discussing how tolerances are associated with equations, it is important to distinguish between explicit and implicit equations. Recall that explicit formulations take the form:

$$x = f(y) \tag{1}$$

were y is known and so x can be directly computed simply by evaluating f at y . Implicit formulations can take one of several forms:

$$x = f(x) \tag{2}$$

$$f(x) = 0 \tag{3}$$

$$f(x) = g(x) \tag{4}$$

In implicit formulations one must solve a possibly nonlinear equation to determine the value of x . In general, there is usually no direct way to do so, and so an iterative approach is used.

The probe/source approach supports implicit constitutive relations, however they must be formulated in fixed-point form (2). This is not a restriction, because all forms are equivalent; an equation in one form can be easily converted to any other.

Explicit equations are clearly associated with nodes or branches because they must assign a value to a signal associated with either a node or a branch. The tolerance for the equation is the same as the tolerance for the signal. Similarly, implicit equations in fixed-point form are also associated with nodes or branches because they too assign a value to either a node or a branch. For example,

$$V(x) \leftarrow f(\dots) \quad (5)$$

or

$$I(x) \leftarrow f(\dots) \quad (6)$$

In these cases, the tolerances associated with the signals can be directly associated with the equation.

Implicit equations formulated using the root (3) or balanced (4) forms should be avoided because it is not possible in general to derive a tolerance for the equation. It is not sufficient to simply associate nodes with implicit equations because scaling could be all wrong. Some suggestions have been made to avoid this problem, but the suggestions do not avoid all problems.

Consider equations of the form $x : f(x) = g(x)$ (which reads: find x such that $f(x) = g(x)$). Extending the form to

$$V(x) : f(\dots) = g(\dots) \quad (7)$$

or

$$I(x) : f(\dots) = g(\dots) \quad (8)$$

adds two benefits:

- Indicates a likely tolerance for the branch equation.
- Indicates to the topology checker how branch x should be treated.

Adding an (optional) tolerance specification on each equation is possible but not desirable because the branch equation may be arbitrarily scaled and so may not fit any existing tolerance. Since each equation may have a different scaling, each may need a different tolerance, which could lead to an explosion in the number of tolerances needed. The probe/source approach avoids this by

constraining modelers to formulate their equations so that the small number of tolerances that exist for signals can be used directly and naturally used with the equations as well.

These tolerance issues behoove use to try to minimize the need of the user to employ the root and balanced forms. In MAST, the forms were largely employed to

- Specify potentials, which originally could not be directly specified. This restriction has been eliminated in all modern languages.
- Write equations of the form

$$\dot{x} = f(x) \tag{9}$$

This could be resolved by explicitly allowing d/dt operator on left-hand side of the signal assignment statement, or by providing a time-integral operator (such as *integ()*).

Once these cases have been resolved, there are very few common cases where this general form is useful. One case is the ideal opamp, where one wants to write

$$V(out) : V(in) = 0 \tag{10}$$

This can be written using the fixed-point form as follows:

$$V(out) \leftarrow V(out) + V(in) \tag{11}$$

While this works, it is much more obscure than (10). I suggest that we allow the notation of (10) and define it to be equivalent to (11) in its behavior and treatment of tolerances.

4 Example

This example is given to illustrate how a system of equations with associated tolerances is constructed within the framework of what has been presented so far. This example will show how to use the information given in a model description and will also show what information is still needed to construct a complete description that can be reliably simulated.

The system, a simple motor servo loop shown in Figures 2 and 3, consists of both signal-flow and conservative components. The descriptions of the components are shown below. They are highly stylized and are not assumed to be written in any particular language. The *source*, *amp*, and *sensor* descriptions are written using signal-flow semantics because they are modeled as being unidirectional and only concerned with potentials. The *motor* and *wheel* descriptions are written using conservative semantics because the constitutive equations for these models relate the potential and flow at the terminals.

Figure 2 — Block diagram of system for which equations are to be formulated. The source, amplifier, and sensor descriptions are written using signal-flow semantics, whereas the motor and wheel are conservative models. The components are labeled with upper case designators, the terminals are in lower case, and nodes labels are circled.

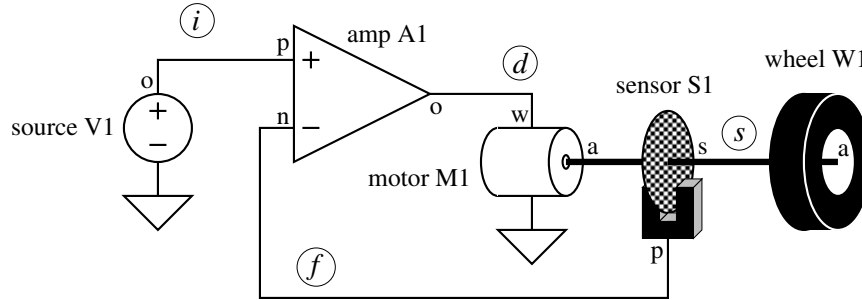


Figure 3 — Netlist for servo system of Figure 2. Parameter and terminal assignment is designated with the syntax $x \leftarrow y$, where x represents the parameter or terminal name and y represents the number or node being passed in.

| | | | |
|----|--|--------|--|
| V1 | (o \leftarrow i) | source | (A \leftarrow 1, f \leftarrow 1) |
| A1 | (p \leftarrow i, n \leftarrow f, o \leftarrow d) | amp | (A \leftarrow 100) |
| M1 | (w \leftarrow d, a \leftarrow s) | motor | (A \leftarrow 1k, B \leftarrow 0.01) |
| S1 | (s \leftarrow s, p \leftarrow f) | sensor | (A \leftarrow 1m) |
| W1 | (a \leftarrow s) | wheel | (I \leftarrow 10) |

The signal declaration are given in Figures 4 and 5. The signal-flow models are shown in Figure 6. In all of these descriptions a nature is given for the potential at the terminals. In addition, each equation is an assignment to one of the terminal potentials, and so the nature for each of these terminal potentials is also associated with the corresponding equation.

The conservative models are shown in Figure 7. In these descriptions a nature is given for both the potential and the flow at the terminals. In addition, all of the equations are assignments to either a terminal potential or a terminal flow, and so the nature for each of these terminal potentials and flows is also associated with the corresponding equation.

Figure 4 — Nature declarations for voltage, current, torque and angle.

| | | | |
|---|---|---|---|
| <pre>nature V { units "V"; tolerance 1μV; }</pre> | <pre>nature I { units "A"; tolerance 1pA; }</pre> | <pre>nature τ { units "NM"; tolerance 1μNM; }</pre> | <pre>nature φ { units "rads"; tolerance 1μRads; }</pre> |
|---|---|---|---|

Figure 5 — Disciplines for electrical (conservative), voltage (signal flow), rotational (conservative), and phase (signal flow).

| | | | |
|---|--|---|--|
| <pre>discipline electrical { potential V; flow I; }</pre> | <pre>discipline voltage { potential V; }</pre> | <pre>discipline rotational { potential φ; flow τ; }</pre> | <pre>discipline phase { potential φ; }</pre> |
|---|--|---|--|

Figure 6 — Abstract descriptions of the signal-flow components. The *source*, *amp*, and *sensor* are written using signal-flow semantics because they are modeled as being unidirectional and only concerned with potentials. t is time, a global simulator parameter.

| | | |
|--|--|---|
| <pre><i>source</i> output voltage terminal: o parameter real A parameter real f { V(o) ← A sin(2πft) }</pre> | <pre><i>amp</i> input voltage terminal: p, n output voltage terminal: o parameter real A { V(o) ← AV(p, n) }</pre> | <pre><i>sensor</i> input phase terminal: s output voltage terminal: p parameter real A { V(p) ← Aφ(s) }</pre> |
|--|--|---|

Figure 7 — Abstract descriptions of the conservative components. The *motor* and *wheel* are written using conservative semantics because the constitutive equations for these models relate the potential and flow at the terminals.

| <i>motor</i> | <i>wheel</i> |
|---|---|
| electrical terminal: w | rotational terminal: a |
| rotational terminal: a | parameter real I |
| parameter real A, B | { |
| { | $\tau(a) \leftarrow I \frac{d^2 \phi(a)}{dt^2}$ |
| $V(w) \leftarrow A \frac{d\phi(a)}{dt}$ | } |
| $\tau(a) \leftarrow BI(w)$ | |
| } | |

4.1 Equation Formulation

To make the example more manageable, a reduced form of the sparse tableau equation formulation method is used.¹ The method encodes KPL (Kirchhoff's Potential Law) into the branch equations by replacing terminal potentials with node potentials. Compared to Sparse Tableau, this formulation eliminates the B KPL equations as well as the B branch potential unknowns. However, it leaves the N KFL equations, the N node potential unknowns, the B branch equations, and the B branch flow unknowns. This equation formulation method can also be interpreted as a form of modified-nodal analysis that has been extended by explicitly giving all branch equations, not just those that cannot be formulated in nodal analysis form (i.e., even branch equations that specify flows as a function of potentials are included).

In the following equations, P is used to denote potentials of undefined nature and F is used to denote flows of undefined nature. The first four equations represent KFL applied to each of the four nodes. The remaining equations are the branch equations for the components. The first four unknowns are the node potentials, and the remaining unknowns are the branch flows. Initially, natures are only associated with the branch equations because they are taken from the descriptions. The natures will be propagated to the KFL (Kirchhoff's Flow Law) equations in a later step.

¹There is nothing special about this particular formulation. It was chosen because it struck a balance between the verbosity of the Sparse Tableau formulation, and the terseness of the Modified-Nodal formulation. Other formulations, such as Sparse Tableau or Modified-Nodal, could have been used without affecting the ideas and conclusions presented in this paper.

| Eqn ID: | Equation | Nature of Equation | Unknown | Nature of Unknown |
|-----------|--|--------------------|------------------------|-------------------|
| KFL i : | $F(\text{source}_o) + F(\text{amp}_p) = 0$ | – | $P(i)$ | – |
| KFL d : | $F(\text{amp}_o) + I(\text{motor}_w) = 0$ | – | $P(d)$ | – |
| KFL s : | $\tau(\text{motor}_a) + F(\text{sensor}_s) + \tau(\text{wheel}_a) = 0$ | – | $P(s)$ | – |
| KFL f : | $F(\text{sensor}_p) + F(\text{amp}_n) = 0$ | – | $P(f)$ | – |
| source: | $V(i) = A \sin(2\pi ft)$ | V | $F(\text{source}_o)$ | – |
| amp 1: | $F(\text{amp}_p) = 0$ | – | $F(\text{amp}_p)$ | – |
| amp 2: | $F(\text{amp}_n) = 0$ | – | $F(\text{amp}_n)$ | – |
| amp 3: | $V(d) = A[V(i) - V(f)]$ | V | $F(\text{amp}_o)$ | – |
| motor 1: | $V(d) = A \frac{d}{dt} \phi(s)$ | V | $I(\text{motor}_w)$ | I |
| motor 2: | $\tau(\text{motor}_a) = BI(d)$ | τ | $\tau(\text{motor}_a)$ | τ |
| sensor 1: | $F(\text{sensor}_s) = 0$ | – | $F(\text{sensor}_s)$ | – |
| sensor 2: | $V(f) - A\phi(s) = 0$ | V | $F(\text{sensor}_p)$ | – |
| wheel: | $\tau(\text{wheel}_a) = I \frac{d^2}{dt^2} \phi(s)$ | τ | $\tau(\text{wheel}_a)$ | τ |

There are a few natures missing from a few branch equations and unknowns. These equations and unknowns were automatically generated by the probe/source formulation in order to map the signal-flow semantics into conservative semantics. Generating these additional equations and unknowns is beneficial because it allows signal-flow terminals to be directly connected to conservative terminals. However, for those nodes with no conservative terminals, these extra equations are unnecessary and will eventually be eliminated.

The following set of equations is just the KFL equations after the natures have been propagated from the branches to the nodes. The natures for the unknowns can be taken directly from the declarations of the terminals connected to the node. The nature of the equations is derived from the nature associated with the various terms in the KFL equation.

| Eqn ID: | Equation | Nature of Equation | Unknown | Nature of Unknown | From |
|-----------|--|--------------------|-----------|-------------------|---|
| KFL i : | $F(\text{source}_o) + F(\text{amp}_p) = 0$ | – | $V(i)$ | V | source _o , amp _p |
| KFL d : | $F(\text{amp}_o) + I(\text{motor}_w) = 0$ | I | $V(d)$ | V | motor _w |
| KFL s : | $\tau(\text{motor}_a) + F(\text{sensor}_s) + \tau(\text{wheel}_a) = 0$ | τ | $\phi(s)$ | ϕ | motor _a , wheel _a |
| KFL f : | $F(\text{sensor}_p) + F(\text{amp}_n) = 0$ | – | $V(f)$ | V | sensor _p , amp _n |

Nodes i and f are signal-flow only nodes, which implies corresponding KFL equations can be eliminated. As a result, the variables $F(\text{source}_o)$, $F(\text{amp}_p)$, $F(\text{amp}_n)$, and $F(\text{sensor}_p)$ are not used and can be eliminated. Once the variables are gone, equations ‘amp 1’ and ‘amp 2’ are no longer

needed. Finally, equation ‘sensor 1’ indicates that zero can be substituted for $F(\text{sensor}_s)$, eliminating further need for this equation and unknown. The resulting set of equations and tolerances becomes:

| Eqn ID: | Equation | Nature of Equation | Unknown | Nature of Unknown |
|-----------|---|--------------------|------------------------|-------------------|
| KFL d : | $F(\text{amp}_o) + I(\text{motor}_w) = 0$ | I | $V(i)$ | V |
| KFL s : | $\tau(\text{motor}_a) + 0 + \tau(\text{wheel}_a) = 0$ | τ | $V(d)$ | V |
| source: | $V(i) = A \sin(2\pi ft)$ | V | $\phi(s)$ | ϕ |
| amp 3: | $V(d) = A[V(i) - V(f)]$ | V | $V(f)$ | V |
| motor 1: | $V(d) = A \frac{d}{dt} \phi(s)$ | V | $F(\text{amp}_o)$ | – |
| motor 2: | $\tau(\text{motor}_a) = BI(d)$ | τ | $I(\text{motor}_w)$ | I |
| sensor 2: | $V(f) = A\phi(s)$ | V | $\tau(\text{motor}_a)$ | τ |
| wheel: | $\tau(\text{wheel}_a) = I \frac{d^2}{dt^2} \phi(s)$ | τ | $\tau(\text{wheel}_a)$ | τ |

The branch equation given for the wheel contains a second-order derivative. Most circuit simulators are only able to handle first-order differential equations. It is necessary to reduce the second-order differential equation given for the wheel to two first-order differential equations as follows:

$$\omega = \frac{d}{dt} \phi(s) \quad (12)$$

$$\tau(\text{wheel}_a) = I \frac{d}{dt} \omega \quad (13)$$

The difficulty comes in determining the nature for the new equation (12) and the new unknown (ω). If second derivatives are restricted such that they can only be applied to objects with natures, then the nature for the new equation and unknown can be derived from the nature of the argument. In this case, the nature of the argument is ϕ , so the nature for both (12) and ω is denoted $d\phi/dt$. How this new nature is derived is discussed in Section 4.3.

| Eqn ID: | Equation | Nature of Equation | Unknown | Nature of Unknown |
|----------|--|---------------------|------------------------|---------------------|
| wheel 1: | $\omega = \frac{d}{dt} \phi(s)$ | $\frac{d}{dt} \phi$ | ω | $\frac{d}{dt} \phi$ |
| wheel 2: | $\tau(\text{wheel}_a) = I \frac{d}{dt} \omega$ | τ | $\tau(\text{wheel}_a)$ | τ |

The system of equations can be further reduced by substituting for explicitly given flow variables ($\tau(\text{motor}_a)$, $\tau(\text{wheel}_a)$) and eliminating associated branch equations (‘motor 2’, ‘wheel 2’). This is the step that converts the equations into modified nodal analysis form. Finally, propagate natures onto signal-flow output flows from the nodes they are connected to. In this case, the nature for KFL d is propagated to $F(\text{amp}_o)$ because terminal o of ‘amp’ is connected to node d .

| Eqn ID: | Equation | Nature of Equation | Unknown | Nature of Unknown |
|-----------|---|--------------------|---------------------|--------------------|
| KFL d : | $I(\text{amp}_o) + I(\text{motor}_w) = 0$ | I | $V(i)$ | V |
| KFL s : | $BI(d) + I\frac{d}{dt}\omega = 0$ | τ | $V(d)$ | V |
| source: | $V(i) = A \sin(2\pi ft)$ | V | $\phi(s)$ | ϕ |
| amp 3: | $V(d) = A[V(i) - V(f)]$ | V | $V(f)$ | V |
| motor 1: | $V(d) = A\frac{d}{dt}\phi(s)$ | V | $I(\text{amp}_o)$ | I |
| sensor 2: | $V(f) = A\phi(s)$ | V | $I(\text{motor}_w)$ | I |
| wheel 1: | $\omega = \frac{d}{dt}\phi(s)$ | $\frac{d}{dt}\phi$ | ω | $\frac{d}{dt}\phi$ |

Another optimization would be to use equation ‘source’ to substitute for $V(i)$, eliminating both the equation and the unknown. In general, it is possible to eliminate a branch equation in this manner if it corresponds to a grounded independent potential source. One can find such branch equations by looking for singleton rows in the Jacobian (rows with only one entry). This test could also be used as an alternate test for determining that equations ‘amp 1’, ‘amp 2’, and ‘sensor 1’ should be eliminated.

This is the final set of equations that will be solved by the simulator. It is presumed to be solvable by an iterative solver because there are an equal number of equations and unknowns, and because there are natures, and their corresponding tolerances, associated with every equation and node.

A final optimization would be to eliminate nodes that were not being output from the list of unknowns.

4.2 Summary of the Rules of Elimination

The rules used to eliminate equations and unknowns are:

1. Can eliminate all branch equations that explicitly specify a flow and the corresponding branch flow unknowns. This converts the Reduced Sparse Tableau to Modified-Nodal Analysis form.
2. Can eliminate a branch equation and a node potential unknown if the branch is associated with a grounded independent potential source. One can detect a grounded independent potential source by inspecting the Jacobian and looking for a *row* that has only one entry. The row of the entry corresponds to the branch equation, and the column corresponds to the node potential unknown.
3. If a node is driven by a grounded potential source (either independent or dependent), and if the branch flow for the source is not used as a controlling quantity for another component,

then the KFL equation and the branch flow unknown associated with the source can be eliminated. With a little care, the branch flow can still be computed at very low cost for use as an output (it should be naturally computed as the residual for the eliminated KFL equation). One can detect a grounded potential source (independent or dependent) that is not used as a controlling quantity for another component by inspecting the Jacobian and looking for a *column* that has only one entry. The row of the entry corresponds to the KFL equation, and the column corresponds to the branch flow unknown.

4. The final optimization, that of discarding the uninteresting potentials is difficult to implement because it requires that evaluations be scheduled so that they occur in the proper order and any feedback loops be broken by retaining one potential along the loop in the list of unknowns. This is also troublesome in an interactive environment because the list of unknowns to save can change. This would require either the structure of the equations to change to add the eliminated unknown and equation back in, or a special mechanism to be added that would grab and save the result of an “intermediate” step in the model evaluation computations. In other words, to retain the independence between model equations, the potential is not really eliminated, instead it is hidden from the system of simultaneous equations and retained as an intermediate variable. In addition, the equations are not eliminated, but combined in a serial manner. This requires considerable additional machinery be added to the simulator without actually reducing the number of arithmetic operations — rather the computations are just localized and so perhaps a bit faster.

The eliminations described in Item 1 are already being performed in Spectre.

It seems that the eliminations described in Item 2 are probably best avoided. They act to eliminate a row and column in the Jacobian associated with a singleton (there is only one nonzero in both the row and column), and so little efficiency is gained by performing the elimination. Second, it eliminates a node potential, which is often needed for output. Thus, if the elimination is performed, then some a posteriori action will be needed to retrieve the node potential. This action would probably be expensive enough to offset the benefit of performing the elimination. However, this optimization may be needed with signal-flow current because there will be no tolerances available for the potentials.

The eliminations described in Item 3 are the ones that could really help Spectre in two situations. First, grounded sources are usually used to implement power supplies, clocks, and sometimes even grounds. In these cases the KFL equations associated with nodes driven by these sources can have hundreds or thousands of terms. These terms do not cause fill-ins but still have to be factored and so represent some burden. Second, large signal-flow systems will generate a large number of KFL equations that are useless and that can be eliminated, which speeds the simulator somewhat and avoids the need to find tolerances for the equations, which eliminates the need for the user to assign

an nature to the equation. This is important because the user would not be aware of this equation and so would not know how to assign a nature.

The reductions proposed in Item 4 are not expected to be beneficial enough (slight performance increase) to overcome their cost (increased complexity of the code).

Reminder to Author # 1

Should we also consider eliminating KFL equations is there is only one conservative terminal connected to the node?

4.3 Dynamics

Reminder to Author # 2

What about derivatives that are not linearly accessible. Example, $\frac{dB}{dH} = \frac{dB}{dt} \left(\frac{dH}{dt} \right)^{-1}$. Or there is the VCO example, $V(out) \leftarrow \cos(\text{idtmod}(f(V(in))))$.

The nature used in equation ‘wheel 1’ and the unknown ω refer to the time derivative of ϕ . This type of equation and unknown are generated by the simulator whenever higher order derivatives and integrals are present. There are two ways to derive tolerances for these equations and unknowns, compute them, or have the users specify them. I will present both approaches, and suggest that both are needed.

A tolerance for the time derivative of ϕ can be computed by dividing the tolerance for ϕ by the timestep h . To see this, assume the time-derivative is approximated using backward Euler.

$$\omega = \frac{d}{dt}\phi(s) \approx \frac{1}{h}(\phi(s(t_i)) - \phi(s(t_{i-1}))) \quad (14)$$

Multiply through by h to have the equation formulated in units of phase.

$$h\omega \approx \phi(s(t_i)) - \phi(s(t_{i-1})) \quad (15)$$

The iterative solver would have converged on this equation if

$$h\omega - \phi(s(t_i)) + \phi(s(t_{i-1})) < \epsilon_\phi \quad (16)$$

where ϵ_ϕ is taken to be the absolute tolerance for ϕ . Hence, the iterative solver would have converged on the original equation if

$$\omega - \frac{1}{h}(\phi(s(t_i)) - \phi(s(t_{i-1}))) < \frac{\epsilon_\phi}{h} \quad (17)$$

This is equivalent to using $\epsilon_\omega = \epsilon_\phi/h$ as the absolute tolerance for ω .

The second approach would be to simply have the user give a nature, and hence an absolute tolerance, for ω . One would need a mechanism for automatically finding this tolerance from ϕ because this equation and unknown are automatically generated by the simulator. If the nature for ϕ points to natures for its time-derivative and time-integral, then given ϕ it would be possible to find tolerances for ω , as well as higher-order derivatives and integrals.

It would probably be desirable to have access to both mechanisms because neither one is completely satisfying. In the first approach, that of computing the tolerance by dividing through by h , the absolute tolerance tends to go to zero when h is large, which might result in an artificially tight tolerance that precludes convergence. Similarly, for small h , the second approach, where the user gives a constant absolute tolerance, will likely be so tight as to preclude convergence. It seems that a combination of the two will result in a reasonable absolute tolerance for both large and small time steps.

4.4 Observations

- This natural blending of signal-flow and conservative semantics brings several benefits. It allows signal-flow terminals to be directly connected to conservative terminals. It also allows components to be converted from signal-flow to conservative in the course of a typical top-down design or bottom-up verification flow without difficulty.
- With the capability of signal-flow models being extended to the point where they are comparable to conservative components, and with the signal-flow and conservative signals being accessed in the same manner, the user need not distinguish between signal-flow and conservative semantics at all. Rather, they only need understand that signals must be declared if used, and need not be declared if not used. Thus, if only the potential of a terminal is used (it is probed or sourced), then only the potential needs to be declared for the terminal (the discipline associated with the terminal need only specify a nature for the potential).

Blending signal-flow and conservative semantics in this manner provides a capability that is considerably more powerful than any approach that forces the semantics to be kept separate, and is also very easy to learn and use.

- The direction modes ('input' and 'output') are not needed for the probe/source approach. Determining whether a terminal is used for input or output can easily be done by noticing whether the potential associated with the terminal is probed or sourced in the behavioral description. Specifying the modes on the terminals does provide a nice sanity check.
- The concept of branches is foreign in a signal-flow context. Instead, it is desirable to directly

access the signals associated with the terminals. In a description that mixes both signal-flow and conservative terminals, it might be useful to assume that signal-flow terminals have implicitly declared branches from the terminal to ground whose name is the same as the name of the terminal.

5 Conclusion

This document shows by way of a fairly comprehensive example that natures on branch equations and unknowns can easily be propagated to all equations and unknowns in the final system of equations. It also shows how signal-flow semantics and conservative semantics can be naturally combined. It shows how tolerances can be associated with each equation and each unknown without putting an unnecessary burden on the user. Finally, it discusses the issue of determining the tolerances for the additional unknowns and equations that result when converting higher-order integro-differential equations into first-order differential equations.