1. Issues list
This is a list of issues or questions that I have with regard to the current state of dot-1 (tony).

1. See attached info on variables and expressions

2. (page 17) Request that WFCMap be moved from Signals/SignalGroups to Timing for the following reasons:
   - if WFCMap is associated with Signals it can never be adjusted according to domain
   - the Timing block is where the wfc’s get defined on for each signal for a given pat-exec and hence is the appropriate place to define mapping
   - the Signal/SignalGroups block has no way of knowing if wfc chars are being used that don’t actually exist. The Timing block would have this knowledge.

Timing name {
    SignalGroups name;
    WCFMap {
        sigref_expr {
            { z->x; 01->x; }
        }
    }
    WaveformTable W { }
}
2. New Variable and Expression Definitions

Table 1: Variable and Expression Usage

<table>
<thead>
<tr>
<th>expr-type</th>
<th>variable types</th>
<th>where defined</th>
<th>where used</th>
<th>syntax examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>time_expr</td>
<td>time integer</td>
<td>Spec</td>
<td>Timing</td>
<td>’23.0ns/2+16.5e-9-t2’</td>
</tr>
<tr>
<td></td>
<td>real @label</td>
<td>Spec Timing</td>
<td></td>
<td>’t1/i + r - t2’</td>
</tr>
<tr>
<td>real_expr</td>
<td>time integer</td>
<td>Spec</td>
<td>Pattern</td>
<td>same as a timing expressions</td>
</tr>
<tr>
<td></td>
<td>real @label</td>
<td>Spec Timing</td>
<td></td>
<td>except that it may be used in a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>non-timing context</td>
</tr>
<tr>
<td>integer_expr</td>
<td>integer other?</td>
<td>SignalGroups</td>
<td>Pattern</td>
<td>PatternCharacteristics -&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NumberVectors ‘max * mode’;</td>
</tr>
<tr>
<td>sigvar_expr</td>
<td>SignalVariable</td>
<td>PatternGroups</td>
<td>Pattern</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V { grp = ‘sv1’; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V { grp = ‘sv2[1..5]’; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>V { grp = ‘sv3[5 4 2 3 1]’; }</td>
</tr>
<tr>
<td>logical_expr</td>
<td>integer time</td>
<td>SignalGroups</td>
<td>Pattern</td>
<td>If ‘i == 0’ {}</td>
</tr>
<tr>
<td></td>
<td>real SignalVariable</td>
<td>Spec</td>
<td></td>
<td>If ‘period &gt; 23.0ns’ {}</td>
</tr>
<tr>
<td></td>
<td>SignalGroup</td>
<td>Spec SignalGroups</td>
<td></td>
<td>If ‘value &lt; 3e-6’ {}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Spec Timing</td>
<td></td>
<td>If ‘sv2[1..5] == 11000’ {}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SignalGroups</td>
<td></td>
<td>If ‘(s1==H)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SignalGroups</td>
<td></td>
<td>If ’grp != HHHH’ {}</td>
</tr>
</tbody>
</table>

The following variable types are to be defined in sec 10 - SignalGroups

1. **Integer** - According to dot-0 an integer can be used in a time_expr, however there is no way to define a spec variable of type integer. If dot-1 defines an integer variable, can it then be used in a time_expr:
   ```
   SignalGroups {
     k Integer;
   }
   Timing -> WaveformTable -> Waveforms -> sigref_expr ->
   01 { ‘2ns+k*(0.5ns)’ U/D; }
   ```

2. **SignalVariable** - New type in dot-1. Was previously name WFC type. Name is changed to reflect that it follows similar rules to Signals and SignalGroups.

3. **Enumeration** - New type in dot-1

4. **Real** - This type is already allowed in the Spec block. Why do we want it to be in a SignalGroup block as well? I suggest that it be removed from dot-1.

The following expression types will be defined in sec 5.n

1. **logical_expr** - new type in dot-1

2. **sigvar_expr** - new type in dot-1
A sigvar_expr is an ordered list of elements that operates like a sigref_expr but is not associated with any signal names. Its application is to hold signal data and to pass signal data between Patterns and Procedures/Macros. The output function of a sigvar_expr is an ordered string of wfc’s. SignalVariable expressions are enclosed in single quotes. A SignalVariable expression may be assigned list of wfc values to a signal-variable. The signal-variable may be used to transfer the wfc values to either another signal-variable or to signal or signal-group.

3. **cellref_expr** - new type in dot-1. As defined in dot-1.

4. **integer_expr** - new type in dot-1
   
   Is an “integer_expr” the same thing as “logical_expr”? Should we allow reference to “integer_expr” to indicate that the only allowed result is an integer?
   
   Also, should we have a reference type of “boolean_expr” which is a logical expression that is being used as a boolean?

5. **integer_list** - An integer_list is NOT a referenced type in the language. Whenever it is to be used it should be referenced as in the syntax definition as “ (INTEGER)+ “. This signifies a set of space separated integer numbers. There should not be any usage of comma separated integers. Do we put this non-definition into the language? I think not - tony. Example usage is:

   ```
   V { signname[5 4 3 2 1] = 11001; }
   
   BistRegister reg {
      TapPositions 0 2 4 6 8 10 12;
   }
   ```

6. **real_expr** - New type in dot-1. “real” variables are already allowed in a spec block. What is not defined is “real_expr”. If we want to reference such a term in dot-1 then we need to add a definition of it.
   
   A real_expr has the exact syntax as a timing expression. The difference is that this form of reference is used when the real number is not being used to represent a timing value (i.e., in a pattern).

7. **sigref_expr, time_expr** - These expression types are already defined in 1450-1999 and need not be elaborated.
3. Operators and Functions

Two changes have been made to the table currently in dot-1. The unary operators are deleted, and the extra columns showing what operators are used by expression type have been added.

<table>
<thead>
<tr>
<th>Op</th>
<th>Definition</th>
<th>time</th>
<th>real</th>
<th>int</th>
<th>logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>min()</td>
<td>minimum value</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>max()</td>
<td>maximum value</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>()</td>
<td>parenthesis</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>table 3, table 4 of IEEE Std. 1450-1999</td>
<td>SI units &amp; prefixes</td>
<td>YES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>+</td>
<td>add</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>less than (boolean value)</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than (boolean value)</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less or equal (boolean value)</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater or equal (boolean value)</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>!</td>
<td>negation (boolean value)</td>
<td></td>
<td>YES</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>and (boolean value)</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or (boolean value)</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>==</td>
<td>equal (boolean value)</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!=</td>
<td>not equal (boolean value)</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>bit-wise negation</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>bit-wise and</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bit-wise inclusive or</td>
<td>YES</td>
<td>YES</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bit-wise exclusive or</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>^~</td>
<td>bit-wise equivalence</td>
<td>YES</td>
<td>YES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>^~~, ^~^</td>
<td>reduction and</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Op</td>
<td>Definition</td>
<td>time</td>
<td>real</td>
<td>int</td>
<td>logical</td>
</tr>
<tr>
<td>----</td>
<td>------------</td>
<td>------</td>
<td>------</td>
<td>-----</td>
<td>---------</td>
</tr>
<tr>
<td>¬&amp;</td>
<td>reduction-nand (not-and)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>reduction-or</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>¬</td>
<td></td>
<td>reduction-nor (not-or)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>reduction-xor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^=^,^_=</td>
<td>reduction-xnor</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left-shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>right-shift</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:=</td>
<td>conditional expression</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>assignment</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td></td>
</tr>
</tbody>
</table>
4. Signal Mapping Using SignalVariable and Expressions (sigvar_expr)

This example illustrates the following capabilities:

1. definition of SignalVariables in a SignalGroups block
2. use of parameters in a Macro (or Procedure call)
3. use of # to update the parameter values
4. application of parameters in a Macro (or Procedure)
5. use of logic expressions (logic_expr) with signal variables in a Macro (or Procedure)
6. process of re-using signal groups (un-mapped) as signal variables (mapped)

```
Pattern pat_a {
  Macro mac_a {
    sig_a[1..5] = 10101;
    sig_b[1..5] = 11011;
    grp_c = 01010;
    sig_d = 00100;
  }
}

Signals {
  sig_a[1..5] In;
  sig_b[1..5] In;
  sig_c[1..5] In;
  sig_d In { Length 5; ScanIn; }
}

SignalGroups {
  grp_c = 'sig_c[1..5]';
}

MacroDefs {
  mac_a {
    V { sig_a[1..5] = #; }
  }
```

V { sig_b[1..5] = #; }
V { grp_c = #; }
Shift {
  V { sig_d = #; }
}

===================== mapped =====================
Signals {
  sig_aa[1..5] In;
  sig_bb[1..5] In;
  sig_cd In { Length 10; ScanIn; }
}

SignalGroups {
  sig_a[1..5] SignalVariable;
  sig_b[1..5] SignalVariable;
  grp_c SignalVariable { Length 5; }
  grp_c[1..5] SignalVariable;
  sig_d SignalVariable { Length 5; }
}

MacroDefs {
  mac_a {
    C { sig_a[1..5] = #; sig_b[1..5] = #; grp_c = #; sig_d = #; }
    C { grp_c[5 4 3 2 1] = 'grp_c'; }
    V { sig_aa[1..5] = 'sig_a[1..5]'; }
    V { sig_bb[1..5] = 'sig_b[5 3 1 2 4]'; }
    // V { sig_bb[3 4 2 5 1] = 'sig_a[]'; } // alternate stmt
    Shift {
      V { sig_cd = 'grp_d[1..5]' 'grp_c[1..5]' ; }
      // V { sig_cd = 'grp_d[1..5] + grp_c[1..5]' ; } // alternate?
      // V { sig_cd = 'grp_d[1..5]' 00000 ; } // alternate
      // V { sig_cd = 00000 'grp_c[1..5]' ; } // alternate
    }
  }
}


5. Using Logic Expressions (logic_expr) in CTL

This example illustrates the following capabilities:

1. logic expression using & (and) operation
2. logic expression made up of signals, scan signals, and core signals
3. application in CTL of logic expression to define enable logic

```plaintext
Signals {
    sig_1 In;
    sig_2 In;
}

Environment {
    CTL {
        Internal {
            sig_1 {
                IsConnected In {
                    StateElement Scan cell_a1;
                    IsEnabledBy Logic 'A & B & C' {
                        Signal A sig_2;
                        ScanSignal B cell_b3;
                        CoreSignal C core_x.sig_y;
                    }
                }
            }
        }
    }
}
```
6. Using Boolean Expressions (boolean_expr) in Patterns, Macros, and Procedures

This example illustrates the following capabilities:

1. use of parameters on a Macro (or Procedure)
2. use of logic expression in conditional-If/Else statements in a pattern
3. use of parameters in logic expression to create wfc data in a vector

Signals {
    sig_1 In; sig_2 In; sig_3 In; sig_4 In;
    sig[5..10] In;
}
SignalGroups {
    grp_1 = 'sig_1 + sig_2 + sig_3 + sig_4';
    grp_2 = 'sig[5..10]';
    grp_3 = 'grp_1 + grp_2';
    sv_1 = SignalVariable { Length 4; }
}

Pattern pat_1 {
    Macro mac_1 { sig_1 = 1; sig_2 = 0; }
    Macro mac_2 { grp_1 = 1100; sv_1 = 0011; }
}

MacroDefs {
    mac_1 {
        C { sig_1 = #; sig_2 = #; }
        If ‘sig_1 == 1’ { V { sig_1 = A; } }
        If ‘(sig_1 == 1) & (sig_2 == 0)’ { V { sig_1 = B; } }
    }
    mac_2 {
        C ( grp_1 = #; sv_1 = #; )
        If ‘grp_1 == 1100’ {
            V { grp_1 = ‘sv_1’; grp_2 = 111111; }
        } Else {
            V { grp_3 = ‘sv_1’ 000000; }
        }
    }
}