Changes and Additions to 1450.1 document (6/21/01)

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. Tony: sig_var, integer_list, boolean_expr, and all the examples included in this doc.

2. Tony: (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.

```plaintext
Timing name {
    SignalGroups name;
    WCFMap {
        sigref_expr {
            { z->x; 01->x; }
        }
    }
    WaveformTable W { }
}
```

3. Peter: requested to add issue #3 to this list: the Waveform event CompareSubstitute, defined to support enhanced diagnostic return and "known-good die" learning capability. Greg attached the issue to the bottom of these minutes for subsequent review by the group.

4. Doug: identified that he owes the group a write-up on the "merge()" function, currently still open from last discussions on the BreakPoint designation issue.

5. Doug: identified that the Else construct needs to be added to the current spec.
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>engr_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</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 except that it may be used in a non-engr unit context. e.g. volts/ns.</td>
</tr>
<tr>
<td></td>
<td>real @label</td>
<td>Spec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer_expr</td>
<td>integer</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>integer_list</td>
<td>integer</td>
<td>BistStructures</td>
<td>1,2,3,4,5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SignalVariables</td>
<td>3, 6, 4, 5, 1, 2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>CTL</td>
<td>1..5</td>
<td></td>
</tr>
<tr>
<td>sigvar_expr</td>
<td>SignalVariable</td>
<td>PatternGroups</td>
<td>Pattern</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</td>
<td>SignalGroups</td>
<td>Pattern</td>
<td>If ‘i == 0’ {}</td>
</tr>
<tr>
<td></td>
<td>time real</td>
<td>Spec</td>
<td></td>
<td>If ‘period &gt; 23.0ns’ {}</td>
</tr>
<tr>
<td></td>
<td>SignalVariable</td>
<td>Spec</td>
<td></td>
<td>If ‘value &lt; 3e-6’ {}</td>
</tr>
<tr>
<td></td>
<td>SignalGroup</td>
<td>SignalGroups</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>
<tr>
<td>bool_expr</td>
<td>SignalGroup</td>
<td>Spec</td>
<td>Pattern</td>
<td>If ‘grp != HHHH’ {}</td>
</tr>
</tbody>
</table>

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

1. **Integer**
   A variable of type “Integer” may be defined in the Signals block. It may be used in a logic_expr or bool_expr within a Pattern, Procedure or Macro block. An Integer variable may also be used in an engr_expr within a Timing block.
   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. In dot-1 an integer variable can be defined and then be used in a time_expr:

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? To 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. **boolean_expr**
   A boolean expression (bool_expr) is a logical expression that evaluates to a true or false. 
   *Change references in document from logical_expr to bool_expr i.e., in pattern->If conditions.*

3. **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.

   ```
   sig_var[1..5] = 11100;
   sig_var[5, 4, 2, 3, 1] = 00011;
   ```

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

5. **integer_expr**
   An integer_expr is ...

   ```
   SignalGroups {
     k Integer;
   } // end SignalGroups
   
   Timing { WaveformTable { Waveforms { sigref {
     XY { '2ns+k*(0.5ns)' U/D; }
   }}}} // end Timing
   
   Pattern P {
     C { 'k = k + 1'; }
     If 'k >= 99' {} 
     Loop k {} 
   } // end Pattern
   ```

6. **integer_list**
   An integer_list is a reference type used to specify an ordered list of integer values. Allowed operators in an integer_list are the comma “,” which is the required integer separation character and the ellipe “..” which specifies a range of integers. Example usage is:

   ```
   V { signame[5, 4, 3, 2, 1] = 11001; }
   
   BistRegister reg {
     TapPositions 0, 2, 4, 6, 8, 10, 12;
     Connection prpg_sigs 0..25;
   }
   ```

7. **real_expr**
An expression of type real is defined in a Spec table as defined in 1450-1999 document. The format of a real number is of the form: <number>e<+\-><number>. Real number can be used to represent things that are not standard SI units. For example a slew rate in volts/ns.

8. **engr_expr**
An engineering expression (engr_expr) is the generic term for any expression of the form <number><prefix><SI unit>. For example, ‘23ns’ is a time expression, ‘10uf’ is a capacitance expression.
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>integer</th>
<th>logical</th>
<th>bool</th>
<th>sigvar</th>
</tr>
</thead>
<tbody>
<tr>
<td>min()</td>
<td>minimum value</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>max()</td>
<td>maximum value</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>()</td>
<td>parenthesis</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>+</td>
<td>add</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>%</td>
<td>modulus</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than (boolean value)</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than (boolean value)</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</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>
<td>YES</td>
<td>NO</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>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>!</td>
<td>negation (boolean value)</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>and (boolean value)</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or (boolean value)</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>==</td>
<td>equal (boolean value)</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>!=</td>
<td>not equal (boolean value)</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>~</td>
<td>bit-wise negation</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>&amp;</td>
<td>bit-wise and</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bit-wise inclusive or</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>^</td>
<td>bit-wise exclusive or</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>^</td>
<td>bit-wise equivalence</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

Table 2—Operators and functions allowed in expressions
Table 2—Operators and functions allowed in expressions

<table>
<thead>
<tr>
<th>Op</th>
<th>Definition</th>
<th>time</th>
<th>real</th>
<th>integer</th>
<th>logical</th>
<th>bool</th>
<th>sigvar</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>reduction and (not-and)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>reduction or</td>
<td></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>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>reduction xor</td>
<td></td>
<td></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>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left shift</td>
<td></td>
<td></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>
<td></td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>conditional expression</td>
<td></td>
<td></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>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>..</td>
<td>range operator (elipsis)</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>YES</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 = #; }
        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_b[1..5]'; } // alternate stmt
        Shift {
            C { grp_c[5 4 3 2 1] = 'grp_c'; } \texttt{ need to be outside shift?}
            V { sig_cd = 'sig_d[1..5] 'grp_c[1..5]'; } 
            // V { sig_cd = 'sig_d[1..5]' 00000 ; } // alternate stmt
            // V { sig_cd = 00000 'grp_c[1..5]' ; } // alternate stmt
        }
    }
}
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’ {
                        A { Type Signal; Name sig_2; }
                        B { Type ScanSignal; Name cell_b3; }
                        C { Type CoreSignal; Name 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_3 = A; }}
        If '(sig_1 == 1) & (sig_2 == 0)' { V { sig_3 = B; }}
    }
    mac_2 {
        C { grp_1 = #; sv_1 = #; }
        If 'grp_1 == 1100' {
            V { grp_3 = 'sv_1' 111111; }
        }
        Else {
            V { grp_3 = 'sv_1' 000000; }
        }
    }
}