Binning

1 Conceptual Model

The proposed binning conceptual model is comprised of pass and fail soft bin groups, counters, device specs[1], zero or more soft-to-hardware bin maps, bin setting mechanisms, bin based events, and notions of a null bin and bin arbitration.

The motivation behind these elements and the structure imposed on them, besides the obvious classification of devices based on test results and collection of granular test data, is to enable writing easy to maintain code. Specifically, we exploit the relationship between device specifications and software bins so that editing (adding, deleting, enabling, disabling) device specifications and bins can alter the execution of a properly designed flow without having to edit the code describing that flow.

1.1 Top Level Container

This container has an id and contains general information and two groups of bins: pass bins and fail bins. The conceptual model for each group is the same hence, one syntax should suffice for both.

The general information consists of a user-settable/readable property *ContinueOnFail* which is used as the default on fail bins for which this property is not explicitly set.

1.2 Bin Groups

Each group contains one or more axes which in turn contains one or more bins. If the group contains only one axis, the axis may be anonymous. If the group contains two or more axes, each axis must have an identifier unique to the group. Each axis contains one or more bin definitions. Each bin definition identifier must be unique to the axis.

ContactOpens	ContactShorts	Functional	Timing
5	5	6	7

Table 1: Fail bin group with anonymous axis

	3.00GHz	2.93GHz	2.66GHz	ClockSpeed
8Mb	1	3	3	
4Mb	2	4	4	
CacheSize				

Table 2: Pass bin group with two labeled axes

[1] Device specs to be supported by permitting binning axes and spec arrays to be passed to a test

Table 1 and Table 2 are representations of typical soft bin groups. Axis labels are shown in bold white on black background and soft bin names are shown in bold black on white. Table 1 is in effect a 1 x 4 array, i.e., 1 anonymous axis with 4 soft bin definitions. If there is a soft-to-hard bin map, each of the array elements shown in yellow must be mapped to a hardware bin. Hardware bin numbers imposed by an associated bin map are shown in green.

Table 2 is a 3 x 2 array, axes labeled *ClockSpeed* and *CacheSize* which have 3 and 2 soft bins respectively. Again, if there is a soft-to-hard bin map, each of the array elements shown in yellow must be mapped to a hardware bin.

The group has user-settable/readable property *color* of type *String* which serves as the default color for bins under that group, and a read-only property which returns the number of Axes in the group as an unsigned Integer.

1.3 Bin Axes

Axes provide a software mechanism to write reusable tests that do for example, speed binning, i.e., multiple related good device classifications. When there is only one axis in a group, be it named or anonymous, the STIL notion of a bin maps directly to what most testers define as a bin. When there is more than one axis, each cell in the resulting array maps directly to what most testers define as a bin.

An Axis has a name which is optional if there is only one Axis. The name must be unique in the bin group containing the Axis. In addition, it has a user-settable property that determines which soft bin, the one with the lowest or highest index, is mapped to a hard bin when more than one Bin on the Axis is set. The lowest is the default.

1.4 Bins

1.4.1 Null Bin

The null bin is used to define generalized actions. For example, a TestBase fail action for most production tests or flows derived from it, is to bin and stop, i.e., a single action that takes a bin argument. If the argument is null, it neither sets the bin nor stops, if it is a user-defined bin it sets the bin and then stops[2].

[2] Set TestBase fail bin data member to NoBin, then override the fail bin data member from a derived test or flow with a user-defined bin. Stop action semantics unwind the stack. No post actions are carried out after the stop action in its local postaction context. No Flownodes or Flownode post actions are executed after the stop action but the postactions of containing Flows are, i.e., in a hierarchy of Flows, each Flow skips subsequent Flownodes but executes its own postactions.

1.4.2 User Defined Bins

Data Type	Data Description	Comment
Const String	color	Hexadecimal RGB value or text, e.g., red. Used for print or display purposes. Default value is the group color.
Boolean	enable	Disabled bins are treated as no-ops at run-time. Bins are enabled by default.
Const String	general descriptor	May require several, e.g., character for a wafer map, short string for restricted real estate descriptor, and verbose descriptor.
Const enumerated/string/id	name	Allow for conversion to number or spec storage ? See *user defined name* in 1450.0.
Const Integer	number	Bin number is optional but must be unique per Axis. Default value is the index (which starts at zero) + 1.
Const Boolean	pass/fail	Whether a bin is a pass or a fail bin is determined by the group in which it is defined
Mutable Integer	retest	Unsigned, non-zero causes retest on fail up to N times[3]

Table 3: User-settable soft bin properties

Table 3 shows only the bin properties that are programmed directly by the user via definition syntax. User-defined bins may be referred to by name or number and interrogated for additional information (see section 1.9 and Table 4).

Data Type	Data Description	Comment
Const Integer	index	An index into the array of bins (Axis) where the bin resides (unsigned).
Const Integer	counter	The number of times a bin has been set. See section 1.7 and Figure 6 (unsigned)
Const Boolean	is_set	True if the bin is set. See Figure 6.

Table 4: Read-only soft bin properties

1.5 Hard Bins

Same conceptual model and syntax as user-defined soft bins (see

[3] If the program stops and retest is non-zero, retest is decremented and an OnRetest event is generated. If retest is zero on stop, the program stops and restores retest's original value.

Table 3 and Table 4) if required. Only 1 Axis can legally be specified for hard bins, anonymous or named.

1.6 Bin Setting and Arbitration

Bins are automatically unset by the OnStart event handler before any Test or Flow is executed. When a bin is set, it remains set until the next OnStart event or a ClearBin statement is encountered. Bins may be set only in the context of FlowNode, Test, and/or Flow post/pass/fail actions via one of the mechanisms described in Table 5.

Mechanism	**Parameters**	**Action**
SetBin	1. bin name, number, or variable	Set bin and continue.
SetBinStop[4]	1. bin name, number, or variable 2. Boolean ContinueOnFail	Continue if parameter 1 is the null bin, otherwise set bin and based on parameter 2 either stop (generates OnFinish event) or continue.
SetBinRetest	1. bin name, number, or variable 2. Boolean ContinueOnFail 3. Integer retest	Unsigned Integer: same as SetBinStop with additional retest actions. Retest schemes: a. If the program stops and retest is non-zero, retest is decremented and an OnRetest event is generated. If retest is zero on stop, the program stops and restores retest's original value. b. make retest Boolean and use an unsigned Integer data element (in the softbin group container?) to control the maximum number of reprobes. c. provide bin independent retest syntax.

Table 5: Soft Bin Setting Mechanisms

Looking back on the Table 2 example, independent tests may reach different conclusions regarding device speed, e.g., an at-speed functional test passes the device at 2.93GHz whereas a timing test determines that the device meets its 3.00GHz spec. For various purposes, hardware binning being among them, one of the two bins, presumably the lesser 2.93GHz bin is chosen to act upon.

[4] *Stop*, a related mechanism, has nothing to do with binning.

If, in the Table 2 example, the at-speed functional test fails and a timing test still determines that the device meets its 3.00GHz spec, then the fail bin should be acted upon.

If multiple fail bins are set the lowest (or highest, user-settable) index bin is acted upon.

Additionally, the syntax provides means for interpreting any combination of pass and fail bins that have been set.

1.7 Counters

1.7.1 Soft Bin Counters

Proposed: there is a counter group for each object in the binning hierarchy beginning with the top level. Each counter in the group is initialized/re-initialized by a different event (see
Table **6** for a list of supported events) and automatically incremented when a bin is set[5]. The counters are chained such that incrementing a group at the lowest level, increments every group above it. The entire structure is repeated for each site.

1	**On[Load]**
2	**On[LotStart]**
3	**On[WaferStart]**
4	**On[Retest]**
5	**On[Start]**

Table 6: Counter group consists of one counter for each of these events

The Table 1 / Table 2 example is used to describe the counters mechanism. Assuming single site, there are 75 counters in this example, one group per each of the fifteen objects in the hierarchy shown in Table 7 times five counters per group, one counter per event shown in
Table 6.

[5] The language provides read-only access to these counters.

As the events in

Table 6 occur, each of the corresponding counters in the 15 groups of

Table 7 is initialized.

1	**Top Level**	**Soft Bin Groups**	**Axes**	**Bins**
2		Pass		
3			ClockSpeed	
4				3.00GHz
5				2.93GHz
6				2.66GHz
7			CacheSize	
8				8Mb
9				4Mb
10		Fail		
11			Anonymous	
12				ContactOpens
13				ContactShorts
14				Functional
15				Timing

Table 7: a counter group per row per site

When a bin is set, the corresponding counter hierarchy is incremented. For example, a timing test sets the 2.93GHz bin, incrementing the 2.93GHz counter group (Table 7, row 5), the ClockSpeed counter group (Table 7, row 3), the Pass counter group (Table 7, row 2), and the Top Level counter group (Table 7, row 1).

Support bin yield alarms (too high, too low).

1.7.2 Hard Bin Counters

Same as soft bin counters.

1.7.3 Counter Based Events

These are events triggered by counter contents or calculations involving counter contents and handled by EntryPoints, e.g., scrubbing and/or re-probing for up to three consecutive contact test failures, characterizing every Nth good device, or stopping and raising an alarm when device yield drops below a certain point..

1.8 User-defined Device Specifications

Proposed: for software maintenance purposes it is useful to allow one or more sets of user-defined device specs per axis. The intent is to be able to write tests that can iterate

simultaneously over axis bins and their corresponding specs. To be used for this purpose, each named specification must be able to hold a one or more values (min, max, or min and max) per bin[6].

In our example, one set of device specifications may be used for ac timing tests for a micro-processor. A single timing test can then iterate over each spec, e.g., data setup time, iterate over the timing values associated with the 3.00GHz, 2.93GHz, and 2.66GHz portions of that spec and bin the device accordingly.

Adding another spec, e.g., data hold time, then becomes a matter of adding in effect, a named variable with three values (assuming minimums only), one for each bin under *ClockSpeed*. Adding another *ClockSpeed* becomes a matter of adding a bin and a corresponding value to each of the specs associated with the *ClockSpeed* axis. Disabling one or more *ClockSpeed* bins can be implemented as a simple Boolean function.

1.9 Data Access

Proposed: access is provided for both stored and processed data, e.g., the contents of a counter (stored) and the dominant bin if more than one is set[7] (processed).

Stored data may be accessed via the object hierarchy using, e.g., a dot syntax: *a.b.c* where data element *c* is contained by object *b* which is contained by object *a*. If *b* is an array then *c* of the first element of *b* may be accessed by, e.g., *a.b[0].c* (consider associative arrays for binning, i.e., the index may be other than numeric).

Processed data may be accessed via the object hierarchy using, e.g., a dot syntax: *a.b.c()* where function *c* is a member of object *b* which is contained by object *a*.

Appropriate member functions need to be defined for each object type.

1.10 Soft to Hard Bin Maps

STIL.4 provides the capability to define zero or more named sot to hard bin maps and a mechanism for selecting the currently active map. Each BinMap must contain a reference to the soft Bin definitions and if mapping is done the hard Bin definitions to be mapped.

A single mapping in a bin map, maps one or more soft bins to a single hard bin where each soft bin reference must be from a different axis. Bin references in a single mapping must all refer to Pass or all refer to Fail bins. If both Pass and Fail Bins are set, mapping is done on the basis of Fail Bins.

Each map has no less than one entry for each cell in the array of enabled software bins formed by the bin axes, and no more than one entry for each cell in the array of enabled

[6] STIL.0 *Spec* and *Category* syntax may be applicable here: *Spec* per *Axis*, *Category* per *Bin*.
[7] The dominant bin is used for soft to hard bin mapping, for example.

and disabled software bins[8]. For single axis bin definitions, that means one entry per software bin. For multi-axis arrays, that means one entry for each combination of software bins. Each entry maps a software bin array cell to a hardware bin.

Assuming all bins are enabled, the example used in this document requires one entry for each fail-bin in the anonymous one dimensional array, and one entry for each of the combinations of ClockSpeed/CacheSize axis bins, i.e., 3 x 2 equals 6 entries.

2 Syntax

2.1 Definitions

```
softbin_property =
    <
      Color = String;              |     // Hex RGB or name
      Enable = Boolean;            |
      Number = Integer;            |     // Default: index + 1, no 2 Axis bins have same number
      Retest = Integer;            |     // Unsigned, Hold off on semantics
      Terse = String;              |
      Verbose = String;            |
      WafermapChar = Character;
    >
softbin_definition =
    Bin SOFTBIN_NAME; |                  // Sets property Name
    Bin SOFTBIN_NAME { softbin_property* }
```

Figure 1: Soft bin definition

[8] In other words, disabled bin cell entries are optional.

```
(SoftBinDefs|HardBinDefs) BINDEFS_NAME {
  (ContinueOnFail = Boolean;)      // Contained bin default, false if unspecified
  Pass {
    (Color = String;)              // Contained bin default, "green" if unspecified
    (softbin_definition)+      |
    ( Axis AXIS_NAME {
      (MapBinLowest|MapBinHighest);// Map bin with highest/lowest index when 2+ set
      (softbin_definition)+
      } )+
    }

  Fail {
    (Color = String;)              // Contained bin default, "red" if unspecified
    (softbin_definition)+      |
    ( Axis AXIS_NAME {
      (MapBinLowest|MapBinHighest);// Map bin with highest/lowest index when 2+ set
      (softbin_definition)+
      } )+
  }
}
```

Figure 2: Soft bin definition block

A `SoftBinDefs` example in line with Table 1 and Table 2 follows:

```
SoftBinDefs bindefs {
  Pass {
    Axis ClockSpeed {          // Pass index 0
      MapBinHighest;
      Bin "3.00GHz";           // ClockSpeed index 0
      Bin "2.93GHz";           // ClockSpeed index 1
      Bin "2.66GHz";           // ClockSpeed index 2
    }
    Axis CacheSize {           // Pass index 1
      MapBinHighest;
      Bin "8Mb";               // CacheSize index 0
      Bin "4Mb";               // CacheSize index 1
    }
  }
  Fail {
    Bin ContactOpens;
    Bin ContactShorts;
    Bin Functional;
    Bin Timing;
  }
}
```

```
unary_bin_reference =
     <
       ((Pass|Fail.)(Axes[AXIS_NAME|AXIS_INDEX].)Bins[BIN_NAME|BIN_INDEX]) |
        BIN_NAME|BIN_NUMBER
     >

softbin_reference  = unary_bin_reference
hardbin _reference = unary_bin_reference

(BinMap BIN_MAP_NAME {
  SoftBins SOFT_BIN_DEF_NAME;
  HardBins HARD_BIN_DEF_NAME;
  ( Map softbin_reference+ hardbin _reference; )*
})*
```

Figure 3: Soft to hard bin mapping block

Here are two equivalent pass soft to hard bin mapping examples, first:

```
Map Pass.Axes[CacheSize ].Bins[    "8Mb"]
    Pass.Axes[ClockSpeed].Bins["3.00GHz"]  1;
```

secondly, since the ids are unique within the SoftBinDefs context[9]:

```
Map "8Mb" "3.00GHz" 1;
```

For when the fail axis is anonymous, here are four equivalent examples:

```
Map ContactOpens   5;
Map Bins[ContactOpens]   NullBin;
Map Fail.Bins[ContactOpens]   5;
Map Fail.Axes[0].Bins[0]   5;
```

The `Map` statement requires that all bin descriptors refer to the same group, i.e., either `Pass` or `Fail`, and that each bin listed comes from a different axis in that group. Each `Map` statement must contain a unique combination of software bins. The totality of `Map` statements must cover all combinations. For the example used in this document, that requires 3 x 2 = 6 `Map` statements to map all `Pass` bins.

Changes from P1450-4-D18-SyntaxSummary11-28-2007.pdf:

- Removed Integer option in "**`Bin SOFTBIN_NAME (`***Integer***`);`**" because: in the context of *Axis* it is unclear what the relationship between that Integer and the index of a *Bin* is. A gap in the number sequence and duplicate integers pose problems with iterating over *Bin*s if that Integer is the index.

[9] The hard bin number is unique because it has to refer to a Pass bin since all Map statement bin references must be either Pass or Fail.

- Added individual soft bin properties as per Table 3. This captures the intent of the Integer above using a different syntax.
- Added keyword Map to each line under *BinMap* to conform with 1450.0 general STIL statement form: **Keyword** (OPTIONAL_TOKENS)*;
- Required at least one soft bin for *Map* statement.
- Substituted white-space for `->` in statements inside the *BinMap* to mimic 1450.1 *NameMaps* syntax (that syntax puts the use of keyword Map in question):

```
NameMaps VECTOR_ASSOCIATIONS {
    Signals {
        "A" "top_test.PI[0]";
        "B1" "top_test.PI[1]";
        "C1" "top_test.PI[2]";
        "D11" "top_test.PI[3]";
    }
    SignalGroups {
        _PI "top_test.PI";
        _PO "top_test.PO";
    }
    Variable { _PATCOUNT "PATTERN"; }
}
```

2.2 Usage and Data Access

2.2.1 Group Property Access

```
group_property =
  <
   Color          |    // String (hexadecimal RGB or name, e.g., red
   countSetBins   |    // Integer,  unsigned
   Axes.Size           // Integer, number of Axes in the group, always > 0
  >
```

Figure 4: Group Property Access

A group property access example follows:

```
Pass.Axes.Size
```

2.2.2 Axis Property Access

```
axis_property =
   <
    Bins.Size                  // Integer, number of Bins on the Axis
    Name                       // String
   >
group = < Pass|Fail >

group.Axes[Integer|AXIS_NAME].axis_property
```

Figure 5: Axis Property Access

Two equivalent axis property access examples follow (numeric index must be unsigned):

```
Pass.Axes[ClockSpeed].Bins.Size
Pass.Axes[0].Bins.Size
```

2.2.3 Bin Property Access

```
counter_reset_event =
   <
    On[Load|LotEnd|LotStart|Reset|Start|WaferEnd|WaferStart|
       <UserKeyword>]
   >
bin_property =
   <
    Color                          |    // String
    ContinueOnFail                 |    // Boolean – Reconsider under Phase2
    counter.counter_reset_event    |    // Unsigned
    Enabled                        |    // Boolean
    Index                          |    // Unsigned
    IsFailBin                      |    // Boolean
    isSet                          |    // Boolean
    Name                           |    // String
    Number                         |    // Integer
    retest.(current|Original)      |    // Unsigned
    Terse                          |    // String
    Verbose                        |    // String
    WafermapChar                        // Character
   >
group = < Pass|Fail >

group.Bins[Unsigned|SOFTBIN_NAME].bin_property  |
group.Axes[Unsigned|AXIS_NAME].Bins[Unsigned|SOFTBIN_NAME].bin_property
```

Figure 6: Bin Property Access

For example, the expression[10]:

```
Pass.Axes[ClockSpeed].Bins["3.00GHz"].Index
```

yields `0`, the index of bin ”3.00GHz”, whereas:

```
bindefs.Pass.Axes[ClockSpeed].Bins[0].Name
```

yields the bin name string `”3.00GHz”`[11], assuming an instance of type `SoftBinDefs` or `HardBinDefs` named `bindefs` exists.

[10] This form requires that the program knows the in-use instance, e.g., `bindefs`.

[11] The returned string does not include quotes, regardless of whether they were used in the bin name or not unless *String* provides a mechanism for getting an unquoted version.

Expression:

```
Pass.ClockSpeed[”3.00GHz”].counter.OnLoad
Pass.ClockSpeed[”3.00GHz”].counter.On[Load]
```

yields an Integer representing the number of times bin `”3.00GHz”` has been set since the test program was loaded.

2.2.4 Setting Bins

Binning code is restricted to specific blocks[12]. For tests and flows, that is in the PostAction, PassAction, and FailAction blocks. For flow-nodes, that is in the PostAction and exit-port action blocks.

Statements `SetBin`, `Stop`, and `SetBinStop` are provided because they represent commonly used actions provided by most testers and are easily parsed by both human and machine, e.g.., it is easier to find `SetBinStop` occurrences than the equivalent **`If`** statement syntax provided below.

[12] Re-evaluate with respect to `Skip` action, i.e., maybe PreAction binning should be allowed.

```
unary_ bin_expr =
   <
    (Pass | Fail.)(Axes[BIN_AXIS_NAME | AXIS_INDEX].)Bins[BIN_NAME | BIN_INDEX] |
    BIN_NAME |
    BIN_NUMBER |
    BIN_VAR_NAME
   >
unary_ axis_expr =
   <
    (Pass | Fail.)Axes[BIN_AXIS_NAME | AXIS_INDEX] |
    BIN_AXIS_NAME
   >
unary_bin_identifier =  // No longer needed ?
   <
    Bin = Bin;                      |
    Index = Integer;                |
    Name = User_defined_name;       |     // See 1450.0
    Number  = Integer;
   >
multi_bin_expr =
   <
   All                              |     // All Bins in BinDefs
   All Pass|Fail                    |     // All Bins in group
   All unary_ axis_expr             |     // All Bins in Axis
   >

SetBin unary_ bin_expr | multi_bin_expr;       |
SetBinStop unary_ bin_expr | multi_bin_expr; |
SetBin { unary_bin_identifier }                | // No longer needed ?
SetBinStop { unary_bin_identifier }

ClearBin unary_ bin_expr | multi_bin_expr;

Stop;       // No binning: listed because it is related to setBinStop
```

Figure 7: Binning Actions

The `SetBin` statement sets the bin only, i.e., does not stop. The `Stop` statement stops unconditionally[13]. `SetBinStop` statement semantics are equivalent to the following[14]:

[13] Bubbles up terminating the initiating EntryPoint Test or Flow, i.e., Stop is the last instruction executed in the action block that contains the Stop statement. All FlowNodes that would normally be executed after the Stop statement if any, are skipped but Post, Pass and/or Fail actions of containing flows are executed.

[14] Makes assumptions about syntax which is incompletely defined in syntax document.

```
If FailBin != NoBin
  { SetBin This.FailBin; If ContinueOnFail == False
Stop; }
```

2.2.5 NoBin Property Access

NoBin is a special bin defined by the standard. It has no user-settable properties. Figure 8 shows the values returned when *NoBin* properties are interrogated. Refer to Figure 6 for the definition of *counter_reset_event*.

```
Color                          // String "grey"
ContinueOnFail                 // Boolean True
counter.counter_reset_event    // Integer,  default = 0, incremented when set
Enabled                        // Boolean True
Index                          // -1
IsFailBin                      // Boolean False
isSet.counter_reset_event      // Boolean,    default = False, becomes True when set
Name                           // String "NoBin"
Number                         // 0
retest.(current|Original)      // Integer 0|0
Terse                          // String "" (empty)
Verbose                        // String "" (empty)
WafermapChar                   // Character ' ‘ (space)
```

Figure 8: NoBin Properties

Issues:

- Allow Bin assignment, i.e., Bin::operator=(const Bin&), ref Index, Number
- MaxUnsigned or Infinity are currently not keywords. What are the merits of each, assuming Infinity could be used across types, e.g., Integer, Real, Seconds, etc

2.2.6 Re-probe

Issue: define format easily mapped to multiple targets. Is there such a thing ? Delay to phase II ?

2.3 Examples

3 Spill-over Issues

Some issues arising in the context of binning spill over into other areas of the language and/or vice versa, begging for some co-ordination to minimize unnecessary proliferation of rules and concepts. These are merely alluded to here, expecting resolution in the greater context:

a. Prior to BinDef property access syntax, the in-use BinDefs must be known (currently described in TestProgram block which comes last). A using statement was under discussion.
b. Initialization: when defining an object such as a variable, it is useful to be able to specify the event that initializes/re-initializes it, e.g., one counter may be re-initialized by event *OnWaferStart*, another by event *OnLoad*, etc.
c. Object member functions: it is useful to specify member functions for integral objects, e.g., string.length(), array.dim(), array.dim(0).length(), etc.
d. It would be helpful tp provide string/number-with-units conversion.
e. Do we provide syntax to allow the user to trigger an EntryPoint event, e.g., trigger OnStart or OnRetest for retest ? Support more standard or user-defined events, e.g., low bin yield alarms ?
   Delayed until phase II.
f. IO syntax, e.g., cin, cout, cerr, clog, stringstream, file IO, formatting, etc.
   Delayed until phase II.
g. A context sensitive keyword for the null bin is required, e.g., NullBin or NoBin.
   Settled on NoBin.
h. Need OnRetest event/EntryPoint (initializes `retest.current` along with OnLoad).
   Done.

In order to adequately describe binning behavior or syntax, some clarification regarding the semantics and/or syntax associated with existing language elements is necessary. Clarification is needed for the:

a. semantics regarding *On Start* and *On SiteStart* EntryPoints:
   On SiteStart has been dropped.
b. use of dot 0 Spec/Category syntax, i.e., how do we access, e.g., *Meas* under *Spec* `tmode_spec`, or *Max* under variable `tplh` under *Category* `tmode` under *Spec* `tmode_spec` ?
   Do we need to access a *Meas* for every *Category*, i.e., min, typ , max (device with different timing may have different measurement result for the same *Spec*) ?
c. dot 4 provided basic test definitions: syntax requirements may change depending on what pre-defined test elements exist in dot 4, e.g., if all we provide is a functional test then in order to be define an DC or AC parametric test, syntax that permits the iterative alteration of a specific level or timing edge is required. If dot 4 provides basic tests to perform linear and binary searches that syntax may take on a different character. Run-time efficiency may be affected by our choices.

Standard test definitions will exist in the form of suggestions, actual standardization delayed until dot5.

d. definition of dimensioned array variables.
Done

e. Does *String* provide a mechanism for getting unquoted version (ref string/ number-with-units conversion) ?

E. J. Wahl
ejwahl@att.net
(304) 647-4784

Table of Contents

1 Conceptual Model 1
1.1 Top Level Container 1
1.2 Bin Groups 1
1.3 Bin Axes 2
1.4 Bins 2
1.4.1 Null Bin 2
1.4.2 User Defined Bins 3
1.5 Hard Bins 3
1.6 Bin Setting and Arbitration 4
1.7 Counters 5
1.7.1 Soft Bin Counters 5
1.7.2 Hard Bin Counters 6
1.7.3 Counter Based Events 6
1.8 User-defined Device Specifications 6
1.9 Data Access 7
1.10 Soft to Hard Bin Maps 7
2 Syntax 8
2.1 Definitions 8
2.2 Usage and Data Access 11
2.2.1 Group Property Access 11
2.2.2 Axis Property Access 11
2.2.3 Bin Property Access 13
2.2.4 Setting Bins 14
2.2.5 NoBin Property Access 16
2.2.6 Re-probe 16
2.3 Examples 16
3 Spill-over Issues 17