

1

Revision History

2

Date	Rev	Init	Change
02/29/08	D0.18	jo	Modified <code>bypass_stmt</code> - removed <code>portindex</code> as option in and updated semantics if <code>VAR_NAME</code> is type integer. Modified <code>ExitPort</code> statement to remove <code>portindex</code> (used as a bypass option). Ordinal index order (starting with 0) is now used instead. <code>PORTLABEL</code> is now required, not optional.
3/24/08	D0.18	jo	Modified <code>TestType</code> , <code>FlowType</code> syntax. Changed “UseDefaults” to “Inherit <code>TestBase</code> ”. Clarified semantics if “Inherit” statement is absent.
3/27/08	D0.18	jo	Added revision table to keep track of changes.
5/15/08	D0.19	jo	Added <code>softbin</code> attributes and <code>bin</code> property access syntax (per Ernie’s recent writeups).
5/22/08	D0.20	jo	Added <code>(Const)</code> attribute to <code>var_type</code> ; Removed <code>Operator</code> attribute from <code>var_elements_stmt</code> (may be added back if/when we determine it’s needed). Updated <code>TestBase</code> definition per discussions with Doug and Ernie. Removed <code>ReturnState</code> keyword as option in <code>exit_port_stmt</code> . Modified <code>Return</code> keyword to remove optional (<code>integer_expr</code>) token. <code>ExecResult</code> from <code>TestBase</code> (or derivatives) is used instead.
5/23/08	D0.21	jo	Removed <code>Return</code> keyword from <code>exit_port_stmt</code> . Moved <code>Stop</code> keyword from <code>exit_port_stmt</code> to <code>action_stmt</code> . In <code>PassActions/FailActions</code> of <code>test_elements_stmt</code> and <code>flow_elements_stmt</code> , and in <code>TestBase</code> , <code>TestType</code> , and <code>FlowType</code> syntax, replaced <code>exit_port_stmt</code> with <code>action_stmt</code> .
7/02/08	D0.22	jo	Added optional <code>TestExec</code> statement to <code>TestBase</code> syntax. <code>STIL .4</code> definition definition of <code>TestBase</code> will not include <code>TestExec</code> . Added <code>SetBinStop</code> alternative to <code>action</code> statement definition. Changed initial value of <code>TestBase Bin</code> parameter from undefined to <code>NoBin</code> . For component values of <code>NoBin</code> , see section 3.2.5 of Ernie’s bin document. In “Default Flow Node” definition, replaced <code><exec_object_name></code> with <code>EXEC_OBJECT_NAME</code> to adhere to the <code>STIL BNF</code> notation rules. Still need to agree on the alternatives for binning syntax shown in Fig. 7 of that same doc. What does <code>Stop</code> or <code>SetBinStop</code> mean from the actions of a <code>Test</code> , as opposed to the actions of a <code>FlowNode</code> ?
8/14/08	D0.23	jo	<ul style="list-style-type: none"> • In <code>initial_value_stmt</code>, updated syntax so that <code>test_element_stmt</code> and <code>flow_element_stmt</code> occur after <code>InitialValue</code> keyword, to bring <code>TestType</code> and <code>FlowType</code> initial value statements in line with those of all other types. Note the inclusion of open/close braces (<code>{ }</code>) to enclose those statements. • Added keyword Port preceding <code>PORTLABEL</code> in <code>ExitPorts</code> statement; removed colon (<code>:</code>) after <code>PORTLABEL</code> in <code>exit_port_stmt</code>. Addition of <code>Port</code> keyword makes the colon unnecessary.

			<ul style="list-style-type: none"> • Modified InitialValue syntax (for variables) to include array initialization (using a comma-separated list). • Added keyword Return <i>integer_expr</i> to <i>exit_port_actions</i>. This causes an immediate return to the caller (either a Test or a Flow). The integer return value sets the FlowNode variable Result. Result is a predefined keyword which can be used only in the FlowNode PostActions or ExitPort clauses. • Updated semantics of Stop statement. Stop now returns immediately to EntryPoint initiator, with appropriate pass/fail result. See text for more details. • Added the option (<i>flownode_stmt</i>)* to <i>test_elements_stmt</i>. This will allow tests to specify a sequence of flownodes for each instance created, as well as in the type definition. • Allow a TestType to derive from a FlowType. This is to allow Flows to be turned into Tests (and thus, presumably, hiding the internals of that flow from graphical Flow Editor tools). • Removed TestDefaults block. Definitions of TestBase and the default flow type stil_dot_4_default need no container; the STIL.4 default flownode definition must occur outside the scope of a Flow or Test. If more than one default definition occurs, the last one listed is used. • Modified the syntax of FlowNode to include the SetResult statement, the introduction of the FlowNode variable Result, and the keyword CurrentExec, which is an alias for the actual Test or Flow named in the TestExec statement. CurrentExec can be used ONLY in the SetResult, the <i>exit_port_expr</i>, or <i>exit_port_stmt</i> statements. • Change STIL 1.0 Flow extension date from 2007 to 2008. • Added angle brackets (< >) to syntax notation for <i>softbin_definition</i>. Expanded SetBin and SetBinStop syntax to include BIN_VAR_NAME, as well as explicit bin name. Added <i>bin_expr</i> notation. • Modified FlowType syntax to allow FLOW_TYPE_NAME to be optional. This allows an unnamed FlowType block which can be used as a default FlowType. See section 6.9 of IEEE_1450_1999_0 (dot0) for additional information on domain names. • Updated STIL.4 default definitions of TestBase, FlowNode, and FlowType. • Change Test and Flow instantiation syntax from Flows { } / Tests { } to Flow FLOWTYPE FLOWNAME; / Test TEST_TYPE TEST_NAME; • Added clarifications about integer values for Pass, Fail, True, and False.
9/04/08	D0.24	jo	<ul style="list-style-type: none"> • Removed SetReturn statement and FlowNode variable Result (per discussion at WG meeting of 09/04/08). • Clarified semantics of Stop action statement. • Added Exit and Return statements. Clarified semantics of those statements. • Updated default FlowNode definition so that FlowNode FAIL

			<p>ExitPort actions sets the execResult of the Test or Flow containing the FlowNode.</p> <ul style="list-style-type: none"> • Updated semantics of default FlowNode. Not required, but if not provided by the user, then substitution of TestExec statements for FlowNodes is NOT allowed. • Updated TestBase definition. • Changed spelling of TestBase mandatory fields from execResult to ExecResult and failBin to FailBin. • Added NullBlock keyword to <i>initial_val_elements</i> metatype. Used for setting <i>stil_block_type</i> Variables or Parameters to a Null block. For TestType or FlowType parameters, this is treated as an optional parameter (the user can, but is not required to, provide a valid override value). If the user does NOT provide a valid override value, the parameter will not be used. If the user does provide a valid overrided value, the parameter will be used.
9/18/08	D0.25	jo	<ul style="list-style-type: none"> • Added semantics regarding user definition of TestBase and default FlowNode (each can be defined or redefined by the user any number of times UNTIL first use. After first use, redefinition is NOT allowed). • Added MultiSiteSerial field to TestBase. Normally initialized to False, meaning that when this test is used in a flows which are running on multiple sites, all are executed in parallel. If set to True, then all sites executes this test (and its containing FlowNode) serially (one after the other; order of execution of sites is not specified). • Updated semantics for Next <i>exit_port_stmt</i>. For the last FlowNode in a sequence, Next; is the same as Return;. This does NOT extend to the “Next FLOW_NODE_NAME;” form, however.
10/02/08	D0.26	jo	<ul style="list-style-type: none"> • Removed Test and Flow from <i>stil_block_type</i>; removed <i>test_elements_stmt</i> and <i>flow_elements_stmt</i> from <i>initial_value_elements</i> metatype. • Modified Parameter syntax for FunctionDefs so that direction (In, Out, InOut) is listed first, prior to the type and name definition, rather than after the type and name definition. This syntax now mirrors that of TestBase, TestType, and FlowType. • Restructured BNF for <i>initial_value_stmt</i> to be more concise. • Restructured BNF for Parameter blocks to use <i>var_elements_stmt</i>. • Updated <i>var_elements_stmt</i> syntax used for specifying arrays, and for initializing scalar or array variables. Changed metatype name for initial value elements from <i>initial_value_elements</i> to <i>initial_value_element</i>. • Added keyword None to <i>initial_value_element</i>. Indicates an optional parameter when used in Parameter blocks of TestTypes, Tests, FlowTypes, or Flows (i.e., does not need to be specified by the user, since it has an initial value, but is ignored by runtime code). Keyword None is not allowed for initial value assignments in Variables blocks.

10/09/08	D0.26	jo	<ul style="list-style-type: none"> • Modified <code>flow_node_stmt</code> to include bare <code>TestExec</code>, which implies that the default <code>FlowNode</code> is used around the <code>TestExec</code>. • Update <code>TestBase</code> default definition to use new “initial value” syntax
10/16/08	D0.26	jo	<ul style="list-style-type: none"> • Removed multiple <code>execute_stmt</code> form of <code>TestExec</code> from <code>flownode_stmt</code> (TestExec { <code>execute_stmt</code> }[*]). This capability was determined to be redundant, and therefore not needed. Changed TestExec NullExec; statement to TestExec; (NullExec keyword added no additional value). • Removed Title <code>STRING</code>; statement from <code>flow_elements_stmt</code> and <code>FlowType</code>. Again, not needed, since each object will have its own name (ID), and if a different title is needed, parameters or variables can serve this purpose.

3 Syntax summary for 1450.4

```

4   stil_block_type =
5       < Category
6         | DCLevels
7         | DCSequence
8         | DCSets
9         | Environment
10        | MacroDefs
11        | Pattern
12        | PatternBurst
13        | PatternExec
14        | Procedures
15        | ScanStructures
16        | Selector
17        | SignalGroups
18        | Signals
19        | Timing >
20
21   real_var_type =
22       < Capacitance // F (Farads)
23         | Compound // combinations of types
24         | Current // A (Amperes)
25         | Double
26         | Frequency // Hz (Herz)
27         | Gain // db (Decibels)
28         | Inductance // H (Henries)
29         | Length // m (Meters)
30         | Power // W (Watts)
31         | Real
32         | Resistance // Ohm (Ohms)
33         | Temperature // Cel (Degrees Celsius)
34         | Time // s (Seconds)
35         | Voltage // V (Volts) >
36
37   var_type =
38       // Boolean values: True/False or Pass/Fail
39       (Const) < Boolean | Integer | SignalRef | SignalVariable | String | real_var_type | stil_block_type >
40
41   initial_value_element =
42       < integer_expr // allow if var of type integer_expr
43         | sig_ref_expr // allow if var of type sig_ref_expr
44         | < True | False > // allow if var of type Boolean – from dot1, 0 = TRUE, 1 = FALSE.
45         | string // allow if var of type String
46         | wfc_string // allow if var of type SignalVariable
47         | real_expr // allow if var of type real_var_type
48         | BLOCK_NAME // allow if var of type stil_block_type - assign from predefined block
49         | None // allow for all types in Parameter blocks ONLY. Not allowed for
50                 // initial values in variables blocks. If parameter is initialized to None
51                 // and not overridden to a value other than None during instantiation,
52                 // it is ignored during execution.
53       >
54
55   initial_value_list = initial_value_element \ initial_value_list, initial_value_element
56

```

```

57 // Use either the explicit softbin name, or use a softbin variable name (variable of type Bin).
58 bin_expr = (BIN_AXIS.)SOFTBIN_NAME | BIN_VAR_NAME
59
60 action =
61     < VAR_NAME = expr; // Scalar variables
62     | VAR_NAME[integer_expr] = expr; // Array variables
63     // Only one element of each bin axis can be active at any one time. For 3 separate axes, you can
64     // have 3 separate softbins (one from each axis) set – the binmap handles the various intersections
65     // when the part is binned.
66     | SetBin bin_expr;
67
68     // Return to the caller, “bubbling up” through the levels of FlowNodes, Flows and Tests, until the
69     // initiating EntryPoint is reached. At each level, execute the PostActions and PassActions or
70     // FailActions for Tests and Flows, but SKIP execution of FlowNodes (including any FlowNode
71     // PostActions or ExitPort actions).
72     | Stop; // terminate the initiating On* condition.
73
74     // SetBinStop is equivalent, semantically, to:
75     If (<bin_name> != NoBin) { SetBin <bin_name>; Stop;} Else { // No Action }
76     | SetBinStop bin_expr;
77
78     // Exit – Immediately stop execution of current block (test, flow, or flow node), and jump directly to the
79     // initiating EntryPoint (On* condition) for end-of-test processing. The pass/fail result returned to the
80     // EntryPoint initiator (which would normally be the ExecResult of the Test or Flow specified by the
81     // EntryPoint), is specified by the integer_expr token of the “Exit <integer_expr>;” statement. This
82     // statement is generally intended to be used for exceptions, such as hardware failures, which require
83     // immediate termination of test execution.
84     | Exit integer_expr; // terminate the initiating On* condition.
85
86 action_stmt =
87     < If boolean_expr | Else If boolean_expr | Else { // Usual rules for If/Else If/Else apply
88         ( action )*
89     }
90     | ( action )* >
91
92 bypass_actions =
93     <
94     // For FlowNodes, Flows, and Tests
95     // Skip only Test execution or FlowNode sequence execution, resume at entry to PostActions Block
96     // Normal processing continues from there. Execute PostActions, and PassActions/FailActions (as
97     // determined by execResult of Test or Flow), or ExitPort selection Actions (as determined by flownode
98     // exit port selector). Selection of Pass/Fail path (for Tests and Flows) is controlled by execResult (which
99     // can be forced to a desired state prior to Bypass statement). Selection of ExitPort path (for flownodes)
100    // is also controlled by boolean expressions – typically, the return status of the test executed by the
101    // FlowNode. This return state (or value of any other boolean expression) can be forced to a desired state
102    // prior to the Bypass statement.
103    Bypass;
104
105    // For Flows and Tests only.
106    // Skip Test and PostActions, resume execution at entry to PassActions/FailActions.
107    // If Pass or Fail specified, follow that path. Otherwise, VAR_NAME is a string variable which specifies
108    // either PASS or FAIL. If using VAR_NAME, and it’s an empty string, no bypass action occurs
109    // If SkipActions specified, don’t execute PassActions/FailActions (equivalent to a return,
110    // since there’s no branching from a Test or Flow)
111    | Bypass (GoTo <Pass | Fail | VAR_NAME > ( SkipActions ) );
112

```

```

113 // For FlowNodes only.
114 // Skip Test and PostActions, resume execution at entry to specified ExitPort.
115 // If SkipActions specified, don't execute don't execute the exit port actions.
116 // PORTLABEL is a string specifying a port label. VAR_NAME is either a string variable or an integer variable.
117 // If a string variable, it specifies the bypass exit port by label. If using VAR_NAME, and it's an empty string,
118 // no bypass action occurs. If an integer variable, it specifies the bypass exit port by index (based on the
119 // ordinal order of the exit ports, from top of list to bottom, with the first index being 0.
120 | Bypass (GoTo < PORTLABEL | VAR_NAME > ( SkipActions ) );
121
122 bypass_stmt =
123   < bypass_actions | If boolean_expr { bypass_actions } >
124
125 exit_port_stmt =
126   < action_stmt
127     | Next (FLOW_NODE_NAME); // For last FlowNode in a sequence, "Next;" == "Return;"
128     | Return; // Stop execution of FlowNodes. Return to execution chain of containing Test or
129     // Flow (PostActions, PassActions, or FailActions). ExecResult of containing Test
130     // or Flow can be set either by the FlowNode PostActions or ExitPort actions
131     // (typically, based on the results of the most recent TestExec object), or by the
132     // PostActions of the containing Test or Flow (a typical use here might be to set the
133     // ExecResult of the containing Test or Flow based on some combination of
134     // previously-executed tests). Typically used to construct subflows which can be used
135     // in place of Tests.
136   >
137
138 func_stmt =
139   < FUNC_NAME ; // name of a Function from FunctionDefs block
140   | FUNC_NAME { // name of a Function from FunctionDefs block
141     (VAR_NAME = expr)*
142     (VAR_NAME = [ expr (expr)+];)* // For arrays (?)
143   } // end func_stmt
144   >
145
146 test_elements_stmt =
147   < VAR_NAME = expr;
148   | VAR_NAME = [ expr (expr)+]; // For arrays (?)
149   | PreActions { ( < action_stmt | bypass_stmt )* } )
150   | FuncExec func_stmt | FuncExec { ( func_stmt )* } | ( flownode_stmt )*
151   | PostActions { ( action_stmt )* }
152   | PassActions { ( action_stmt )* }
153   | FailActions { ( action_stmt )* }
154   >
155
156 test_instance_stmt =
157   // create a named Test from a TestType, using the default elements for the TestType
158   TEST_TYPE TEST_INSTANCE_NAME;
159
160   // create a named Test from a TestType, overriding some or all of the default elements of the TestType.
161   // Any element specified in the instantiation (i.e., PreActions, FailActions) completely replaces that
162   // element as specified in the type definition.
163   | TEST_TYPE TEST_INSTANCE_NAME { ( test_elements_stmt )* }
164
165 run_result_item = integer | integer:integer
166
167 run_result_list = run_result_item | run_result_list, run_result_item
168

```

```

169 // Note that exit_port_expr can use single values or run_result_lists on the RHS of an equality expression
170 // i.e., "ExecResult == 0;" or " ExecResult == 0,1,5,7;" or "ExecResult == -3:-1,1,4:6;"
171 exit_port_expr = <integer_expr == run_result_list | boolean_expr >
172
173 flownode_stmt =
174     FlowNode (NODE_NAME) {
175         ( PreActions { ( < action_stmt | bypass_stmt )* } )
176         // NullExec allowed in TestExec statement of FlowNode ONLY in default FlowNode
177         // definition (a FlowNode defined OUTSIDE the scope of a Test or Flow).
178         ( TestExec execute_stmt | TestExec; )
179         ( PostActions { ( action_stmt)* } )
180         ( ExitPorts {
181             ( Port PORTLABEL boolean_expr { (exit_port_stmt)* } )*
182         } ) // end ExitPorts
183     } // end FlowNode
184 | TestExec execute_stmt // Default FlowNode is used around a bare TestExec statement in this context.
185
186 flow_elements_stmt =
187     < VAR_NAME = expr;
188     | VAR_NAME = [ expr (expr)+]; // For arrays (?)
189     | PreActions { ( < action_stmt | bypass_stmt )* } )
190     | ( flownode_stmt )*
191     | PostActions { ( action_stmt )* } )
192     | PassActions { ( action_stmt )* } // end PassActions
193     | FailActions { ( action_stmt )* } // end FailActions
194     >
195
196 flow_instance_stmt =
197     // create a named Flow from a FlowType, using the default elements for the FlowType
198     < (FLOW_TYPE) FLOW_INSTANCE_NAME ; // if FLOW_TYPE is not specified, then the
199         // STIL.4 default FLOW_TYPE is used
200     // create a named Flow from a FlowType, overriding some or all of the default elements of the FlowType.
201     // Any element specified in the instantiation (i.e., PreActions, flownodes, PassActions) completely
202     // replaces that element as specified in the type definition.
203     | (FLOW_TYPE) FLOW_INSTANCE_NAME { ( flow_elements_stmt )* } > // if FLOW_TYPE is not specified, then the
204         // STIL.4 default FlowType (the unnamed
205         // FlowType) is used.
206
207 test_execute_stmt =
208     < TEST_NAME ; // execute a named Test
209     | TEST_TYPE; // create and execute a temporary inline Test (named _INLINE_TEST)
210         // from TestType, using the type's default element value.
211     | TEST_TYPE { ( test_elements_stmt )* } > // Create and execute a temporary inline Test
212         // (named _INLINE_TEST) from TestType,
213         // overriding the type's default element values
214
215 flow_execute_stmt =
216     < FLOW_NAME; // execute a named Flow
217     | FLOW_TYPE; // create and execute a temporary inline Flow(named _INLINE_FLOW)
218         // from FlowType, using the type's default element values
219     | FLOW_TYPE { ( flow_elements_stmt )* } > // Create and execute an a temporary inline Flow
220         // (named _INLINE_FLOW) from FlowType,
221         // overriding the type's default element values
222
223 execute_stmt = < test_execute_stmt | flow_execute_stmt >
224

```

```

225
226   var_elements_stmt =
227       // Do we need an Operator attribute? Intended to identify variables whose values are set by a
228       // query to an operator and the operator's response (i.e., lot ID)
229       < var_type VAR_NAME;           // Uninitialized scalar variable
230       | var_type VAR_NAME[integer_expr]; // Uninitialized array (length integer_expr) variable
231       | var_type VAR_NAME = initial_value_element; // Initialized scalar variable
232       | var_type VAR_NAME[(integer_expr)] = [initial_value_list]; // Initialize array elements to distinct values
233       | var_type VAR_NAME[integer_expr] = initial_value_element; // Initialize all array element to same value
234       >
235
236   =====
237   STIL 1.0 {
238       ( Flow 2008; )+
239   }
240
241   =====
242   Variables ( VAR_DOMAIN ) {
243       // If "Design 2005" is specified in the STIL 1.0 statement (above),
244       // all 1450.1 syntax will also accepted.
245       var_elements_stmt*
246   }
247
248   =====
249   FunctionDefs {
250       ( FUNCTION_NAME {
251           ( Parameters {
252               ( (<In | Out | InOut>) var_elements_stmt ) *
253           ) // end Parameters
254       })* // end function_name
255   } // end FunctionDefs
256
257   =====
258   softbin_attribute =
259   <
260       Color <String>; | // Hex, RGB, or name
261       Number <Unsigned Integer>; |
262       Retest <Unsigned Integer>; |
263       Terse <String>; |
264       Verbose <String>; |
265       WafermapChar <simple_character>; // From P1450.1999 BNF
266   >
267
268   softbin_definition =
269   Bin < SOFTBIN_NAME ;
270       | SOFTBIN_NAME { (softbin_attribute)* }
271   >
272
273   BinDefs (BIN_DEF_NAME) { // soft bin definitions
274       Pass { // Increment Pass-bin if ReturnState is Pass
275           ( Color <String>; ) // Contains default color for all Pass bins, "green" if unspecified
276           (softbin_definition)* |
277           ( Axis BIN_AXIS_NAME {
278               (softbin_definition)*
279           } ) *
280       } // end Pass

```

```

281     Fail {           // Increment Fail-bin if ReturnState is Fail
282         (Color <String>); // Contains default color for all Fail bins, "red" if unspecified
283         (softbin_definition)* |
284         ( Axis BIN_AXIS_NAME {
285             (softbin_definition)*
286         })*
287     } // end Fail
288 }
289
290 // In the second form of the statement shown below, if more than one BIN_AXIS_NAME.SOFTBIN_NAME
291 // expression is present, a comma is used to separate the list
292 bin_map_stmt =
293     Map SOFTBIN_NAME integer; |
294     Map [(BIN_AXIS_NAME.SOFTBIN_NAME)+ (, )] integer;
295
296 =====
297 BinMap BIN_MAP_NAME { // soft to hard bin mapping
298     (bin_map_stmt)*
299 }
300


---


301 Bin Property access syntax
302 counter_reset_event =
303 <
304     OnLoad |
305     OnLotStart |
306     OnRetest |
307     OnSiteStart | // What's the difference between OnSiteStart
308     OnStart | // and OnStart ?
309     OnWaferStart
310 >
311
312 bin_property =
313 <
314     Color | // String
315     ContinueOnFail | // Boolean
316     counter.counter_reset_event | // Unsigned
317     Enabled | // Boolean
318     Index | // Unsigned
319     isSet.counter_reset_event | // Boolean
320     Name | // String
321     Number | // Integer
322     retest.(current|Original) | // Unsigned
323     Terse | // String
324     Verbose | // String
325     WafermapChar // Character
326 >
327
328 group = < Pass|Fail >
329
330 (BINDEFS_NAME.) group[Unsigned| SOFTBIN_NAME ].bin_property |
331 (BINDEFS_NAME.) group[Unsigned| AXIS_NAME ][Unsigned| SOFTBIN_NAME ].bin_property

```

```

332 =====
333 TestBase {
334     ( Parameters {
335         ((<In | Out | InOut>) var_elements_stmt ) *
336     }) // end Parameters
337     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
338     ( PreActions { ( < action_stmt | bypass_stmt )* } )
339     // Allows implementers to include a dummy TestExec in TestBase. The dummy TestExec can,
340     // for instance, display a warning or error that an actual Test did not include its own TestExec or
341     // FuncExec, or that an actual Flow did not include at least one FlowNode (which can be empty).
342     (TestExec; )
343     ( PostActions { ( action_stmt)* } )
344     ( PassActions { ( action_stmt)* } ) // end PassActions
345     ( FailActions { ( action_stmt)* } ) // end FailActions
346 } // end TestBase
347
348 =====
349 TestType TEST_TYPE_NAME {
350     // If not inheriting from a previously-defined TEST_TYPE_NAME (either vendor-defined or user-defined),
351     // a TestType will inherit from TestBase. The “Inherit TestBase” statement is not required, but is
352     // recommended. If no Inherit statement is present, the behavior is equivalent to “Inherit TestBase”
353     ( Inherit TestBase; | Inherit TEST_TYPE_NAME ; | Inherit FLOW_TYPE_NAME ; )
354     ( Parameters {
355         ((<In | Out | InOut>) var_elements_stmt ) *
356     }) // end Parameters
357     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
358     ( PreActions { ( < action_stmt | bypass_stmt )* } )
359     (TestExec; | FuncExec func_stmt | FuncExec { ( func_stmt )* }
360     | flownode_stmt+ ) // Allows inline instantiation of a flow in a Test. That flow can't be reused.
361     ( PostActions { ( action_stmt)* } )
362     ( PassActions { ( action_stmt)* } ) // end PassActions
363     ( FailActions { ( action_stmt)* } ) // end FailActions
364 } // end TestType
365
366 =====
367     ( Test test_instance_stmt )* // Instantiate named Test from TestType
368
369 =====
370 // A FlowType can be named or unnamed. Only one unnamed FlowType is allowed. If an unnamed
371 // FlowType is not provided by the user, then a default unnamed FlowType must be provided by the
372 // environment or toolset. The contents of this default unnamed FlowType are specified below.
373 FlowType (FLOW_TYPE_NAME) {
374     // If not inheriting from a previously-defined FLOW_TYPE_NAME (either vendor-defined or user-defined),
375     // a FlowType will inherit from TestBase. The “Inherit TestBase” statement is not required, but is
376     // recommended. If no Inherit statement is present, the behavior is equivalent to “Inherit TestBase”
377     ( Inherit TestBase; | Inherit FLOW_TYPE_NAME ; )
378     ( Parameters {
379         ((<In | Out | InOut>) var_elements_stmt ) *
380     }) // end Parameters
381     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
382     ( PreActions { ( < action_stmt | bypass_stmt )* } )
383     (flownode_stmt)*
384     ( PostActions { ( post_action_stmt)* } )
385     ( PassActions { ( action_stmt)* } ) // end PassActions
386     ( FailActions { ( action_stmt)* } ) // end FailActions
387 } // end FlowType

```

```

388
389 =====
390 ( Flow flow_instance_stmt ) * // Instantiate named Flow from FlowType
391
392 =====
393 TestProgram TEST_PROGRAM_NAME {
394     DUTType "DUT NAME STRING";
395     SocketDef "SOCKET NAME STRING";
396     // Variables are global to test program and below; local to site (a copy for each site)
397     ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN ) *
398     ( BinDefs BIN_DEF_NAME; )
399     ( BinMap BIN_MAP_NAME; )
400
401     EntryPoints {
402         ( OnException execute_stmt )
403         ( OnFinish execute_stmt )
404         ( OnLoad execute_stmt )
405         ( OnLotEnd execute_stmt )
406         ( OnLotStart execute_stmt )
407         ( OnMultiSiteDisable execute_stmt )
408         ( OnMultiSiteEnable execute_stmt )
409         ( OnPatternLoad execute_stmt )
410         ( OnPowerDown execute_stmt )
411         ( OnReset execute_stmt )
412         ( OnSiteEnd execute_stmt )
413         ( OnSiteStart execute_stmt )
414         ( OnStart execute_stmt )
415         ( OnUnload execute_stmt )
416         ( OnWaferEnd execute_stmt )
417         ( OnWaferStart execute_stmt )
418     } // End Entry Points
419 } // end TestProgram
420
421 =====

```

For Multi-Site

```

422
423
424 // PROPOSED – How do we want to specify different test programs for different sites?
425 // Is this necessary?
426 ( SiteDefs (SITE_DEF_NAME) {
427
428     // Variables are global across all sites (one copy across all sites)
429     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
430
431     ( NumSites integer_expr ; )
432
433     // One statement per site. Only one can be default. If there are more sites than statements, then all
434     // unspecified sites will use the default program.
435     // It has been stated that we'd like to accommodate a scenario in which the same test program is
436     // running on all sites, but a different flow is running on each site. Using the syntax proposed here,
437     // it is possible do this by specifying different test programs for each site. The test programs are
438     // identical EXCEPT that the Flow specified by OnStart is different.
439     ( (Default) SITE_NAME TestProgram TEST_PROGRAM_NAME;)+
440 })
441
442 SPEC_NAME.NumCategories returns the number of categories in a spec block (as an integer)
443
444 SPEC_NAME.Category[I].Name returns the category name corresponding to the Ith category in a spec block (first
445 category is index 0)
446
447 SPEC_NAME.Category[I].NumVariables returns the number of variables in a Ith category (first category is index 0)
448 in a spec block
449
450 SPEC_NAME.Category[I].Variables[J].Name returns the variable name corresponding to the Jth variable in the Ith
451 category within a spec block.
452

```

STIL.4 mandatory requirements for TestBase

```

453
454 // The STIL.4-mandated contents for TestBase are shown below. If a user provides an alternate
455 // definition, that definition MUST include all the elements shown below. The purpose of
456 // TestBase is to provide a common set of elements for all TestType and FlowType definitions.
457
458 // The user can define (and redefine) TestBase any number of times prior to first use. After first use,
459 // redefinition of TestBase is NOT allowed. First use of TestBase is defined as the first user definition of a
460 // TestType or FlowType, or first instantiation of a Flow using default FlowType.
461 // It is HIGHLY recommended that a user definition of TestBase occur prior to definition of any
462 // TestTypes, Tests, FlowTypes, Flows, or TestPrograms.
463 TestBase {
464     Parameters {
465         Out Const String TestID = ""; // Empty string – set when test or flow is instantiated.
466         Out Integer ExecResult = 0; // Return value from execution
467                                     // ExecResult == 0 -> Pass, execResult != 0 -> Fail
468                                     // 0 = Pass, non-zero = Fail
469                                     // Define Pass = 0; Define Fail = !Pass
470                                     // For the normal case of only one fail mode, ExecResult = 1 on fail.
471         In Bin FailBin = NoBin; // For component values, see section 3.2.5 of Ernie's bin doc.
472         In Integer MultiSiteSerial = False; // From dot1, p15, table 5, False = 1;
473     }
474     PreActions { } // No PreActions
475     // TestExec;
476     PostActions { } // No PostActions
477     // Semantics: PassActions or FailActions are selected by the implied arbiter:
478     // if (ExecResult == Pass) { Do PassActions } Else { Do FailActions }
479     PassActions { } // No PassActions. Pass Actions are executed if execResult == Pass
480     FailActions { // Fail Actions are executed if execResult == Fail
481         SetBinStop FailBin; // Perform binning based on the current value of FailBin, and stop
482                             // If value of FailBin is undefined, then no action (binning and stopping)
483                             // takes place.
484                             // This is the equivalent to:
485                             // If (FailBin != NoBin) { SetBin FailBin; Stop }
486     } // end FailActions
487 } // end TestBase
488
489

```

490

STIL.4 definition of default FlowNode

491 // When an unnamed (global) FlowNode definition occurs OUTSIDE of any other scope (i.e., outside
 492 // the scope of Tests or Flows, which is the only other place where it's allowed to define a FlowNode),
 493 // that FlowNode definition becomes the default FlowNode definition. If the user does NOT define a
 494 // default FlowNode, the following definition is used.
 495
 496 // The user can define (and redefine) the default FlowNode definition any number of times prior to first use.
 497 // After first use, redefinition of the default FlowNode is NOT allowed. First use of the default FlowNode
 498 // is defined as the first user definition of a FlowType, or first instantiation of a Flow using default
 499 // FlowType.
 500 // **It is HIGHLY recommended that a user definition of the default FlowNode occur prior to**
 501 // **definition of any TestTypes, Tests, FlowTypes, Flows, or TestPrograms.**

502

503

FlowNode {

504

PreActions { }

505

TestExec NullExec; // In actual use, an EXEC_OBJECT_NAME is substituted for NullExec.

506

PostActions { }

507

ExitPorts {

508

Port FAIL CurrentExec.ExecResult == Fail {

509

// Set the ExecResult of the Test or Flow containing

510

// this FlowNode. Either the hard-coded value Fail

511

// or the ExecResult of the Test or Flow which this

512

// FlowNode executed can be used. CurrentExec is an

513

// alias for EXEC_OBJECT_NAME

514

ExecResult = Fail;

515

// or

516

// ExecResult = CurrentExec.ExecResult;

517

Stop; // Stop execution of FlowNodes – return control

518

// to containing Test or Flow, and execute any

519

// PostActions and PassActions/FailActions of

520

// Test or Flow.

521

}

522

Port PASS CurrentExec.ExecResult == Pass { Next; }

523

} // end ExitPorts

524

} // end FlowNode

525

526

527

STIL.4 definition of default FlowType

528

529

// If not provided by the user, the STIL.4 environment or toolset MUST provide for an unnamed default

530

// FlowType as defined below.

531

FlowType {

532

// No Inherit keyword – so this FlowType includes all elements defined by TestBase above

533

// Parameters – use TestBase Parameters

534

// Variables – no local variables

535

// PreActions – no PreActions

536

// Flownodes – no flownodes specified in the type – will be provided at instantiation

537

// setResult CurrentExec.execResult;

538

// PostActions – no PostActions – use TestBase PostActions

539

// PassActions – no PassActions – use TestBase PassActions

540

FailActions {

541

Stop; // Stop, but do no binning here. Binning is done either at the FlowNode or the Test level.

542

}

543

}

544

STIL.4 definition of TestType NoOpType and Test NoOp

```

545
546
547 // In order to help demonstrate concepts in STIL.4, it's necessary to define a TestType, and show how a
548 // Test can be instantiated using this type. Therefore, we define an example TestType called NoOpType,
549 // and instantiate a Test called NoOp from it. This test can be used to illustrate any of the flow concepts
550 // without getting bogged down in the definition of the many actual TestTypes that might be needed on any
551 // specific tester.
552
553 TestType NoOpType {
554     // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
555     // Parameters – use TestBase Parameters
556     // Variables – no local variables
557     // PreActions – no PreActions
558     // No TestExec. No operation performed. Or – use TestExec; (with no tokens).
559     //     Test library must contain a Test named NoOp, which sets the value of
560     //     NoOp.ExecResult to the appropriate value (PASS (0) or FAIL (non-zero))
561     // PostActions – no PostActions
562     // PassActions – no PassActions
563     // FailActions – use FailActions as defined in TestBase
564 } // end TestType NoOp
565
566 Test NoOpType NoOp

```