

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; (<code>NullExec</code> 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 (<code>ID</code>), and if a different title is needed, parameters or variables can serve this purpose.
03/04/09	D0.27	jo	<ul style="list-style-type: none"> • Added semantics description for <code>func_exec</code> call. • Updated <code>bin_properties</code> per Ernie’s latest binning document. <ul style="list-style-type: none"> ○ Removed <code>counter_reset_event</code> from <code>isSet</code> property (not needed). ○ Removed <code>ContinueOnFail</code> bin property. Deferred to phase 2. ○ Added <code>IsFailBin</code> property. • Added variable attributes per discussions on 11/20/08 and 12/04/08. Added .Size operator for arrays. • Added property queries for <code>Specs/Categories</code> and <code>Bin/BinAxis</code> hierarchies. • Added IncludeOnce keyword and semantics. • Added scoping rules for variables, specs, etc.
03/11/09	D0.27	jo	<ul style="list-style-type: none"> • Fixed typo in <code>IncludeOnce</code> statement (reference to <code>MSoft #pragma once</code>, instead of <code>IncludeOnce</code>) • In <code>Spec/Category</code> hierarchy access syntax, make <code>SPEC_NAME</code> optional. • Removed <code>OnSiteStart/OnSiteEnd</code> <code>EntryPoint</code> keywords. Can’t figure out how these might differ from <code>OnStart</code> (do we need an <code>OnEnd</code> – or maybe <code>OnTestStart/OnTestEnd</code>?).
03/19/09	D0.27	jo	<ul style="list-style-type: none"> • In <code>Spec/Category</code> hierarchy access syntax, update access syntax and semantics. Will need to do the same for binning.
05/20/09	D0.27	jo	<ul style="list-style-type: none"> • Updated <code>binmap</code> syntax per latest discussions and Ernie’s binning document (see lines 325-350). • Updated access syntax (lines 360-414) for groups (<code>Pass Fail</code>), <code>Axes</code>, and <code>Bins</code> per latest discussions and Ernie’s binning document. • Changed all references to “Unsigned (integer)” to <code>Integer</code>, specifying value range limitation <code>>=0</code>

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 is the name of a formal parameter as defined for FUNC_NAME in FunctionDefs block.
142   // expr is a value or variable accessible in the scope from which func_name is called.
143   // For In parameter types, expr can be a variable or a value. The value of expr is passed to the formal
144   // parameter during the function call.
145   // For InOut parameter types, expr must be a variable accessible from the calling scope; the value of expr is
146   // assigned to the formal parameter, and if modified inside the function call, the variable used for expr is
147   // updated; the updated value is available in the calling scope.
148   // For Out parameter types, expr must be a variable. Upon returning to the calling scope, the variable used
149   // for expr contains the value assigned inside the function call.
150   (VAR_NAME = expr);*
151   (VAR_NAME = [ expr (expr)+];)* // For arrays (?)
152   } // end func_stmt
153   >
154
155 test_elements_stmt =
156   < VAR_NAME = expr;
157   | VAR_NAME = [ expr (expr)+]; // For arrays (?)
158   | PreActions { ( < action_stmt | bypass_stmt )* } )
159   | FuncExec func_stmt | FuncExec { ( func_stmt )* } | ( flownode_stmt )*
160   | PostActions { ( action_stmt )* }
161   | PassActions { ( action_stmt )* }
162   | FailActions { ( action_stmt )* }
163   >
164
165 test_instance_stmt =
166   // create a named Test from a TestType, using the default elements for the TestType
167   TEST_TYPE TEST_INSTANCE_NAME;
168

```

```

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

```

```

225     < FLOW_NAME; // execute a named Flow
226     | FLOW_TYPE; // create and execute a temporary inline Flow(named _INLINE_FLOW)
227                 // from FlowType, using the type's default element values
228     | FLOW_TYPE { (flow_elements_stmt)* } > // Create and execute an a temporary inline Flow
229                                             // (named _INLINE_FLOW) from FlowType,
230                                             // overriding the type's default element values
231
232     execute_stmt =< test_execute_stmt | flow_execute_stmt >
233
234     var_attributes =
235     < Permissions RW | RRW | RO; // Default value if not specified is RW (what do each of these mean?)
236     | Owner System | User; // Default value if not specified is User
237     | Description <string>; // Default value if not specified is empty string (“”).
238     >
239
240     // If variable is an array type, the number of elements in that array is given by VAR_NAME.Size. Arrays are
241     // indexed from 0 to<VAR_NAME.Size-1>. .Size will return the number of elements in an array, not the total
242     // memory required by each element .Size is illegal for non-array variables.
243     var_elements_stmt =
244         // Do we need an Operator attribute? Intended to identify variables whose values are set by a
245         // query to an operator and the operator's response (i.e., lot ID)
246
247         // Uninitialized scalar variable
248         < var_type VAR_NAME;
249         | var_type VAR_NAME { (var_attributes)* }
250
251         // Uninitialized array variable (length integer_expr)
252         | var_type VAR_NAME[integer_expr];
253         | var_type VAR_NAME[integer_expr] { (var_attributes)* }
254
255         // Initialized scalar variable.
256         | var_type VAR_NAME = initial_value_element;
257         | var_type VAR_NAME = initial_value_element { (var_attributes)* }
258
259         // Initialize array elements to distinct values
260         | var_type VAR_NAME[integer_expr] = [initial_value_list];
261         | var_type VAR_NAME[integer_expr] = [initial_value_list] { (var_attributes)* }
262
263         // Initialize all array element to same value
264         var_type VAR_NAME[integer_expr] = initial_value_element;
265         | var_type VAR_NAME[integer_expr] = initial_value_element { (var_attributes)* }
266     >
267
268     =====
269     STIL 1.0 {
270         ( Flow 2008; )+
271     }
272
273     =====
274     Variables ( VAR_DOMAIN ) {
275         // If “Design 2005” is specified in the STIL 1.0 statement (above),
276         // all 1450.1 syntax will also accepted.
277         var_elements_stmt*
278     }
279
280     IncludeOnce; // A statement to be used in files to be included. If present in a file, and that file is included

```

```

281 // into another file (using the Include "FILE_NAME"; statement - dot0, section 10.1) more than
282 // once, subsequent Include are ignored. Draws from Microsoft's #pragma once.
283 =====
284 FunctionsDefs {
285     ( FUNCTION_NAME {
286         ( Parameters {
287             ((<In | Out | InOut>) var_elements_stmt ) *
288         }) // end Parameters
289     })* // end function_name
290 } // end FunctionDefs
291
292 =====
293 softbin_attribute =
294 <
295     Color <String>; | // Hex, RGB, or name
296     Number <Integer>; | // Integer must be >= 0
297     Retest <Integer>; | // Integer must be >= 0
298     Terse <String>; |
299     Verbose <String>; |
300     WafermapChar <simple_character>; // From P1450.1999 BNF
301 >
302
303 softbin_definition =
304 Bin < SOFTBIN_NAME ;
305     | SOFTBIN_NAME { (softbin_attribute)* }
306 >
307
308 BinDefs (BIN_DEF_NAME) { // soft bin definitions
309     Pass { // Increment Pass-bin if ReturnState is Pass
310         (Color <String>;) // Contains default color for all Pass bins, "green" if unspecified
311         (softbin_definition)* |
312         ( Axis BIN_AXIS_NAME {
313             (softbin_definition)*
314         } ) *
315     } // end Pass
316     Fail { // Increment Fail-bin if ReturnState is Fail
317         (Color <String>;) // Contains default color for all Fail bins, "red" if unspecified
318         (softbin_definition)* |
319         ( Axis BIN_AXIS_NAME {
320             (softbin_definition)*
321         } ) *
322     } // end Fail
323 }
324
325 // In the second form of the statement shown below, if more than one softbin expression
326 // (Pass | Fail).Axes[BIN_AXIS_NAME].Bins[SOFTBIN_NAME]
327 // is present, white space is used to separate the list of softbin expressions
328 // If SOFTBIN_NAME names are unique across bin groups (Pass or Fail), there is no need to specify the Pass or
329 // Fail group
330 bin_map_stmt =
331     Map ((Pass | Fail).Bins[SOFTBIN_NAME | integer ] integer; |
332     Map (Pass | Fail). Axes[BIN_AXIS_NAME | integer ].Bins[SOFTBIN_NAME | integer ]+ ] integer;
333
334 // When indexing Bin or Axis arrays, either the softbin or bin axis name can be used, or an integer >= 0 can be used.
335
336

```

```

337 // Binmap entries showing softbin to hardbin mapping when bin axes are used follow below:
338
339 Map Pass.Axes[CacheSize].Bins["8Mb"] Pass.Axes[ClockSpeed].Bins["3.00GHz"] 1;
340 Map Pass.Bins["8Mb"] 1;
341
342 // Binmap entries showing softbin to hardbin mapping when no bin axes are used (i.e., an anonymous axis) follow:
343 Map Fail.Bins[ContactOpens] 5;
344 Map Fail.Axes[0].Bins[ContactOpens] 5;
345
346 JO NOTE: The use of index 0 as an anonymous array axis (where bin axes are indexed starting from 1),
347 as shown on line 344, is in conflict with comments at lines 240-242 regarding 0-based variable array
348 access, and at lines 415-440 regarding 0-based spec/category array access. We either need to switch to
349 1-based array access, or use an alternate form to indicate anonymous bin axis access (or prohibit it
350 altogether).
351
352
353 =====
354     BinMap BIN_MAP_NAME { // soft to hard bin mapping
355         (bin_map_stmt)*
356     }
357
358
359 =====
360 // Group Property Access Syntax
361 group_property =
362 <
363     Color |// String (hexadecimal RGB or name, e.g., red)
364     isAnyBinSet |// Boolean
365     Size |// Integer – number of axes. If no axes, returns 0. If one or more axes, returns value >=0
366 >
367
368 (BINDEFS_NAME.)group.group_property
369
370 // Axis Property Access Syntax
371 axis_property =
372 <
373     highestSetBin |// Integer – value returned will be >=0. (What value is returned if no bin is set ?)
374     lowestSetBin |// Integer – value returned will be >=0. (What value is returned if no bin is set ?)
375     Size |// Unsigned
376     Name |// String
377 >
378
379 group = < Pass|Fail >
380 (BINDEFS_NAME.)group.Axes[Integer|AXIS_NAME].axis_property
381
382 // Bin Property access syntax
383 counter_reset_event =
384 <
385     OnLoad |
386     OnLotStart |
387     OnRetest |
388     OnSiteStart | // What's the difference between OnSiteStart
389     OnStart | // and OnStart ?
390     OnWaferStart
391 >
392

```

```

393  bin_property =
394  <
395      Color |                               // String (hexadecimal RGB or name, e.g., red)
396      counter.counter_reset_event | // Unsigned
397      Enabled |                             // Boolean
398      Index |                               // Unsigned
399      isFailBin |                           // Boolean
400      isSet |                               // Boolean
401      Name |                               // String
402      Number |                             // Integer
403      retest.(current|Original) | // Unsigned
404      Terse |                              // String
405      Verbose |                            // String
406      WafermapChar |                       // Character
407  >
408
409  group = < Pass|Fail >
410
411  Binefs access syntax
412  (BINDEFS_NAME.) group.Bins[Integer | SOFTBIN_NAME ],bin_property |
413  (BINDEFS_NAME.) group.Axes [Integer | AXIS_NAME ][Integer | SOFTBIN_NAME ],bin_property
414  Pass.Axes[CacheSize].Bins["8Mb"] Pass.Axes[ClockSpeed].Bins["3.00GHz"] 1;

```

```

415 // Spec and Category access syntax. If spec blocks are specified using Specs/Variables/Catetories notation rather
416 // than Spec/Categories/Variables notation, same notation still applies (since in STIL dot0, spec variables are global
417 // in scope with the combination of category name + variable name being unique).
418 SPEC_NAME.Size returns the number of categories in a spec block (as an integer)
419
420 (SPEC_NAME.)Category[I] returns the Ith category in a spec block (first category is index 0), which can then be
421 // further indexed, as shown below.
422
423 (SPEC_NAME.)Category[I].Name returns a string containing the category name corresponding to the Ith category in
424 // a spec block (first category is index 0)
425
426 (SPEC_NAME.)Category[I].Size returns the number of variables in the Ith category (first category is index 0) in a
427 // spec block
428
429 (SPEC_NAME.)Category[I].Variables[J] returns the Jth variable in the Ith category within a spec block, suitable for
430 // further indexing, as shown below. The value of the variable is determined by the currently-active selector, unless
431 // that selection is overridden explicitly, as shown below by the .[Min | Typ | Max | Meas ] syntax.
432
433 (SPEC_NAME.)Category[I].Variables[J].Name returns a string containing the variable name of the Jth variable in
434 // the Ith category within a spec block.
435
436 (SPEC_NAME.)Category[I].Variables[J].[Min | Typ | Max | Meas] returns the Jth variable in the Ith category within
437 // a spec block. The value of the variable is determined by the specification of .[Min | Typ | Max | Meas].
438
439 // In the above syntax, the explicit category names or variable names can be substituted for Category[I] and
440 // Variable[J]. (example to follow).
441
442 =====
443 TestBase {
444     ( Parameters {
445         ( (<In | Out | InOut>) var_elements_stmt ) *
446     } ) // end Parameters
447     ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN ) *
448     ( PreActions { ( < action_stmt | bypass_stmt ) * } )
449     // Allows implementers to include a dummy TestExec in TestBase. The dummy TestExec can,
450     // for instance, display a warning or error that an actual Test did not include its own TestExec or
451     // FuncExec, or that an actual Flow did not include at least one FlowNode (which can be empty).
452     ( TestExec; )
453     ( PostActions { ( action_stmt ) * } )
454     ( PassActions { ( action_stmt ) * } ) // end PassActions
455     ( FailActions { ( action_stmt ) * } ) // end FailActions
456 } // end TestBase
457
458 =====
459 TestType TEST_TYPE_NAME {
460     // If not inheriting from a previously-defined TEST_TYPE_NAME (either vendor-defined or user-defined),
461     // a TestType will inherit from TestBase. The "Inherit TestBase" statement is not required, but is
462     // recommended. If no Inherit statement is present, the behavior is equivalent to "Inherit TestBase"
463     ( Inherit TestBase; | Inherit TEST_TYPE_NAME ; | Inherit FLOW_TYPE_NAME ; )
464     ( Parameters {
465         ( (<In | Out | InOut>) var_elements_stmt ) *
466     } ) // end Parameters
467     ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN ) *
468     ( PreActions { ( < action_stmt | bypass_stmt ) * } )
469     ( TestExec; | FuncExec func_stmt | FuncExec { ( func_stmt ) * }
470     | flownode_stmt+ ) // Allows inline instantiation of a flow in a Test. That flow can't be reused.

```

```

471     ( PostActions { ( action_stmt)* } )
472     ( PassActions { ( action_stmt)* } ) // end PassActions
473     ( FailActions { ( action_stmt)* } ) // end FailActions
474 } // end TestType
475
476 =====
477     ( Test test_instance_stmt )* // Instantiate named Test from TestType
478
479
480
481
482
483
484 =====
485 // A FlowType can be named or unnamed. Only one unnamed FlowType is allowed. If an unnamed
486 // FlowType is not provided by the user, then a default unnamed FlowType must be provided by the
487 // environment or toolset. The contents of this default unnamed FlowType are specified below.
488 FlowType (FLOW_TYPE_NAME) {
489     // If not inheriting from a previously-defined FLOW_TYPE_NAME (either vendor-defined or user-defined),
490     // a FlowType will inherit from TestBase. The “Inherit TestBase” statement is not required, but is
491     // recommended. If no Inherit statement is present, the behavior is equivalent to “Inherit TestBase”
492     ( Inherit TestBase; | Inherit FLOW_TYPE_NAME ; )
493     ( Parameters {
494         ( (<In | Out | InOut>) var_elements_stmt )*
495     } ) // end Parameters
496     ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN )*
497     ( PreActions { ( <action_stmt | bypass_stmt )* } )
498     ( flownode_stmt )*
499     ( PostActions { ( post_action_stmt)* } )
500     ( PassActions { ( action_stmt)* } ) // end PassActions
501     ( FailActions { ( action_stmt)* } ) // end FailActions
502 } // end FlowType
503
504 =====
505     ( Flow flow_instance_stmt )* // Instantiate named Flow from FlowType
506
507 =====
508 TestProgram TEST_PROGRAM_NAME {
509     SocketDef "SOCKET NAME STRING";
510     // Variables are global to test program and below; local to site (a copy for each site)
511     ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN )*
512     ( BinDefs BIN_DEF_NAME; )
513     ( BinMap BIN_MAP_NAME; )
514
515     EntryPoints {
516         ( OnException execute_stmt )
517         ( OnFinish execute_stmt )
518         ( OnLoad execute_stmt )
519         ( OnLotEnd execute_stmt )
520         ( OnLotStart execute_stmt )
521         ( OnMultiSiteDisable execute_stmt )
522         ( OnMultiSiteEnable execute_stmt )
523         ( OnPatternLoad execute_stmt )
524         ( OnPowerDown execute_stmt )
525         ( OnReset execute_stmt )
526         ( OnStart execute_stmt )
527         ( OnUnload execute_stmt )

```

```
527         ( OnWaferEnd execute_stmt )
528         ( OnWaferStart execute_stmt )
529     } // End Entry Points
530 } // end TestProgram
531
532 =====
533
534
```

535 For Multi-Site

```

536
537 // PROPOSED – How do we want to specify different test programs for different sites?
538 // Is this necessary?
539 ( SiteDefs (SITE_DEF_NAME) {
540
541     // Variables are global across all sites (one copy across all sites)
542     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
543
544     ( NumSites integer_expr ; )
545
546     // One statement per site. Only one can be default. If there are more sites than statements, then all
547     // unspecified sites will use the default program.
548     // It has been stated that we'd like to accommodate a scenario in which the same test program is
549     // running on all sites, but a different flow is running on each site. Using the syntax proposed here,
550     // it is possible do this by specifying different test programs for each site. The test programs are
551     // identical EXCEPT that the Flow specified by OnStart is different.
552     ( (Default) SITE_NAME TestProgram TEST_PROGRAM_NAME;)+
553 })
554

```

555 **Variable scoping rules.**

556 Within a test or flow, local scope consists of local variables and parameters. Each name must be unique in that
557 context.

558 Variables, constants, and parameters (including spec variables) declared in the local scope will hide any variables,
559 constants, or parameters of the same name which are declared in the global scope. Global scope includes specs
560 declared in Spec blocks,
561 and variables declared in an unnamed variables block.

562
563 To access the variables in the upper level scope, the complete variable or spec access syntax can be used.

```

564 STIL.4 mandatory requirements for TestBase
565 // The STIL.4-mandated contents for TestBase are shown below. If a user provides an alternate
566 // definition, that definition MUST include all the elements shown below. The purpose of
567 // TestBase is to provide a common set of elements for all TestType and FlowType definitions.
568
569 // The user can define (and redefine) TestBase any number of times prior to first use. After first use,
570 // redefinition of TestBase is NOT allowed. First use of TestBase is defined as the first user definition of a
571 // TestType or FlowType, or first instantiation of a Flow using default FlowType.
572 // It is HIGHLY recommended that a user definition of TestBase occur prior to definition of any
573 // TestTypes, Tests, FlowTypes, Flows, or TestPrograms.
574 TestBase {
575     Parameters {
576         Out Const String TestID = ""; // Empty string – set when test or flow is instantiated.
577         Out Integer ExecResult = 0; // Return value from execution
578                                     // ExecResult == 0 -> Pass, execResult != 0 -> Fail
579                                     // 0 = Pass, non-zero = Fail
580                                     // Define Pass = 0; Define Fail = !Pass
581                                     // For the normal case of only one fail mode, ExecResult = 1 on fail.
582         In Bin FailBin = NoBin; // For component values, see section 3.2.5 of Ernie’s bin doc.
583         In Integer MultiSiteSerial = False; // From dot1, p15, table 5, False = 1;
584     }
585     PreActions { } // No PreActions
586     // TestExec;
587     PostActions { } // No PostActions
588     // Semantics: PassActions or FailActions are selected by the implied arbiter:
589     // if (ExecResult == Pass) { Do PassActions } Else { Do FailActions }
590     PassActions { } // No PassActions. Pass Actions are executed if execResult == Pass
591     FailActions { // Fail Actions are executed if execResult == Fail
592         SetBinStop FailBin; // Perform binning based on the current value of FailBin, and stop
593                             // If value of FailBin is undefined, then no action (binning and stopping)
594                             // takes place.
595                             // This is the equivalent to:
596                             // If (FailBin != NoBin) { SetBin FailBin; Stop }
597     } // end FailActions
598 } // end TestBase
599
600

```

STIL.4 definition of default FlowNode

```

601
602 // When an unnamed (global) FlowNode definition occurs OUTSIDE of any other scope (i.e., outside
603 // the scope of Tests or Flows, which is the only other place where it's allowed to define a FlowNode),
604 // that FlowNode definition becomes the default FlowNode definition. If the user does NOT define a
605 // default FlowNode, the following definition is used.
606
607 // The user can define (and redefine) the default FlowNode definition any number of times prior to first use.
608 // After first use, redefinition of the default FlowNode is NOT allowed. First use of the default FlowNode
609 // is defined as the first user definition of a FlowType, or first instantiation of a Flow using default
610 // FlowType.
611 // It is HIGHLY recommended that a user definition of the default FlowNode occur prior to
612 // definition of any TestTypes, Tests, FlowTypes, Flows, or TestPrograms.
613
614 FlowNode {
615     PreActions { }
616     TestExec NullExec; // In actual use, an EXEC_OBJECT_NAME is substituted for NullExec.
617     PostActions {}
618     ExitPorts {
619         Port FAIL CurrentExec.ExecResult == Fail {
620             // Set the ExecResult of the Test or Flow containing
621             // this FlowNode. Either the hard-coded value Fail
622             // or the ExecResult of the Test or Flow which this
623             // FlowNode executed can be used. CurrentExec is an
624             // alias for EXEC_OBJECT_NAME
625             ExecResult = Fail;
626             // or
627             // ExecResult = CurrentExec.ExecResult;
628             Stop; // Stop execution of FlowNodes – return control
629                 // to containing Test or Flow, and execute any
630                 // PostActions and PassActions/FailActions of
631                 // Test or Flow.
632         }
633         Port PASS CurrentExec.ExecResult == Pass { Next; }
634     } // end ExitPorts
635 } // end FlowNode
636
637

```

STIL.4 definition of default FlowType

```

638
639
640 // If not provided by the user, the STIL.4 environment or toolset MUST provide for an unnamed default
641 // FlowType as defined below.
642 FlowType {
643     // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
644     // Parameters – use TestBase Parameters
645     // Variables – no local variables
646     // PreActions – no PreActions
647     // Flownodes – no flownodes specified in the type – will be provided at instantiation
648     // setResult CurrentExec.execResult;
649     // PostActions – no PostActions – use TestBase PostActions
650     // PassActions – no PassActions – use TestBase PassActions
651     FailActions {
652         Stop; // Stop, but do no binning here. Binning is done either at the FlowNode or the Test level.
653     }
654 }
655

```

STIL.4 definition of TestType NoOpType and Test NoOp

```

656
657
658 // In order to help demonstrate concepts in STIL.4, it's necessary to define a TestType, and show how a
659 // Test can be instantiated using this type. Therefore, we define an example TestType called NoOpType,
660 // and instantiate a Test called NoOp from it. This test can be used to illustrate any of the flow concepts
661 // without getting bogged down in the definition of the many actual TestTypes that might be needed on any
662 // specific tester.
663
664 TestType NoOpType {
665     // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
666     // Parameters – use TestBase Parameters
667     // Variables – no local variables
668     // PreActions – no PreActions
669     // No TestExec. No operation performed. Or – use TestExec; (with no tokens).
670     //     Test library must contain a Test named NoOp, which sets the value of
671     //     NoOp.ExecResult to the appropriate value (PASS (0) or FAIL (non-zero))
672     // PostActions – no PostActions
673     // PassActions – no PassActions
674     // FailActions – use FailActions as defined in TestBase
675 } // end TestType NoOp
676
677 Test NoOpType NoOp

```