

1

Revision History

2

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

3 Syntax summary for 1450.4

```

4   stil_block_type =
5       < Category
6         | DCLevels
7         | DCSequence
8         | DCSets
9         | Environment
10        | Flow
11        | MacroDefs
12        | Pattern
13        | PatternBurst
14        | PatternExec
15        | Procedures
16        | ScanStructures
17        | Selector
18        | SignalGroups
19        | Signals
20        | Test
21        | Timing >
22
23   real_var_type =
24       < Capacitance // F (Farads)
25         | Compound // combinations of types
26         | Current // A (Amperes)
27         | Double
28         | Frequency // Hz (Herz)
29         | Gain // db (Decibels)
30         | Inductance // H (Henries)
31         | Length // m (Meters)
32         | Power // W (Watts)
33         | Real
34         | Resistance // Ohm (Ohms)
35         | Temperature // Cel (Degrees Celsius)
36         | Time // s (Seconds)
37         | Voltage // V (Volts) >
38
39   var_type =
40       // Boolean values: True/False or Pass/Fail
41       (Const) < Boolean | Integer | SignalRef | SignalVariable | String | real_var_type | stil_block_type >
42
43   initial_value_elements =
44       < integer_expr // allow if var of type integer_expr
45         | sig_ref_expr // allow if var of type sig_ref_expr
46         | < True | False > // allow if var of type Boolean – from dot1, 0 = TRUE, 1 = FALSE.
47         | string // allow if var of type String
48         | wfc_string // allow if var of type SignalVariable
49         | real_expr // allow if var of type real_var_type
50         | BLOCK_NAME // allow if var of type stil_block_type - assign from predefined block
51         | NullBlock // allow if var of type stil_block_type – set param to a NullBlock value
52         | { test_elements_stmt+ } // allow if var of type TestType
53         | { flow_elements_stmt+ } // allow if var of type FlowType
54         >
55
56

```

```

57
58 // In the second form of the statement shown below (used for array initialization), if more than one
59 // initial_val_elements statement is expression is present, a comma is used to separate the list
60 initial_value_stmt = InitialValue initial_val_elements | InitialValue initial_val_elements+ (,)
61
62 // Use either the explicit softbin name, or use a softbin variable name (variable of type Bin).
63 bin_expr = (BIN_AXIS.)SOFTBIN_NAME | BIN_VAR_NAME
64
65 action =
66     < VAR_NAME = expr; // Scalar variables
67     | VAR_NAME[INDEX] = expr; // Array variables
68     // Only one element of each bin axis can be active at any one time. For 3 separate axes, you can
69     // have 3 separate softbins (one from each axis) set – the binmap handles the various intersections
70     // when the part is binned.
71     | SetBin bin_expr;
72
73     // Return to the caller, “bubbling up” through the levels of FlowNodes, Flows and Tests, until the
74     // initiating EntryPoint is reached. At each level, execute the PostActions and PassActions or
75     // FailActions for Tests and Flows, but SKIP execution of FlowNodes (including any FlowNode
76     // PostActions or ExitPort actions).
77     | Stop; // terminate the initiating On* condition.
78
79     // SetBinStop is equivalent, semantically, to:
80     // If (<bin_name> != NoBin) { SetBin <bin_name>; Stop;} Else { // No Action }
81     | SetBinStop bin_expr;
82
83     // Exit – Immediately stop execution of current block (test, flow, or flow node), and jump directly to the
84     // initiating EntryPoint (On* condition) for end-of-test processing. The pass/fail result returned to the
85     // EntryPoint initiator (which would normally be the ExecResult of the Test or Flow specified by the
86     // EntryPoint), is specified by the integer_expr token of the “Exit <integer_expr>;” statement. This
87     // statement is generally intended to be used for exceptions, such as hardware failures, which require
88     // immediate termination of test execution.
89     | Exit integer_expr; // terminate the initiating On* condition.
90
91 action_stmt =
92     < If boolean_expr | Else If boolean_expr | Else { // Usual rules for If/Else If/Else apply
93         (action)*
94     }
95     | (action)* >
96
97 bypass_actions =
98     <
99     // For FlowNodes, Flows, and Tests
100    // Skip only Test execution or FlowNode sequence execution, resume at entry to PostActions Block
101    // Normal processing continues from there. Execute PostActions, and PassActions/FailActions (as
102    // determined by execResult of Test or Flow), or ExitPort selection Actions (as determined by flownode
103    // exit port selector). Selection of Pass/Fail path (for Tests and Flows) is controlled by execResult (which
104    // can be forced to a desired state prior to Bypass statement). Selection of ExitPort path (for flownodes)
105    // is also controlled by boolean expressions – typically, the return status of the test executed by the
106    // FlowNode. This return state (or value of any other boolean expression) can be forced to a desired state
107    // prior to the Bypass statement.
108    Bypass;
109
110    // For Flows and Tests only.
111    // Skip Test and PostActions, resume execution at entry to PassActions/FailActions.
112    // If Pass or Fail specified, follow that path. Otherwise, VAR_NAME is a string variable which specifies

```

```

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

```

```

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

```

```

225     | FLOW_TYPE; // create and execute a temporary inline Flow(named _INLINE_FLOW)
226                 // from FlowType, using the type's default element values
227     | FLOW_TYPE { (flow_elements_stmt)* } > // Create and execute an a temporary inline Flow
228                 // (named _INLINE_FLOW) from FlowType,
229                 // overriding the type's default element values
230
231     execute_stmt =< test_execute_stmt | flow_execute_stmt >
232
233     var_elements_stmt =
234         // Do we need an Operator attribute? Intended to identify variables whose values are set by a
235         // query to an operator and the operator's response (i.e., lot ID)
236         < var_type VAR_NAME ;
237         | var_type VAR_NAME {
238             ( Length integer; ) // array of var_type HOW TO INDEX????
239             ( initial_value_stmt )
240         }
241
242     =====
243     STIL 1.0 {
244         ( Flow 2008; )+
245     }
246
247     =====
248     Variables ( VAR_DOMAIN ) { // extensions to 1450.1
249         var_elements_stmt*
250     }
251
252     =====
253     FunctionDefs {
254         ( FUNCTION_NAME {
255             ( Parameters {
256                 ( var_type VAR_NAME (< In | Out | InOut > ; )*)
257                 ( var_type VAR_NAME (< In | Out | InOut > ) { initial_value_stmt } )*)
258             } ) // end Parameters
259         })* // end function_name
260     } // end FunctionDefs
261
262     =====
263     softbin_attribute =
264     <
265         Color <String>; | // Hex, RGB, or name
266         Number <Unsigned Integer>; |
267         Retest <Unsigned Integer>; |
268         Terse <String>; |
269         Verbose <String>; |
270         WafermapChar <simple_character>; // From P1450.1999 BNF
271     >
272
273     softbin_definition =
274     Bin < SOFTBIN_NAME ;
275         | SOFTBIN_NAME { (softbin_attribute)* }
276     >
277
278     BinDefs ( BIN_DEF_NAME ) { // soft bin definitions
279         Pass { // Increment Pass-bin if ReturnState is Pass
280             ( Color <String>; ) // Contains default color for all Pass bins, "green" if unspecified

```

```

281         (softbin_definition)* |
282         ( Axis BIN_AXIS_NAME {
283             (softbin_definition)*
284         } )*
285     } // end Pass
286 Fail { // Increment Fail-bin if ReturnState is Fail
287     (Color <String>); // Contains default color for all Fail bins, "red" if unspecified
288     (softbin_definition)* |
289     ( Axis BIN_AXIS_NAME {
290         (softbin_definition)*
291     } )*
292 } // end Fail
293 }
294
295 // In the second form of the statement shown below, if more than one BIN_AXIS_NAME.SOFTBIN_NAME
296 // expression is present, a comma is used to separate the list
297 bin_map_stmt =
298     Map SOFTBIN_NAME integer; |
299     Map [(BIN_AXIS_NAME.SOFTBIN_NAME)+ (, )] integer;
300
301 =====
302 BinMap BIN_MAP_NAME { // soft to hard bin mapping
303     (bin_map_stmt)*
304 }
305


---


306 Bin Property access syntax
307 counter_reset_event =
308 <
309     OnLoad |
310     OnLotStart |
311     OnRetest |
312     OnSiteStart | // What's the difference between OnSiteStart
313     OnStart | // and OnStart ?
314     OnWaferStart
315 >
316
317 bin_property =
318 <
319     Color | // String
320     ContinueOnFail | // Boolean
321     counter.counter_reset_event | // Unsigned
322     Enabled | // Boolean
323     Index | // Unsigned
324     isSet.counter_reset_event | // Boolean
325     Name | // String
326     Number | // Integer
327     retest.(current|Original) | // Unsigned
328     Terse | // String
329     Verbose | // String
330     WafermapChar // Character
331 >
332
333
334 group = < Pass|Fail >
335
336 (BINDEFS_NAME.) group[Unsigned|SOFTBIN_NAME ].bin_property |

```

337 (BINDEFS_NAME.) *group*[*Unsigned* AXIS_NAME][*Unsigned* SOFTBIN_NAME],*bin_property*
338
339
340
341

```

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

```

```

398     ( PostActions { ( post_action_stmt)* } )
399     ( PassActions { ( action_stmt)* } ) // end PassActions
400     ( FailActions { ( action_stmt)* } ) // end FailActions
401 } // end FlowType
402
403 =====
404 ( Flow flow_instance_stmt )* // Instantiate named Flow from FlowType
405
406 =====
407 TestProgram TEST_PROGRAM_NAME {
408     DUTType "DUT NAME STRING";
409     SocketDef "SOCKET NAME STRING";
410     // Variables are global to test program and below; local to site (a copy for each site)
411     ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN )*
412     ( BinDefs BIN_DEF_NAME; )
413     ( BinMap BIN_MAP_NAME; )
414
415     EntryPoints {
416         ( OnException execute_stmt )
417         ( OnFinish execute_stmt )
418         ( OnLoad execute_stmt )
419         ( OnLotEnd execute_stmt )
420         ( OnLotStart execute_stmt )
421         ( OnMultiSiteDisable execute_stmt )
422         ( OnMultiSiteEnable execute_stmt )
423         ( OnPatternLoad execute_stmt )
424         ( OnPowerDown execute_stmt )
425         ( OnReset execute_stmt )
426         ( OnSiteEnd execute_stmt )
427         ( OnSiteStart execute_stmt )
428         ( OnStart execute_stmt )
429         ( OnUnload execute_stmt )
430         ( OnWaferEnd execute_stmt )
431         ( OnWaferStart execute_stmt )
432     } // End Entry Points
433 } // end TestProgram
434
435 =====

```

For Multi-Site

```

436
437
438 // PROPOSED – How do we want to specify different test programs for different sites?
439 // Is this necessary?
440 ( SiteDefs (SITE_DEF_NAME) {
441
442     // Variables are global across all sites (one copy across all sites)
443     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
444
445     ( NumSites integer_expr ; )
446
447     // One statement per site. Only one can be default. If there are more sites than statements, then all
448     // unspecified sites will use the default program.
449     // It has been stated that we'd like to accommodate a scenario in which the same test program is
450     // running on all sites, but a different flow is running on each site. Using the syntax proposed here,
451     // it is possible do this by specifying different test programs for each site. The test programs are
452     // identical EXCEPT that the Flow specified by OnStart is different.
453     ( (Default) SITE_NAME TestProgram TEST_PROGRAM_NAME;)+
454 })
455
456 SPEC_NAME.NumCategories returns the number of categories in a spec block (as an integer)
457
458 SPEC_NAME.Category[I].Name returns the category name corresponding to the Ith category in a spec block
459 (first category is index 0)
460
461 SPEC_NAME.Category[I].NumVariables returns the number of variables in a Ith category (first category is
462 index 0) in a spec block
463
464 SPEC_NAME.Category[I].Variables[J].Name returns the variable name corresponding to the Jth variable in
465 the Ith category within a spec block.
466

```

```

467
468 STIL.4 mandatory requirements for TestBase
469 // The STIL.4-mandated contents for TestBase are shown below. If a user provides an alternate
470 // definition, that definition MUST include all the elements shown below. The purpose of
471 // TestBase is to provide a common set of elements for all TestType and FlowType definitions.
472 TestBase {
473     Parameters {
474         Out Const String TestID { InitialValue ""; } // Empty string – set when test or flow is instantiated.
475         Out Integer ExecResult { InitialValue 0; } // Return value from execution
476         // ExecResult == 0 -> Pass, execResult != 0 -> Fail
477         // 0 = Pass, non-zero = Fail
478         // Define Pass = 0; Define Fail = !Pass
479         // For the normal case of only one fail mode, ExecResult = 1 on fail.
480         In Bin FailBin { InitialValue NoBin; } // For component values, see section 3.2.5 of Ernie's bin doc.
481     }
482     PreActions { } // No PreActions
483     // TestExec;
484     PostActions { } // No PostActions
485     // Semantics: PassActions or FailActions are selected by the implied arbiter:
486     // if (ExecResult == Pass) { Do PassActions } Else { Do FailActions }
487     PassActions { } // No PassActions. Pass Actions are executed if execResult == Pass
488     FailActions { // Fail Actions are executed if execResult == Fail
489         SetBinStop FailBin; // Perform binning based on the current value of FailBin, and stop
490         // If value of FailBin is undefined, then no action (binning and stopping)
491         // takes place.
492         // This is the equivalent to:
493         // If (FailBin != NoBin) { SetBin FailBin; Stop }
494     } // end FailActions
495 } // end TestBase
496

```

497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551

STIL.4 definition of default FlowNode

```
// When an unnamed (global) FlowNode definition occurs OUTSIDE of any other scope (i.e., outside
// the scope of Tests or Flows, which is the only other place where it's allowed to define a FlowNode),
// that FlowNode definition becomes the default FlowNode definition. If the user wishes to allow using a
// series of TestExec statements in place of FlowNodes, then the user MUST provide a default FlowNode
// definition. If the user does NOT provide a default FlowNode definition, then the user CANNOT
// substitute a TestExec statement in place of FlowNodes, and must provide the complete FlowNode syntax.
FlowNode {
  PreActions { }
  TestExec NullExec; // In actual use, an EXEC_OBJECT_NAME is substituted for NullExec.
  PostActions {}
  ExitPorts {
    Port FAIL CurrentExec.ExecResult == Fail {
      // Set the ExecResult of the Test or Flow containing
      // this FlowNode. Either the hard-coded value Fail
      // or the ExecResult of the Test or Flow which this
      // FlowNode executed can be used. CurrentExec is an
      // alias for EXEC_OBJECT_NAME
      ExecResult = Fail;
      // or
      // ExecResult = CurrentExec.ExecResult;
      Stop; // Stop execution of FlowNodes – return control
            // to containing Test or Flow, and execute any
            // PostActions and PassActions/FailActions of
            // Test or Flow.
    }
    Port PASS CurrentExec.ExecResult == Pass { Next; }
  } // end ExitPorts
} // end FlowNode
```

STIL.4 definition of default FlowType

```
// If not provided by the user, the STIL.4 environment or toolset MUST provide for an unnamed default
// FlowType as defined below.
FlowType {
  // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
  // Parameters – use TestBase Parameters
  // Variables – no local variables
  // PreActions – no PreActions
  // Flownodes – no flownodes specified in the type – will be provided at instantiation
  // setResult CurrentExec.execResult;
  // PostActions – no PostActions – use TestBase PostActions
  // PassActions – no PassActions – use TestBase PassActions
  FailActions {
    Stop; // Stop, but do no binning here. Binning is done either at the FlowNode or the Test level.
  }
}
```

552

553

554

STIL.4 definition of TestType NoOpType and Test NoOp

555

556 // In order to help demonstrate concepts in STIL.4, it's necessary to define a TestType, and show how a

557 // Test can be instantiated using this type. Therefore, we define an example TestType called NoOpType,

558 // and instantiate a Test called NoOp from it. This test can be used to illustrate any of the flow concepts

559 // without getting bogged down in the definition of the many actual TestTypes that might be needed on any

560 // specific tester.

561

562 **TestType** NoOpType {

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

564 // Parameters – use TestBase Parameters

565 // Variables – no local variables

566 // PreActions – no PreActions

567 // No TestExec. No operation performed. Or – use TestExec; (with no tokens).

568 // Test library must contain a Test named NoOp, which sets the value of

569 // NoOp.ExecResult to the appropriate value (PASS (0) or FAIL (non-zero))

570 // PostActions – no PostActions

571 // PassActions – no PassActions

572 // FailActions – use FailActions as defined in TestBase

573 } // end TestType NoOp

574

575 **Test** NoOpType NoOp