

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 overrried 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.

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); // For last FlowNode in a sequence, "Next;" == "Return;"
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     } // end FlowNode
189
190 flow_elements_stmt =
191     < VAR_NAME = expr;
192     | VAR_NAME = [ expr (expr)+]; // For arrays (?)
193     | PreActions { ( < action_stmt | bypass_stmt )* } )
194     | Title STRING ;
195     | ( flownode_stmt )*
196     | PostActions { ( action_stmt )* } )
197     | PassActions { ( action_stmt )* } // end PassActions
198     | FailActions { ( action_stmt )* } // end FailActions
199     >
200
201 flow_instance_stmt =
202     // create a named Flow from a FlowType, using the default elements for the FlowType
203     < (FLOW_TYPE) FLOW_INSTANCE_NAME ; // if FLOW_TYPE is not specified, then the
204     // STIL.4 default FLOW_TYPE is used
205
206     // create a named Flow from a FlowType, overriding some or all of the default elements of the FlowType.
207     // Any element specified in the instantiation (i.e., PreActions, flownodes, PassActions) completely
208     // replaces that element as specified in the type definition.
209     | (FLOW_TYPE) FLOW_INSTANCE_NAME { ( flow_elements_stmt )* } > // if FLOW_TYPE is not specified, then the
210     // STIL.4 default FlowType (the unnamed
211     // FlowType) is used.
212
213 test_execute_stmt =
214     < TEST_NAME ; // execute a named Test
215     | TEST_TYPE; // create and execute a temporary inline Test (named _INLINE_TEST)
216     // from TestType, using the type's default element value.
217
218     | TEST_TYPE { ( test_elements_stmt )* } > // Create and execute a temporary inline Test
219     // (named _INLINE_TEST) from TestType,
220     // overriding the type's default element values
221
222 flow_execute_stmt =
223     < FLOW_NAME; // execute a named Flow
224     | FLOW_TYPE; // create and execute a temporary inline Flow(named _INLINE_FLOW)

```

```

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

```

```

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


---


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

```

```

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

```

```

392     ( PostActions { ( post_action_stmt)* } )
393     ( PassActions { ( action_stmt)* } ) // end PassActions
394     ( FailActions { ( action_stmt)* } ) // end FailActions
395 } // end FlowType
396
397 =====
398 ( Flow flow_instance_stmt )* // Instantiate named Flow from FlowType
399
400 =====
401 TestProgram TEST_PROGRAM_NAME {
402     DUTType "DUT NAME STRING";
403     SocketDef "SOCKET NAME STRING";
404     // Variables are global to test program and below; local to site (a copy for each site)
405     ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN )*
406     ( BinDefs BIN_DEF_NAME; )
407     ( BinMap BIN_MAP_NAME; )
408
409     EntryPoints {
410         ( OnException execute_stmt )
411         ( OnFinish execute_stmt )
412         ( OnLoad execute_stmt )
413         ( OnLotEnd execute_stmt )
414         ( OnLotStart execute_stmt )
415         ( OnMultiSiteDisable execute_stmt )
416         ( OnMultiSiteEnable execute_stmt )
417         ( OnPatternLoad execute_stmt )
418         ( OnPowerDown execute_stmt )
419         ( OnReset execute_stmt )
420         ( OnSiteEnd execute_stmt )
421         ( OnSiteStart execute_stmt )
422         ( OnStart execute_stmt )
423         ( OnUnload execute_stmt )
424         ( OnWaferEnd execute_stmt )
425         ( OnWaferStart execute_stmt )
426     } // End Entry Points
427 } // end TestProgram
428
429 =====

```

For Multi-Site

```

430
431
432 // PROPOSED – How do we want to specify different test programs for different sites?
433 // Is this necessary?
434 ( SiteDefs (SITE_DEF_NAME) {
435
436     // Variables are global across all sites (one copy across all sites)
437     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
438
439     ( NumSites integer_expr ; )
440
441     // One statement per site. Only one can be default. If there are more sites than statements, then all
442     // unspecified sites will use the default program.
443     // It has been stated that we'd like to accommodate a scenario in which the same test program is
444     // running on all sites, but a different flow is running on each site. Using the syntax proposed here,
445     // it is possible do this by specifying different test programs for each site. The test programs are
446     // identical EXCEPT that the Flow specified by OnStart is different.
447     ( (Default) SITE_NAME TestProgram TEST_PROGRAM_NAME;)+
448 })
449
450 SPEC_NAME.NumCategories returns the number of categories in a spec block (as an integer)
451
452 SPEC_NAME.Category[I].Name returns the category name corresponding to the Ith category in a spec block
453 (first category is index 0)
454
455 SPEC_NAME.Category[I].NumVariables returns the number of variables in a Ith category (first category is
456 index 0) in a spec block
457
458 SPEC_NAME.Category[I].Variables[J].Name returns the variable name corresponding to the Jth variable in
459 the Ith category within a spec block.
460

```

```

461
462 // The STIL.4-mandated contents for TestBase are shown below. If a user provides an alternate
463 // definition, that definition MUST include all the elements shown below. The purpose of
464 // TestBase is to provide a common set of elements for all TestType and FlowType definitions.
465
466 // The user can define (and redefine) TestBase any number of times prior to first use. After first use,
467 // redefinition of TestBase is NOT allowed. First use of TestBase is defined as the first user definition of a
468 // TestType or FlowType, or first instantiation of a Flow using default FlowType.
469 TestBase {
470     Parameters {
471         Out Const String TestID { InitialValue ""; } // Empty string – set when test or flow is instantiated.
472         Out Integer ExecResult { InitialValue 0; } // Return value from execution
473 // ExecResult == 0 -> Pass, execResult != 0 -> Fail
474 // 0 = Pass, non-zero = Fail
475 // Define Pass = 0; Define Fail = !Pass
476 // For the normal case of only one fail mode, ExecResult = 1 on fail.
477         In Bin FailBin { InitialValue NoBin; } // For component values, see section 3.2.5 of Ernie’s bin doc.
478         In Integer MultiSiteSerial { InitialValue False; } // From dot1, p15, table 5, False = 1;
479     }
480     PreActions { } // No PreActions
481     // TestExec;
482     PostActions { } // No PostActions
483     // Semantics: PassActions or FailActions are selected by the implied arbiter:
484     // if (ExecResult == Pass) { Do PassActions } Else { Do FailActions }
485     PassActions { } // No PassActions. Pass Actions are executed if execResult == Pass
486     FailActions { // Fail Actions are executed if execResult == Fail
487         SetBinStop FailBin; // Perform binning based on the current value of FailBin, and stop
488 // If value of FailBin is undefined, then no action (binning and stopping)
489 // takes place.
490 // This is the equivalent to:
491 // If (FailBin != NoBin) { SetBin FailBin; Stop }
492     } // end FailActions
493 } // end TestBase
494
495

```

496

STIL.4 definition of default FlowNode

497

// When an unnamed (global) FlowNode definition occurs OUTSIDE of any other scope (i.e., outside

498

// the scope of Tests or Flows, which is the only other place where it's allowed to define a FlowNode),

499

// that FlowNode definition becomes the default FlowNode definition. If the user does NOT define a

500

// default FlowNode, the following definition is used.

501

502

// The user can define (and redefine) the default FlowNode definition any number of times prior to first use.

503

// After first use, redefinition of the default FlowNode is NOT allowed. First use of the default FlowNode

504

// is defined as the first user definition of a FlowType, or first instantiation of a Flow using default

505

// FlowType.

506

FlowNode {

507

PreActions { }

508

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

509

PostActions { }

510

ExitPorts {

511

Port FAIL CurrentExec.ExecResult == Fail {

512

// Set the ExecResult of the Test or Flow containing

513

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

514

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

515

// FlowNode executed can be used. CurrentExec is an

516

// alias for EXEC_OBJECT_NAME

517

ExecResult = Fail;

518

// or

519

// ExecResult = CurrentExec.ExecResult;

520

Stop; *// Stop execution of FlowNodes – return control*

521

// to containing Test or Flow, and execute any

522

// PostActions and PassActions/FailActions of

523

// Test or Flow.

524

}

525

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

526

} *// end ExitPorts*

527

} *// end FlowNode*

528

529

530

STIL.4 definition of default FlowType

531

532

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

533

// FlowType as defined below.

534

FlowType {

535

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

536

// Parameters – use TestBase Parameters

537

// Variables – no local variables

538

// PreActions – no PreActions

539

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

540

// setResult CurrentExec.execResult;

541

// PostActions – no PostActions – use TestBase PostActions

542

// PassActions – no PassActions – use TestBase PassActions

543

FailActions {

544

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

545

}

546

}

547

548

549

550

551
552
553
554
555

STIL.4 definition of TestType NoOpType and Test NoOp

556
557
558
559
560
561
562
563

*// In order to help demonstrate concepts in STIL.4, it's necessary to define a TestType, and show how a
// Test can be instantiated using this type. Therefore, we define an example TestType called NoOpType,
// and instantiate a Test called NoOp from it. This test can be used to illustrate any of the flow concepts
// without getting bogged down in the definition of the many actual TestTypes that might be needed on any
// specific tester.*

564
565
566
567
568
569
570
571
572
573
574
575
576
577

```

TestType NoOpType {
    // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
    // Parameters – use TestBase Parameters
    // Variables – no local variables
    // PreActions – no PreActions
    // No TestExec. No operation performed. Or – use TestExec; (with no tokens).
    // Test library must contain a Test named NoOp, which sets the value of
    // NoOp.ExecResult to the appropriate value (PASS (0) or FAIL (non-zero))
    // PostActions – no PostActions
    // PassActions – no PassActions
    // FailActions – use FailActions as defined in TestBase
} // end TestType NoOp

Test NoOpType NoOp

```