

1 Syntax summary for 1450.4

```

2   stil_block_type =
3       < Category
4         | DCLevels
5         | DCSequence
6         | DCSets
7         | Environment
8         | Flow
9         | MacroDefs
10        | Pattern
11        | PatternBurst
12        | PatternExec
13        | Procedures
14        | ScanStructures
15        | Selector
16        | SignalGroups
17        | Signals
18        | Test
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      < Boolean | Integer | SignalRef | SignalVariable | String | real_var_type | stil_block_type >
40
41  initial_value_stmt =
42      < InitialValue < 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
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        >
50        | test_elements_stmt+ // allow if var of type TestType
51        | flow_elements_stmt+ // allow if var of type FlowType
52      >
53
54

```

```

55  action =
56      < VAR_NAME = expr; // Scalar variables
57      | VAR_NAME[INDEX] = expr; // Array variables
58      | SetBin (BIN_AXIS.)SOFTBIN_NAME; >
59
60  action_stmt =
61      < If | Else If boolean_expr | Else { // Usual rules for If/Else If/Else apply
62          ( action )*
63      }
64      | ( action )* >
65
66  bypass_actions =
67      <
68          // For FlowNodes, Flows, and Tests
69          // Skip only Test execution or FlowNode sequence execution, resume at entry to
70          // PostActions Block. Normal processing continues from there. Execute PostActions, and
71          // PassActions/FailActions (as determined by FailFlag of Test or Flow), or ExitPort
72          // selection Actions (as determined by flownode exit port selector). Selection of Pass/Fail path
73          // (for Tests and Flows) is controlled by FailFlag (which can be forced to a desired state prior to
74          // Bypass statement). Selection of ExitPort path is also controlled by boolean expressions –
75          // typically, the return status of the test executed by the FlowNode. This return state (or value
76          // of any other boolean expression) can be forced to a desired state prior to the Bypass statement.
77          Bypass;
78
79          // For Flows and Tests only.
80          // Skip Test and PostActions, resume execution at entry to PassActions/FailActions.
81          // If Pass or Fail specified, follow that path. Otherwise, VAR_NAME is a string variable which
82          // specifies either PASS or FAIL. If using VAR_NAME, and it's an empty string, no bypass action
83          // occurs. If SkipActions specified, don't execute PassActions/FailActions (equivalent to a return,
84          // since there's no branching from a Test or Flow)
85          | Bypass (GoTo <Pass | Fail | VAR_NAME > ( SkipActions ) );
86
87          // For FlowNodes only.
88          // Skip Test and PostActions, resume execution at entry to specified ExitPort.
89          // If SkipActions specified, don't execute Pass/Fail Actions.
90          // PORTLABEL is a string specifying a port label. VAR_NAME is a string variable which specifies the
91          // port label, or evaluates to an integer specifying the port index. If using VAR_NAME, and it's an
92          // empty string, no bypass action occurs.
93          | Bypass (GoTo <port_index | PORTLABEL | VAR_NAME > ( SkipActions ) );
94
95  bypass_stmt =
96      < bypass_actions | If boolean_expr { bypass_actions } >
97
98  exit_port_stmt =
99      < action_stmt
100      | ReturnState integer_expr; // default to last TestExec return value
101      | Next (FLOW_NODE_NAME); // only allowed in FlowNode
102      | Return (integer_expr); // terminate the Test or Flow; return to caller
103      | Return (integer_expr) (Retest (integer_expr) ); // terminate the Test or Flow; return to caller
104                                     // repeat test specified number of times upon fail
105                                     // If Retest count is missing, use count = 1
106      | Stop; // terminate the initiating On* condition
107      | Reset; // terminate the initiating On* condition; generate an OnReset event
108      >
109
110

```

```

111  func_stmt =
112      < FUNC_NAME ; // name of a Function from FunctionDefs block
113      | FUNC_NAME { // name of a Function from FunctionDefs block
114          (VAR_NAME = expr;)*
115          (VAR_NAME = [ expr (expr)+];)* // For arrays (?)
116      } // end func_stmt
117  >
118
119  test_elements_stmt =
120      < VAR_NAME = expr;
121      | VAR_NAME = [ expr (expr)+]; // For arrays (?)
122      | PreActions { ( < action_stmt | bypass_stmt )* } )
123      | FuncExec func_stmt | FuncExec { ( func_stmt )* }
124      | PostActions { ( action_stmt )* }
125      | PassActions { ( exit_port_stmt )* }
126      | FailActions { ( exit_port_stmt )* }
127  >
128
129  test_instance_stmt =
130      // create a named Test from a TestType, using the default elements for the TestType
131      TEST_TYPE TEST_INSTANCE_NAME;
132
133      // create a named Test from a TestType, overriding some or all of the default elements of the TestType.
134      // Any element specified in the instantiation (i.e., PreActions, FailActions) completely replaces that
135      // element as specified in the type definition.
136      | TEST_TYPE TEST_INSTANCE_NAME { ( test_elements_stmt )* }
137
138  flownode_stmt =
139      FlowNode (NODE_NAME) {
140          ( PreActions { ( < action_stmt | bypass_stmt )* } )
141          ( TestExec execute_stmt | TestExec { ( execute_stmt )* } )
142          ( PostActions { ( action_stmt )* } )
143
144          ( ExitPorts {
145              ( Port <port_index> (PORTLABEL:) boolean_expr { ( exit_port_stmt )* } )*
146          } ) // end ExitPorts
147
148      } // end FlowNode
149
150  flow_elements_stmt =
151      < VAR_NAME = expr;
152      | VAR_NAME = [ expr (expr)+]; // For arrays (?)
153      | PreActions { ( < action_stmt | bypass_stmt )* } )
154      | Title STRING ;
155      | ( flownode_stmt )*
156      | PostActions { ( action_stmt )* } )
157      | PassActions { ( exit_port_stmt )* } // end PassActions
158      | FailActions { ( exit_port_stmt )* } // end FailActions
159  >
160
161
162
163
164
165
166

```

```

167 flow_instance_stmt =
168     // create a named Flow from a FlowType, using the default elements for the FlowType
169     < (FLOW_TYPE) FLOW_INSTANCE_NAME ; // if FLOW_TYPE is not specified, then the
170     // STIL.4 default FLOW_TYPE is used
171
172     // create a named Flow from a FlowType, overriding some or all of the default elements of the FlowType.
173     // Any element specified in the instantiation (i.e., PreActions, flownodes, PassActions) completely
174     // replaces that element as specified in the type definition.
175     | (FLOW_TYPE) FLOW_INSTANCE_NAME { ( flow_elements_stmt )* } > // if FLOW_TYPE is not specified, then the
176     // STIL.4 default FLOW_TYPE is used
177
178 test_execute_stmt =
179     < TEST_NAME ; // execute a named Test
180     | TEST_TYPE ; // create and execute a temporary inline Test (named _INLINE_TEST)
181     // from TestType, using the type's default element value.
182
183     | TEST_TYPE { ( test_elements_stmt )* } > // Create and execute a temporary inline Test
184     // (named _INLINE_TEST) from TestType,
185     // overriding the type's default element values
186
187 flow_execute_stmt =
188     < FLOW_NAME ; // execute a named Flow
189     | FLOW_TYPE ; // create and execute a temporary inline Flow (named _INLINE_FLOW)
190     // from FlowType, using the type's default element values
191     | FLOW_TYPE { ( flow_elements_stmt )* } > // Create and execute an atemporary inline Flow
192     // (named _INLINE_FLOW) from FlowType,
193     // overriding the type's default element values
194
195 execute_stmt = < test_execute_stmt | flow_execute_stmt >
196
197 var_elements_stmt =
198     < var_type VAR_NAME (Const) (Operator) ;
199     | var_type VAR_NAME (Const) (Operator) {
200         ( Length integer ; ) // array of var_type HOW TO INDEX????
201         ( initial_value_stmt )
202     }
203
204 =====
205 STIL 1.0 {
206     ( Flow 2007 ; )+
207 }
208
209 =====
210 Variables ( VAR_DOMAIN ) { // extensions to 1450.1
211     var_elements_stmt*
212 }
213
214 =====
215 FunctionsDefs {
216     ( FUNCTION_NAME {
217         ( Parameters {
218             ( var_type VAR_NAME (< In | Out | InOut > ; ))*
219             ( var_type VAR_NAME (< In | Out | InOut > ) { initial_value_stmt } )*)
220         } ) // end Parameters
221     })* // end function_name
222 } // end FunctionDefs

```

```

223 =====
224 BinDefs (BIN_DEF_NAME) { // soft bin definitions
225   Pass { // Increment Pass-bin if ReturnState is Pass
226     ( Bin SOFTBIN_NAME (integer); )* |
227     ( Axis BIN_AXIS {
228       ( Bin SOFTBIN_NAME (integer); )*
229     })*
230   } // end Pass
231   Fail { // Increment Fail-bin if ReturnState is Fail
232     ( Bin SOFTBIN_NAME (integer); )* |
233     ( Axis BIN_AXIS {
234       ( Bin SOFTBIN_NAME (integer); )*
235     })*
236   } // end Fail
237 }
238
239 SoftBin \ ( SOFTBIN_NAME \ ) // function call to return contents of a softbin in an expression
240
241 bin_map_stmt =
242   SOFTBIN_NAME -> integer; |
243   ( [ BIN_AXIS.SOFTBIN_NAME ] )* -> integer;
244
245 =====
246 BinMap BIN_MAP_NAME { // soft to hard bin mapping
247   (bin_map_stmt)*
248 }
249
250 =====
251 TestBase {
252   ( Parameters {
253     ( (<In | Out | InOut>) var_type VAR_NAME; )*
254     ( (<In | Out | InOut>) var_type VAR_NAME; ) { initial_value_stmt } )*
255   } ) // end Parameters
256   ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN )*
257   ( PreActions { ( < action_stmt | bypass_stmt )* } )
258   ( PostActions { ( action_stmt )* } )
259   ( PassActions { ( exit_port_stmt )* } ) // end PassActions
260   ( FailActions { ( exit_port_stmt )* } ) // end FailActions
261 } // end TestBase
262
263 =====
264 TestType TEST_TYPE_NAME {
265   ( UseDefaults; | Inherit TEST_TYPE_NAME ; ) // UseDefaults means to use TestBase
266   ( Parameters {
267     ( (<In | Out | InOut>) var_type VAR_NAME; )*
268     ( (<In | Out | InOut>) var_type VAR_NAME; ) { initial_value_stmt } )*
269   } ) // end Parameters
270   ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN )*
271   ( PreActions { ( < action_stmt | bypass_stmt )* } )
272   ( TestExec; | FuncExec func_stmt | FuncExec { ( func_stmt )* }
273   | flownode_stmt+ ) // Allows inline instantiation of a flow in a Test. That flow can't be reused.
274   ( PostActions { ( action_stmt )* } )
275   ( PassActions { ( exit_port_stmt )* } ) // end PassActions
276   ( FailActions { ( exit_port_stmt )* } ) // end FailActions
277 } // end TestType
278

```

```

279 =====
280 ( Test test_instance_stmt ) * // Instantiate a specific Test from a given TestType // Either this form . . .
281 ( Tests { ( test_instance_stmt ) * } ) // Instantiate named Test(s) from TestType(s) // or this one
282 =====
283 FlowType FLOW_TYPE_NAME {
284   ( UseDefaults; | Inherit TEST_TYPE_NAME ; ) // UseDefaults means to use TestBase
285   ( Parameters {
286     ( (<In | Out | InOut>) var_type VAR_NAME ; ) *
287     ( (<In | Out | InOut>) var_type VAR_NAME ; ) { initial_value_stmt } ) *
288   } ) // end Parameters
289   ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN ) *
290   ( PreActions { ( < action_stmt | bypass_stmt ) * } )
291   ( Title STRING ; )
292   ( flownode_stmt ) *
293   ( PostActions { ( post_action_stmt* } ) )
294   ( PassActions { ( exit_port_stmt* } ) // end PassActions
295   ( FailActions { ( exit_port_stmt* } ) // end FailActions
296 } // end FlowType
297
298 =====
299 ( Flow flow_instance_stmt ) * // Instantiate a specific Flow from a given FlowType // Either this form . . .
300 ( Flows { ( flow_instance_stmt ) * } ) // Instantiate named Flow(s) from FlowType(s) // or this one
301
302 =====
303 TestProgram TEST_PROGRAM_NAME {
304   DUTType "DUT NAME STRING";
305   SocketDef "SOCKET NAME STRING";
306   // Variables are global to test program and below; local to site (a copy for each site)
307   ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN ) *
308   ( BinDefs BIN_DEF_NAME ; )
309   ( BinMap BIN_MAP_NAME ; )
310
311   EntryPoints {
312     ( OnException execute_stmt )
313     ( OnFinish execute_stmt )
314     ( OnLoad execute_stmt )
315     ( OnLotEnd execute_stmt )
316     ( OnLotStart execute_stmt )
317     ( OnMultiSiteDisable execute_stmt )
318     ( OnMultiSiteEnable execute_stmt )
319     ( OnPatternLoad execute_stmt )
320     ( OnPowerDown execute_stmt )
321     ( OnReset execute_stmt )
322     ( OnSiteEnd execute_stmt )
323     ( OnSiteStart execute_stmt )
324     ( OnStart execute_stmt )
325     ( OnUnload execute_stmt )
326     ( OnWaferEnd execute_stmt )
327     ( OnWaferStart execute_stmt )
328   } // End Entry Points
329 } // end TestProgram
330
331
332
333
334

```

```

335 =====
336 // The TestDefaults block is a container within which the default definitions for TestBase, FlowType, and
337 // FlowNode
338 TestDefaults {
339     // The STIL.4-mandated contents for TestBase are shown below. If a user provides an alternate definition,
340     // that definition MUST include all the elements shown below. The purpose of TestBase is to provide a
341     // common set of elements for all TestType and FlowType definitions.
342     TestBase {
343         // See “STIL.4 mandatory requirements for TestBase” below
344     } // end TestBase
345
346     // STIL.4 default FlowNode definition.
347     FlowNode {
348         // See “STIL.4 definition of default FlowNode” below
349     } // end FlowNode
350
351     // The flowtype name stil_dot_4_default is arbitrary – but it MUST be defined by the
352     // standard, just as we’ve defined what TestBase will contain.
353     FlowType stil_dot_4_default {
354         // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
355         // Parameters – use TestBase Parameters
356         // Variables – no local variables
357         // PreActions – no PreActions
358         // Flownodes – no flownodes specified in the type – will be provided at instantiation
359         // PostActions – no PostActions
360         // PassActions – no PassActions
361         // FailActions – use FailActions as defined in TestBase
362     } // end FlowType
363 } // end TestDefaults
364

```

For Multi-Site

```

365
366
367 // PROPOSED – How do we want to specify different test programs for different sites?
368 // Is this necessary?
369 ( SiteDefs (SITE_DEF_NAME) {
370
371     // Variables are global across all sites (one copy across all sites)
372     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
373
374     ( NumSites integer_expr ; )
375
376     // One statement per site. Only one can be default. If there are more sites than statements, then all
377     // unspecified sites will use the default program.
378     // It has been stated that we'd like to accommodate a scenario in which the same test program is
379     // running on all sites, but a different flow is running on each site. Using the syntax proposed here,
380     // it is possible do this by specifying different test programs for each site. The test program are
381     // identical EXCEPT that the Flow specified by OnStart is different.
382     ( (Default) SITE_NAME TestProgram TEST_PROGRAM_NAME;)+
383 })
384
385 SPEC_NAME.NumCategories returns the number of categories in a spec block (as an integer)
386
387 SPEC_NAME.Category[I].Name returns the category name corresponding to the Ith category in a spec block
388 (first category is index 0)
389
390 SPEC_NAME.Category[I].NumVariables returns the number of variables in a given category in a spec
391 block
392
393 SPEC_NAME.Category[I].Variables[J].Name returns the variable name corresponding to the Jth variable in
394 the Ith category within a spec block.
395

```

```

396
397 STIL.4 mandatory requirements for TestBase
398 // The STIL.4-mandated contents for TestBase are shown below. If a user provides an alternate
399 // definition, that definition MUST include all the elements shown below. The purpose of
400 // TestBase is to provide a common set of elements for all TestType and FlowType definitions.
401 TestBase {
402   Parameters {
403     Out String TestID { InitialValue ""; } // Empty string
404     Out Boolean FailFlag { InitialValue False; } // Did test pass or fail? Orthogonal to Result, ReturnVal
405     Out Real Result { InitialValue NaN; } // Result – orthogonal to ReturnVal, FailFlag
406     Out Int ReturnVal { InitialValue 0; } // Return value from execution – orthogonal to Result,
407     // FailFlag
408     In Bin FailBin; // Initial Value = undefined – equivalent to no bin.
409   }
410   PreActions { } // No PreActions
411   PostActions { } // No PostActions
412   // PassActions or FailActions are selected by the implied arbiter:
413   // if (FailFlag == True) { Do FailActions} Else { Do PassActions }
414   PassActions { } // No PassActions. Pass Actions are executed if FailFlag == False
415   FailActions { // Fail Actions are executed if FailFlag == True
416     SetBin FailBin; // Perform binning based on the current value of FailBin.
417     // If value of FailBin is undefined, then no binning takes place
418     // Do we want to make the above action explicit? i.e.
419     // If (FailBin != Undefined) { SetBin FailBin; }
420     // Vet this against industrial practice (ASAP, SmarTest, OTPL, Envision, Stylus)
421     Return FailFlag; // Return to caller (typically the flow node executor) for decision branching
422   } // end FailActions
423 } // end TestBase
424

```

425 **STIL.4 definition of default FlowNode**
 426 **FlowNode** {
 427 **PreActions** { }
 428 **TestExec** <exec_object_name>;
 429 **PostActions** { }
 430 **ExitPorts** {
 431 Port 0 FAIL: <exec_object_name>.FailFlag == TRUE { **SetBin** <exec_object_name>.FailBin; Stop; }
 432 Port 1 PASS: <exec_object_name>.FailFlag == FALSE { Next; }
 433 } // end ExitPorts
 434 } // end FlowNode

435
 436
 437
 438 **STIL.4 definition of default FlowType**
 439
 440 // The flowtype name **stil_dot_4_default** is arbitrary – but it **MUST** be defined by the
 441 // standard, just as we’ve defined what **TestBase** will contain.

442 **FlowType stil_dot_4_default** {
 443 // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
 444 // Parameters – use TestBase Parameters
 445 // Variables – no local variables
 446 // PreActions – no PreActions
 447 // Flownodes – no flownodes specified in the type – will be provided at instantiation
 448 // PostActions – no PostActions
 449 // PassActions – no PassActions
 450 // FailActions – use FailActions as defined in TestBase
 451 }
 452
 453

454 **STIL.4 definition of TestType NoOpType and Test NoOp**

455
 456 In order to help demonstrate concepts in STIL.4, it’s necessary to define a TestType, and show how a Test
 457 can be instantiated using this type. Therefore, we define an example TestType called NoOpType, and
 458 instantiate a Test called NoOp from it. This test can be used to illustrate any of the flow concepts without
 459 getting bogged down in the definition of the many actual TestTypes that might be needed on any specific
 460 tester.

461
 462 **TestType NoOpType** {
 463 // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
 464 // Parameters – use TestBase Parameters
 465 // Variables – no local variables
 466 // PreActions – no PreActions
 467 // No TestExec. No operation performed. Or – use TestExec; (with no tokens).
 468 // Test library must contain a Test named NoOp, which sets the
 469 // value of NoOp.FailFlag to the appropriate value (Pass or Fail)
 470 // PostActions – no PostActions
 471 // PassActions – no PassActions
 472 // FailActions – use FailActions as defined in TestBase
 473 } // end TestType NoOp
 474
 475 **Test NoOpType NoOp**
 476