

1

Revision History

2

Date	Rev	Init	Change
3/27/08	D0.18	jo	Added revision table to keep track of changes.
5/15/08	D0.19	jo	Added softbin attributes and bin property access syntax (per Ernie's recent writeups).

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       < Boolean | Integer | SignalRef | SignalVariable | String | real_var_type | stil_block_type >
42
43   initial_value_stmt =
44       < InitialValue < 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
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         >
52         | test_elements_stmt+ // allow if var of type TestType
53         | flow_elements_stmt+ // allow if var of type FlowType
54         >
55
56   initial_value_stmt = InitialValue <initial_val_elements> | InitialValue [ (initial_val_elements)+ ]

```

```

57  action =
58      < VAR_NAME = expr; // Scalar variables
59      | VAR_NAME[INDEX] = expr; // Array variables
60      // Only one element of each bin axis can be active at any one time. For 3 separate axes, you can
61      // have 3 separate softbins (one from each axis) set – the binmap handles the various intersections
62      // when the part is binned.
63      | SetBin (BIN_AXIS.)SOFTBIN_NAME; >
64
65  action_stmt =
66      < If | Else If boolean_expr | Else { // Usual rules for If/Else If/Else apply
67          ( action )*
68      }
69      | ( action )* >
70
71  bypass_actions =
72      <
73      // For FlowNodes, Flows, and Tests
74      // Skip only Test execution or FlowNode sequence execution, resume at entry to PostActions Block
75      // Normal processing continues from there. Execute PostActions, and PassActions/FailActions (as
76      // determined by FailFlag of Test or Flow), or ExitPort selection Actions (as determined by flownode
77      // exit port selector). Selection of Pass/Fail path (for Tests and Flows) is controlled by FailFlag (which
78      // can be forced to a desired state prior to Bypass statement). Selection of ExitPort path (for flownodes)
79      // is also controlled by boolean expressions – typically, the return status of the test executed by the
80      // FlowNode. This return state (or value of any other boolean expression) can be forced to a desired state
81      // prior to the Bypass statement.
82      Bypass;
83
84      // For Flows and Tests only.
85      // Skip Test and PostActions, resume execution at entry to PassActions/FailActions.
86      // If Pass or Fail specified, follow that path. Otherwise, VAR_NAME is a string variable which specifies
87      // either PASS or FAIL. If using VAR_NAME, and it's an empty string, no bypass action occurs
88      // If SkipActions specified, don't execute PassActions/FailActions (equivalent to a return,
89      // since there's no branching from a Test or Flow)
90      | Bypass (GoTo <Pass | Fail | VAR_NAME > ( SkipActions ) );
91
92      // For FlowNodes only.
93      // Skip Test and PostActions, resume execution at entry to specified ExitPort.
94      // If SkipActions specified, don't execute don't execute the exit port actions.
95      // PORTLABEL is a string specifying a port label. VAR_NAME is either a string variable or an integer variable.
96      // If a string variable, it specifies the bypass exit port by label. If using VAR_NAME, and it's an empty string,
97      // no bypass action occurs. If an integer variable, it specifies the bypass exit port by index (based on the
98      // ordinal order of the exit ports, from top of list to bottom, with the first index being 0.
99      // JO 2/29/08: Removed portindex as a bypass option. Updated semantics if VAR_NAME is type int
100     | Bypass (GoTo < PORTLABEL | VAR_NAME > ( SkipActions ) );
101
102  bypass_stmt =
103      < bypass_actions | If boolean_expr { bypass_actions } >
104
105  exit_port_stmt =
106      < action_stmt
107      | ReturnState integer_expr; // default to last TestExec return value
108      | Next (FLOW_NODE_NAME); // only allowed in FlowNode
109      | Return (integer_expr) (Retest (integer_expr)); // terminate the Test or Flow; return to caller
110      // repeat test specified number of times upon fail
111      // If Retest count is missing, use count = 1
112      | Stop; // terminate the initiating On* condition

```

```

113     | Reset; // terminate the initiating On* condition; generate an OnReset event
114     >
115
116   func_stmt =
117     < FUNC_NAME ; // name of a Function from FunctionDefs block
118     | FUNC_NAME { // name of a Function from FunctionDefs block
119         (VAR_NAME = expr)*
120         (VAR_NAME = [ expr (expr)+];)* // For arrays (?)
121     } // end func_stmt
122     >
123
124   test_elements_stmt =
125     < VAR_NAME = expr;
126     | VAR_NAME = [ expr (expr)+]; // For arrays (?)
127     | PreActions { ( < action_stmt | bypass_stmt )* } )
128     | FuncExec func_stmt | FuncExec { ( func_stmt )* }
129     | PostActions { ( action_stmt )* }
130     | PassActions { ( exit_port_stmt )* }
131     | FailActions { ( exit_port_stmt )* }
132     >
133
134   test_instance_stmt =
135     // create a named Test from a TestType, using the default elements for the TestType
136     TEST_TYPE TEST_INSTANCE_NAME;
137
138     // create a named Test from a TestType, overriding some or all of the default elements of the TestType.
139     // Any element specified in the instantiation (i.e., PreActions, FailActions) completely replaces that
140     // element as specified in the type definition.
141     | TEST_TYPE TEST_INSTANCE_NAME { ( test_elements_stmt )* }
142
143   flownode_stmt =
144     FlowNode (NODE_NAME) {
145       ( PreActions { ( < action_stmt | bypass_stmt )* } )
146       ( TestExec execute_stmt | TestExec { ( execute_stmt )* } )
147       ( PostActions { ( action_stmt )* } )
148
149       ( ExitPorts {
150         // JO – 02/29/08: Removed portindex (was used as a bypass option. Ordinal index order
151         // (starting with 0) is now used instead. PORTLABEL is now required, not optional.
152         ( PORTLABEL: boolean_expr { ( exit_port_stmt )* } ) *
153       } ) // end ExitPorts
154
155     } // end FlowNode
156
157   flow_elements_stmt =
158     < VAR_NAME = expr;
159     | VAR_NAME = [ expr (expr)+]; // For arrays (?)
160     | PreActions { ( < action_stmt | bypass_stmt )* } )
161     | Title STRING ;
162     | ( flownode_stmt )*
163     | PostActions { ( action_stmt )* } )
164     | PassActions { ( exit_port_stmt )* } // end PassActions
165     | FailActions { ( exit_port_stmt )* } // end FailActions
166     >
167
168

```

```

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

```

```

225 =====
226 softbin_attribute =
227 <
228   Color <String>; | // Hex, RGB, or name
229   Number <Unsigned Integer>; |
230   Retest <Unsigned Integer>; |
231   Terse <String>; |
232   Verbose <String>; |
233   WafermapChar <simple_character>; // From P1450.1999 BNF
234 >
235
236 softbin_definition =
237   Bin SOFTBIN_NAME ; |
238   SOFTBIN_NAME { (softbin_attribute)* }
239
240   BinDefs (BIN_DEF_NAME) { // soft bin definitions
241     Pass { // Increment Pass-bin if ReturnState is Pass
242       (Color <String>;) // Contains default color for all Pass bins, “green” if unspecified
243       (softbin_definition)* |
244       ( Axis BIN_AXIS_NAME {
245         (softbin_definition)*
246       } )*
247     } // end Pass
248     Fail { // Increment Fail-bin if ReturnState is Fail
249       (Color <String>;) // Contains default color for all Fail bins, “red” if unspecified
250       (softbin_definition)* |
251       ( Axis BIN_AXIS_NAME {
252         (softbin_definition)*
253       } )*
254     } // end Fail
255   }
256
257   // In the second form of the statement shown below, if more than one BIN_AXIS_NAME.SOFTBIN_NAME
258   // expression is present, a comma is used to separate the list
259   bin_map_stmt =
260     Map SOFTBIN_NAME integer; |
261     Map[ (BIN_AXIS_NAME.SOFTBIN_NAME )+( , )] integer;
262
263 =====
264   BinMap BIN_MAP_NAME { // soft to hard bin mapping
265     (bin_map_stmt )*
266   }
267
268 -----
268 Bin Property access syntax
269 counter_reset_event =
270 <
271   OnLoad |
272   OnLotStart |
273   OnRetest |
274   OnSiteStart | // What’s the difference between OnSiteStart
275   OnStart | // and OnStart ?
276   OnWaferStart
277 >
278
279
280

```

```

281 bin_property =
282 <
283     Color |                               // String
284     ContinueOnFail |                       // Boolean
285     counter.counter_reset_event |         // Unsigned
286     Enabled |                               // Boolean
287     Index |                                 // Unsigned
288     isSet.counter_reset_event |           // Boolean
289     Name |                                 // String
290     Number |                               // Integer
291     retest.(current|Original) |           // Unsigned
292     Terse |                                 // String
293     Verbose |                             // String
294     WafermapChar |                       // Character
295 >
296
297
298 group = < Pass|Fail >
299
300 (BINDEFS_NAME.) group[Unsigned|SOFTBIN_NAME ].bin_property |
301 (BINDEFS_NAME.) group[Unsigned|AXIS_NAME ][Unsigned|SOFTBIN_NAME ].bin_property
302
303
304
305
306
307
308
309
310
311
312

```

```

313 =====
314 TestBase {
315     ( Parameters {
316         ( (<In | Out | InOut>) var_type VAR_NAME; ) *
317         ( (<In | Out | InOut>) var_type VAR_NAME; ) { initial_value_stmt } ) *
318     } // end Parameters
319     ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN ) *
320     ( PreActions { ( < action_stmt | bypass_stmt ) * } )
321     ( PostActions { ( action_stmt ) * } )
322     ( PassActions { ( exit_port_stmt ) * } ) // end PassActions
323     ( FailActions { ( exit_port_stmt ) * } ) // end FailActions
324 } // end TestBase
325
326 =====
327 TestType TEST_TYPE_NAME {
328     // If not inheriting from a previously-defined TEST_TYPE_NAME (either vendor-defined or user-defined),
329     // a TestType will inherit from TestBase. The “Inherit TestBase” statement is not required, but is
330     // recommended. If no Inherit statement is present, the behavior is equivalent to “Inherit TestBase”
331     // JO 3/24/08: Changed “UseDefaults” to “Inherit TestBase”. Clarified semantics if “Inherit” statement is absent
332     ( Inherit TestBase; | Inherit TEST_TYPE_NAME ; )
333     ( Parameters {
334         ( (<In | Out | InOut>) var_type VAR_NAME; ) *
335         ( (<In | Out | InOut>) var_type VAR_NAME; ) { initial_value_stmt } ) *
336     } // end Parameters
337     ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN ) *
338     ( PreActions { ( < action_stmt | bypass_stmt ) * } )
339     ( TestExec; | FuncExec func_stmt | FuncExec { ( func_stmt ) * }
340     | flownode_stmt+ ) // Allows inline instantiation of a flow in a Test. That flow can't be reused.
341     ( PostActions { ( action_stmt ) * } )
342     ( PassActions { ( exit_port_stmt ) * } ) // end PassActions
343     ( FailActions { ( exit_port_stmt ) * } ) // end FailActions
344 } // end TestType
345
346 =====
347     ( Tests { ( test_instance_stmt ) * } ) // Instantiate named Test(s) from TestType(s)
348
349 =====
350 FlowType FLOW_TYPE_NAME {
351     // If not inheriting from a previously-defined FLOW_TYPE_NAME (either vendor-defined or user-defined),
352     // a FlowType will inherit from TestBase. The “Inherit TestBase” statement is not required, but is
353     // recommended. If no Inherit statement is present, the behavior is equivalent to “Inherit TestBase”
354     // JO 3/24/08: Changed “UseDefaults” to “Inherit TestBase”. Clarified semantics if “Inherit” statement is absent
355     ( Inherit TestBase; | Inherit FLOW_TYPE_NAME ; )
356     ( Parameters {
357         ( (<In | Out | InOut>) var_type VAR_NAME; ) *
358         ( (<In | Out | InOut>) var_type VAR_NAME; ) { initial_value_stmt } ) *
359     } // end Parameters
360     ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN ) *
361     ( PreActions { ( < action_stmt | bypass_stmt ) * } )
362     ( Title STRING ; )
363     ( flownode_stmt ) *
364     ( PostActions { ( post_action_stmt ) * } )
365     ( PassActions { ( exit_port_stmt ) * } ) // end PassActions
366     ( FailActions { ( exit_port_stmt ) * } ) // end FailActions
367 } // end FlowType
368

```

```

369 =====
370 ( Flows { (flow_instance_stmt)* } ) // Instantiate named Flow(s) from FlowType(s)
371
372 =====
373 TestProgram TEST_PROGRAM_NAME {
374     DUTType "DUT NAME STRING";
375     SocketDef "SOCKET NAME STRING";
376     // Variables are global to test program and below; local to site (a copy for each site)
377     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
378     ( BinDefs BIN_DEF_NAME; )
379     ( BinMap BIN_MAP_NAME; )
380
381     EntryPoints {
382         ( OnException execute_stmt )
383         ( OnFinish execute_stmt )
384         ( OnLoad execute_stmt )
385         ( OnLotEnd execute_stmt )
386         ( OnLotStart execute_stmt )
387         ( OnMultiSiteDisable execute_stmt )
388         ( OnMultiSiteEnable execute_stmt )
389         ( OnPatternLoad execute_stmt )
390         ( OnPowerDown execute_stmt )
391         ( OnReset execute_stmt )
392         ( OnSiteEnd execute_stmt )
393         ( OnSiteStart execute_stmt )
394         ( OnStart execute_stmt )
395         ( OnUnload execute_stmt )
396         ( OnWaferEnd execute_stmt )
397         ( OnWaferStart execute_stmt )
398     } // End Entry Points
399 } // end TestProgram
400
401 =====
402 // The TestDefaults block is a container within which the default definitions for TestBase, FlowType, and
403 // FlowNode
404 TestDefaults {
405     // The STIL.4-mandated contents for TestBase are shown below. If a user provides an alternate definition,
406     // that definition MUST include all the elements shown below. The purpose of TestBase is to provide a
407     // common set of elements for all TestType and FlowType definitions.
408     TestBase {
409         // See "STIL.4 mandatory requirements for TestBase" below
410     } // end TestBase
411
412     // STIL.4 default FlowNode definition.
413     FlowNode {
414         // See "STIL.4 definition of default FlowNode" below
415     } // end FlowNode
416
417     // The flowtype name stil_dot_4_default is arbitrary – but it MUST be defined by the
418     // standard, just as we've defined what TestBase will contain.
419     FlowType stil_dot_4_default {
420         // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
421         // Parameters – use TestBase Parameters
422         // Variables – no local variables
423         // PreActions – no PreActions
424         // Flownodes – no flownodes specified in the type – will be provided at instantiation

```

```
425         // PostActions – no PostActions
426         // PassActions – no PassActions
427         // FailActions – use FailActions as defined in TestBase
428     } // end FlowType
429 } // end TestDefaults
430
```

For Multi-Site

```

431
432
433 // PROPOSED – How do we want to specify different test programs for different sites?
434 // Is this necessary?
435 ( SiteDefs (SITE_DEF_NAME) {
436
437     // Variables are global across all sites (one copy across all sites)
438     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
439
440     ( NumSites integer_expr ; )
441
442     // One statement per site. Only one can be default. If there are more sites than statements, then all
443     // unspecified sites will use the default program.
444     // It has been stated that we'd like to accommodate a scenario in which the same test program is
445     // running on all sites, but a different flow is running on each site. Using the syntax proposed here,
446     // it is possible do this by specifying different test programs for each site. The test programs are
447     // identical EXCEPT that the Flow specified by OnStart is different.
448     ( (Default) SITE_NAME TestProgram TEST_PROGRAM_NAME;)+
449 })
450
451 SPEC_NAME.NumCategories returns the number of categories in a spec block (as an integer)
452
453 SPEC_NAME.Category[I].Name returns the category name corresponding to the Ith category in a spec block
454 (first category is index 0)
455
456 SPEC_NAME.Category[I].NumVariables returns the number of variables in a given category in a spec
457 block
458
459 SPEC_NAME.Category[I].Variables[J].Name returns the variable name corresponding to the Jth variable in
460 the Ith category within a spec block.
461

```

```

462
463 STIL.4 mandatory requirements for TestBase
464 // The STIL.4-mandated contents for TestBase are shown below. If a user provides an alternate
465 // definition, that definition MUST include all the elements shown below. The purpose of
466 // TestBase is to provide a common set of elements for all TestType and FlowType definitions.
467 TestBase {
468   Parameters {
469     Out String TestID { InitialValue ""; } // Empty string
470     Out Boolean FailFlag { InitialValue False; } // Did test pass or fail? Orthogonal to Result, ReturnVal
471     Out Real Result { InitialValue NaN; } // Result – orthogonal to ReturnVal, FailFlag
472     Out Int ReturnVal { InitialValue 0; } // Return value from execution – orthogonal to Result,
473     // FailFlag
474     In Bin FailBin; // Initial Value = undefined – equivalent to no bin.
475   }
476   PreActions { } // No PreActions
477   PostActions { } // No PostActions
478   // Semantics: PassActions or FailActions are selected by the implied arbiter:
479   // if (FailFlag == True) { Do FailActions} Else { Do PassActions }
480   PassActions { } // No PassActions. Pass Actions are executed if FailFlag == False
481   FailActions { // Fail Actions are executed if FailFlag == True
482     SetBin FailBin; // Perform binning based on the current value of FailBin.
483     // If value of FailBin is undefined, then no binning takes place
484     // This is the equivalent to:
485     // If (FailBin != Undefined) { SetBin FailBin; }
486     // Vet this against industrial practice (ASAP, SmarTest, OTPL, Envision, Stylus)
487     Return FailFlag; // Return to caller (typically the flow node executor) for decision branching
488   } // end FailActions
489 } // end TestBase
490

```

```

491
492 FlowNode {
493     PreActions { }
494     TestExec <exec_object_name>;
495     PostActions { }
496     ExitPorts {
497         Port 0 FAIL: <exec_object_name>.FailFlag == TRUE { SetBin <exec_object_name>.FailBin; Stop; }
498         Port 1 PASS: <exec_object_name>.FailFlag == FALSE { Next; }
499     } // end ExitPorts
500 } // end FlowNode

```

STIL.4 definition of default FlowType

```

501
502
503
504
505
506 // The flowtype name stil_dot_4_default is arbitrary – but it MUST be defined by the
507 // standard, just as we’ve defined what TestBase will contain.
508 FlowType stil_dot_4_default {
509     // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
510     // Parameters – use TestBase Parameters
511     // Variables – no local variables
512     // PreActions – no PreActions
513     // Flownodes – no flownodes specified in the type – will be provided at instantiation
514     // PostActions – no PostActions
515     // PassActions – no PassActions
516     // FailActions – use FailActions as defined in TestBase
517 }

```

STIL.4 definition of TestType NoOpType and Test NoOp

520
521
522 In order to help demonstrate concepts in STIL.4, it’s necessary to define a TestType, and show how a Test
523 can be instantiated using this type. Therefore, we define an example TestType called NoOpType, and
524 instantiate a Test called NoOp from it. This test can be used to illustrate any of the flow concepts without
525 getting bogged down in the definition of the many actual TestTypes that might be needed on any specific
526 tester.

```

527
528 TestType NoOpType {
529     // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
530     // Parameters – use TestBase Parameters
531     // Variables – no local variables
532     // PreActions – no PreActions
533     // No TestExec. No operation performed. Or – use TestExec; (with no tokens).
534     // Test library must contain a Test named NoOp, which sets the
535     // value of NoOp.FailFlag to the appropriate value (Pass or Fail)
536     // PostActions – no PostActions
537     // PassActions – no PassActions
538     // FailActions – use FailActions as defined in TestBase
539 } // end TestType NoOp

```

```

540
541 Test NoOpType NoOp
542
543
544

```

```
545 Flow myflowtype myflowname;
546 Flow myflowtype myflowname2 { }
547 Flow myflowtype myflowname3 { }
548
549
550 Flows {
551     myflowtype myflowname;
552     myflowtype myflowname2 { }
553     myflowtype myflowname3 { }
554 }
```