

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> (lines 57-66). <code>SetBinStop</code> is a combination of <code>SetBin <>; Stop;</code> 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 (lines 494-502), replaced <code><exec_object_name></code> with <code>EXEC_OBJECT_NAME</code> to adhere to the <code>STIL</code> BNF 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> ? |
| | | | |

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_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      // Need to describe the semantics of Stop based on particular testers
65      | Stop; // terminate the initiating On* condition >
66      | SetBinStop (BIN_AXIS.)SOFTBIN_NAME; // Equivalent to a combination of the above two statements.
67
68  action_stmt =
69      < If | Else If boolean_expr | Else { // Usual rules for If/Else If/Else apply
70          ( action )*
71      }
72      | ( action )* >
73
74  bypass_actions =
75      <
76      // For FlowNodes, Flows, and Tests
77      // Skip only Test execution or FlowNode sequence execution, resume at entry to PostActions Block
78      // Normal processing continues from there. Execute PostActions, and PassActions/FailActions (as
79      // determined by FailFlag of Test or Flow), or ExitPort selection Actions (as determined by flownode
80      // exit port selector). Selection of Pass/Fail path (for Tests and Flows) is controlled by FailFlag (which
81      // can be forced to a desired state prior to Bypass statement). Selection of ExitPort path (for flownodes)
82      // is also controlled by boolean expressions – typically, the return status of the test executed by the
83      // FlowNode. This return state (or value of any other boolean expression) can be forced to a desired state
84      // prior to the Bypass statement.
85      Bypass;
86
87      // For Flows and Tests only.
88      // Skip Test and PostActions, resume execution at entry to PassActions/FailActions.
89      // If Pass or Fail specified, follow that path. Otherwise, VAR_NAME is a string variable which specifies
90      // either PASS or FAIL. If using VAR_NAME, and it's an empty string, no bypass action occurs
91      // If SkipActions specified, don't execute PassActions/FailActions (equivalent to a return,
92      // since there's no branching from a Test or Flow)
93      | Bypass (GoTo <Pass | Fail | VAR_NAME > ( SkipActions ) );
94
95      // For FlowNodes only.
96      // Skip Test and PostActions, resume execution at entry to specified ExitPort.
97      // If SkipActions specified, don't execute don't execute the exit port actions.
98      // PORTLABEL is a string specifying a port label. VAR_NAME is either a string variable or an integer variable.
99      // If a string variable, it specifies the bypass exit port by label. If using VAR_NAME, and it's an empty string,
100     // no bypass action occurs. If an integer variable, it specifies the bypass exit port by index (based on the
101     // ordinal order of the exit ports, from top of list to bottom, with the first index being 0.
102     | Bypass (GoTo < PORTLABEL | VAR_NAME > ( SkipActions ) );
103
104  bypass_stmt =
105      < bypass_actions | If boolean_expr { bypass_actions } >
106
107  exit_port_stmt =
108      < action_stmt | Next (FLOW_NODE_NAME);>
109
110  func_stmt =
111      < FUNC_NAME ; // name of a Function from FunctionDefs block
112      | FUNC_NAME { // name of a Function from FunctionDefs block

```

```

113         (VAR_NAME = expr;)*
114         (VAR_NAME = [ expr (expr)+];)* // For arrays (?)
115     } // end func_stmt
116 >
117
118 test_elements_stmt =
119     < VAR_NAME = expr;
120     | VAR_NAME = [ expr (expr)+]; // For arrays (?)
121     | PreActions { ( < action_stmt | bypass_stmt )* } )
122     | FuncExec func_stmt | FuncExec { ( func_stmt )* }
123     | PostActions { ( action_stmt )* }
124     | PassActions { ( action_stmt )* }
125     | FailActions { ( action_stmt )* }
126 >
127
128 test_instance_stmt =
129     // create a named Test from a TestType, using the default elements for the TestType
130     TEST_TYPE TEST_INSTANCE_NAME;
131
132     // create a named Test from a TestType, overriding some or all of the default elements of the TestType.
133     // Any element specified in the instantiation (i.e., PreActions, FailActions) completely replaces that
134     // element as specified in the type definition.
135     | TEST_TYPE TEST_INSTANCE_NAME { ( test_elements_stmt )* }
136
137 flownode_stmt =
138     FlowNode (NODE_NAME) {
139         ( PreActions { ( < action_stmt | bypass_stmt )* } )
140         ( TestExec execute_stmt | TestExec { ( execute_stmt )* } )
141         ( PostActions { ( action_stmt )* } )
142         ( ExitPorts {
143             ( PORTLABEL: boolean_expr { ( exit_port_stmt )* } )
144         } ) // end ExitPorts
145     } // end FlowNode
146
147 flow_elements_stmt =
148     < VAR_NAME = expr;
149     | VAR_NAME = [ expr (expr)+]; // For arrays (?)
150     | PreActions { ( < action_stmt | bypass_stmt )* } )
151     | Title STRING ;
152     | ( flownode_stmt )*
153     | PostActions { ( action_stmt )* } )
154     | PassActions { ( action_stmt )* } // end PassActions
155     | FailActions { ( action_stmt )* } // end FailActions
156 >
157
158 flow_instance_stmt =
159     // create a named Flow from a FlowType, using the default elements for the FlowType
160     < (FLOW_TYPE) FLOW_INSTANCE_NAME ; // if FLOW_TYPE is not specified, then the
161                                         // STIL.4 default FLOW_TYPE is used
162
163     // create a named Flow from a FlowType, overriding some or all of the default elements of the FlowType.
164     // Any element specified in the instantiation (i.e., PreActions, flownodes, PassActions) completely
165     // replaces that element as specified in the type definition.
166     | (FLOW_TYPE) FLOW_INSTANCE_NAME { ( flow_elements_stmt )* } > // if FLOW_TYPE is not specified, then the
167                                         // STIL.4 default FLOW_TYPE is used
168

```

```

169
170 test_execute_stmt =
171     < TEST_NAME; // execute a named Test
172     | TEST_TYPE; // create and execute a temporary inline Test (named _INLINE_TEST)
173                     // from TestType, using the type's default element value.
174
175     | TEST_TYPE { ( test_elements_stmt )* } > // Create and execute a temporary inline Test
176                                             // (named _INLINE_TEST) from TestType,
177                                             // overriding the type's default element values
178
179 flow_execute_stmt =
180     < FLOW_NAME; // execute a named Flow
181     | FLOW_TYPE; // create and execute a temporary inline Flow (named _INLINE_FLOW)
182                     // from FlowType, using the type's default element values
183     | FLOW_TYPE { ( flow_elements_stmt )* } > // Create and execute an a temporary inline Flow
184                                             // (named _INLINE_FLOW) from FlowType,
185                                             // overriding the type's default element values
186
187 execute_stmt =< test_execute_stmt | flow_execute_stmt >
188
189 var_elements_stmt =
190     // Do we need an Operator attribute? Intended to identify variables whose values are set by a
191     // query to an operator and the operator's response (i.e., lot ID)
192     < var_type VAR_NAME ;
193     | var_type VAR_NAME {
194         ( Length integer; ) // array of var_type HOW TO INDEX????
195         ( initial_value_stmt )
196     }
197
198 =====
199 STIL 1.0 {
200     ( Flow 2007; )+
201 }
202
203 =====
204 Variables ( VAR_DOMAIN ) { // extensions to 1450.1
205     var_elements_stmt*
206 }
207
208 =====
209 FunctionsDefs {
210     ( FUNCTION_NAME {
211         ( Parameters {
212             ( var_type VAR_NAME (< In | Out | InOut > ; ))*
213             ( var_type VAR_NAME (< In | Out | InOut >) { initial_value_stmt } ))*
214         } ) // end Parameters
215     })* // end function_name
216 } // end FunctionDefs
217
218
219
220
221
222
223
224

```

```

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

```

```

281
282 bin_property =
283 <
284     Color |                               // String
285     ContinueOnFail |                       // Boolean
286     counter.counter_reset_event |         // Unsigned
287     Enabled |                             // Boolean
288     Index |                               // Unsigned
289     isSet.counter_reset_event |          // Boolean
290     Name |                                // String
291     Number |                              // Integer
292     retest.(current|Original) |          // Unsigned
293     Terse |                               // String
294     Verbose |                             // String
295     WafermapChar |                       // Character
296 >
297
298
299 group = < Pass|Fail >
300
301 (BINDEFS_NAME.) group[Unsigned| SOFTBIN_NAME ].bin_property |
302 (BINDEFS_NAME.) group[Unsigned| AXIS_NAME ][Unsigned| SOFTBIN_NAME ].bin_property
303
304
305
306
307
308
309
310
311
312
313

```

```

314 =====
315 TestBase {
316     (Parameters {
317         ((<In | Out | InOut>) var_type VAR_NAME;)*
318         ((<In | Out | InOut>) var_type VAR_NAME;){ initial_value_stmt })*
319     }) // end Parameters
320     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
321     (PreActions { (< action_stmt | bypass_stmt )* })
322     // Allows implementers to include a dummy TestExec in TestBase. The dummy TestExec can,
323     // for instance, display a warning or error that an actual Test did not include its own TestExec or
324     // FuncExec, or that an actual Flow did not include at least one FlowNode (which can be empty).
325     (TestExec;)
326     (PostActions { ( action_stmt)* })
327     (PassActions { ( action_stmt)* }) // end PassActions
328     (FailActions { ( action_stmt)* }) // end FailActions
329 } // end TestBase
330
331 =====
332 TestType TEST_TYPE_NAME {
333     // If not inheriting from a previously-defined TEST_TYPE_NAME (either vendor-defined or user-defined),
334     // a TestType will inherit from TestBase. The “Inherit TestBase” statement is not required, but is
335     // recommended. If no Inherit statement is present, the behavior is equivalent to “Inherit TestBase”
336     (Inherit TestBase; | Inherit TEST_TYPE_NAME ; )
337     (Parameters {
338         ((<In | Out | InOut>) var_type VAR_NAME;)*
339         ((<In | Out | InOut>) var_type VAR_NAME;){ initial_value_stmt })*
340     }) // end Parameters
341     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
342     (PreActions { (< action_stmt | bypass_stmt )* })
343     (TestExec; | FuncExec func_stmt | FuncExec { ( func_stmt )* }
344     | flownode_stmt+ ) // Allows inline instantiation of a flow in a Test. That flow can't be reused.
345     (PostActions { ( action_stmt)* })
346     (PassActions { ( action_stmt)* }) // end PassActions
347     (FailActions { ( action_stmt)* }) // end FailActions
348 } // end TestType
349
350 =====
351     (Tests { ( test_instance_stmt )* }) // Instantiate named Test(s) from TestType(s)
352
353 =====
354 FlowType FLOW_TYPE_NAME {
355     // If not inheriting from a previously-defined FLOW_TYPE_NAME (either vendor-defined or user-defined),
356     // a FlowType 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 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     (Title STRING ;)
366     (flownode_stmt)*
367     (PostActions { ( post_action_stmt)* })
368     (PassActions { ( action_stmt)* }) // end PassActions
369     (FailActions { ( action_stmt)* }) // end FailActions

```

```

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

```

```
426         // PreActions – no PreActions
427         // Flownodes – no flownodes specified in the type – will be provided at instantiation
428         // PostActions – no PostActions
429         // PassActions – no PassActions
430         // FailActions – use FailActions as defined in TestBase
431     } // end FlowType
432 } // end TestDefaults
433
```

For Multi-Site

```

434
435
436 // PROPOSED – How do we want to specify different test programs for different sites?
437 // Is this necessary?
438 ( SiteDefs (SITE_DEF_NAME) {
439
440     // Variables are global across all sites (one copy across all sites)
441     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
442
443     ( NumSites integer_expr ; )
444
445     // One statement per site. Only one can be default. If there are more sites than statements, then all
446     // unspecified sites will use the default program.
447     // It has been stated that we'd like to accommodate a scenario in which the same test program is
448     // running on all sites, but a different flow is running on each site. Using the syntax proposed here,
449     // it is possible do this by specifying different test programs for each site. The test programs are
450     // identical EXCEPT that the Flow specified by OnStart is different.
451     ( (Default) SITE_NAME TestProgram TEST_PROGRAM_NAME;)+
452 })
453
454 SPEC_NAME.NumCategories returns the number of categories in a spec block (as an integer)
455
456 SPEC_NAME.Category[I].Name returns the category name corresponding to the Ith category in a spec block
457 (first category is index 0)
458
459 SPEC_NAME.Category[I].NumVariables returns the number of variables in a given category in a spec
460 block
461
462 SPEC_NAME.Category[I].Variables[J].Name returns the variable name corresponding to the Jth variable in
463 the Ith category within a spec block.
464

```

```

465
466 STIL.4 mandatory requirements for TestBase
467 // The STIL.4-mandated contents for TestBase are shown below. If a user provides an alternate
468 // definition, that definition MUST include all the elements shown below. The purpose of
469 // TestBase is to provide a common set of elements for all TestType and FlowType definitions.
470 TestBase {
471   Parameters {
472     Out Const String TestID { InitialValue ""; } // Empty string
473     Out Integer execResult { InitialValue 0; } // Return value from execution
474                                           // execResult == 0 -> PASS, execResult != 0 -> FAIL
475                                           // 0 = PASS, non-zero = FAIL
476                                           // Define PASS = 0; Define FAIL = !PASS
477     In Bin failBin { InitialValue NoBin; } // For component values, see section 3.2.5 of Ernie's bin doc.
478   }
479   PreActions { } // No PreActions
480   PostActions { } // No PostActions
481   // Semantics: PassActions or FailActions are selected by the implied arbiter:
482   // if (execResult == PASS) { Do PassActions } Else { Do FailActions }
483   PassActions { } // No PassActions. Pass Actions are executed if execResult == PASS
484   FailActions { // Fail Actions are executed if execResult == FAIL
485     SetBin failBin; // Perform binning based on the current value of failBin.
486                     // If value of failBin is undefined, then no binning takes place
487                     // This is the equivalent to:
488                     // If (failBin != Undefined) { SetBin failBin; }
489     // Vet this against industrial practice (ASAP, SmarTest, OTPL, Envision, Stylus)
490   } // end FailActions
491 } // end TestBase
492

```

STIL.4 definition of default FlowNode

```

493
494 FlowNode {
495     PreActions { }
496     TestExec EXEC_OBJECT_NAME;
497     PostActions { }
498     ExitPorts {
499         Port 0 FAIL: EXEC_OBJECT_NAME.execResult == FAIL    { SetBin EXEC_OBJECT_NAME.failBin; Stop; }
500         Port 1 PASS: EXEC_OBJECT_NAME.execResult == PASS { Next; }
501     } // end ExitPorts
502 } // end FlowNode
503
504
505

```

STIL.4 definition of default FlowType

```

506
507
508 // The flowtype name stil_dot_4_default is arbitrary – but it MUST be defined by the
509 // standard, just as we’ve defined what TestBase will contain.
510 FlowType stil_dot_4_default {
511     // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
512     // Parameters – use TestBase Parameters
513     // Variables – no local variables
514     // PreActions – no PreActions
515     // Flownodes – no flownodes specified in the type – will be provided at instantiation
516     // PostActions – no PostActions
517     // PassActions – no PassActions
518     // FailActions – use FailActions as defined in TestBase
519 }
520
521

```

STIL.4 definition of TestType NoOpType and Test NoOp

```

522
523
524 In order to help demonstrate concepts in STIL.4, it’s necessary to define a TestType, and show how a Test
525 can be instantiated using this type. Therefore, we define an example TestType called NoOpType, and
526 instantiate a Test called NoOp from it. This test can be used to illustrate any of the flow concepts without
527 getting bogged down in the definition of the many actual TestTypes that might be needed on any specific
528 tester.
529

```

```

530 TestType NoOpType {
531     // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
532     // Parameters – use TestBase Parameters
533     // Variables – no local variables
534     // PreActions – no PreActions
535     // No TestExec. No operation performed. Or – use TestExec; (with no tokens).
536     // Test library must contain a Test named NoOp, which sets the
537     // value of NoOp.FailFlag to the appropriate value (Pass or Fail)
538     // PostActions – no PostActions
539     // PassActions – no PassActions
540     // FailActions – use FailActions as defined in TestBase
541 } // end TestType NoOp
542

```

```

543 Test NoOpType NoOp
544
545
546

```

```
547 Flow myflowtype myflowname;
548 Flow myflowtype myflowname2 { }
549 Flow myflowtype myflowname3 { }
550
551
552 Flows {
553     myflowtype myflowname;
554     myflowtype myflowname2 { }
555     myflowtype myflowname3 { }
556 }
```