

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> .

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

```

```

113         (VAR_NAME = [ expr (expr)+];)* // For arrays (?)
114     } // end func_stmt
115 >
116
117 test_elements_stmt =
118     < VAR_NAME = expr;
119     | VAR_NAME = [ expr (expr)+]; // For arrays (?)
120     | PreActions { ( < action_stmt | bypass_stmt )* } )
121     | FuncExec func_stmt | FuncExec { ( func_stmt )* }
122     | PostActions { ( action_stmt )* }
123     | PassActions { ( action_stmt )* }
124     | FailActions { ( action_stmt )* }
125 >
126
127 test_instance_stmt =
128     // create a named Test from a TestType, using the default elements for the TestType
129     TEST_TYPE TEST_INSTANCE_NAME;
130
131     // create a named Test from a TestType, overriding some or all of the default elements of the TestType.
132     // Any element specified in the instantiation (i.e., PreActions, FailActions) completely replaces that
133     // element as specified in the type definition.
134     | TEST_TYPE TEST_INSTANCE_NAME { ( test_elements_stmt )* }
135
136 flownode_stmt =
137     FlowNode (NODE_NAME) {
138         ( PreActions { ( < action_stmt | bypass_stmt )* } )
139         ( TestExec execute_stmt | TestExec { ( execute_stmt )* } )
140         ( PostActions { ( action_stmt )* } )
141         ( ExitPorts {
142             ( PORTLABEL: boolean_expr { ( exit_port_stmt )* } )*)
143         } ) // end ExitPorts
144     } // end FlowNode
145
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  test_execute_stmt =
170      < TEST_NAME; // execute a named Test
171      | TEST_TYPE; // create and execute a temporary inline Test (named _INLINE_TEST)
172                        // from TestType, using the type's default element value.
173
174      | TEST_TYPE { ( test_elements_stmt )* } > // Create and execute a temporary inline Test
175                        // (named _INLINE_TEST) from TestType,
176                        // overriding the type's default element values
177
178  flow_execute_stmt =
179      < FLOW_NAME; // execute a named Flow
180      | FLOW_TYPE; // create and execute a temporary inline Flow (named _INLINE_FLOW)
181                        // from FlowType, using the type's default element values
182      | FLOW_TYPE { ( flow_elements_stmt )* } > // Create and execute an a temporary inline Flow
183                        // (named _INLINE_FLOW) from FlowType,
184                        // overriding the type's default element values
185
186  execute_stmt =< test_execute_stmt | flow_execute_stmt >
187
188  var_elements_stmt =
189      // Do we need an Operator attribute? Intended to identify variables whose values are set by a
190      // query to an operator and the operator's response (i.e., lot ID)
191      < var_type VAR_NAME ;
192      | var_type VAR_NAME {
193          ( Length integer; ) // array of var_type HOW TO INDEX????
194          ( initial_value_stmt )
195      }
196
197  =====
198  STIL 1.0 {
199      ( Flow 2007; )+
200  }
201
202  =====
203  Variables ( VAR_DOMAIN ) { // extensions to 1450.1
204      var_elements_stmt*
205  }
206
207  =====
208  FunctionsDefs {
209      ( FUNCTION_NAME {
210          ( Parameters {
211              ( var_type VAR_NAME (< In | Out | InOut > ; ))*
212              ( var_type VAR_NAME (< In | Out | InOut > ) { initial_value_stmt } )*)
213          } ) // end Parameters
214      })* // end function_name
215  } // end FunctionDefs
216
217
218
219
220
221
222
223
224

```

```

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 { ( action_stmt)* } ) // end PassActions
323     (FailActions { ( action_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     (Inherit TestBase; | Inherit TEST_TYPE_NAME ; )
332     ( Parameters {
333         ((<In | Out | InOut>) var_type VAR_NAME;)*
334         ((<In | Out | InOut>) var_type VAR_NAME;) { initial_value_stmt })*
335     }) // end Parameters
336     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
337     (PreActions { ( < action_stmt | bypass_stmt )* } )
338     (TestExec; | FuncExec func_stmt | FuncExec { ( func_stmt )* }
339     | flownode_stmt+ ) // Allows inline instantiation of a flow in a Test. That flow can't be reused.
340     (PostActions { ( action_stmt)* } )
341     (PassActions { ( action_stmt)* } ) // end PassActions
342     (FailActions { ( action_stmt)* } ) // end FailActions
343 } // end TestType
344
345 =====
346     ( Tests { ( test_instance_stmt )* } ) // Instantiate named Test(s) from TestType(s)
347
348 =====
349 FlowType FLOW_TYPE_NAME {
350     // If not inheriting from a previously-defined FLOW_TYPE_NAME (either vendor-defined or user-defined),
351     // a FlowType will inherit from TestBase. The "Inherit TestBase" statement is not required, but is
352     // recommended. If no Inherit statement is present, the behavior is equivalent to "Inherit TestBase"
353     (Inherit TestBase; | Inherit FLOW_TYPE_NAME ; )
354     ( Parameters {
355         ((<In | Out | InOut>) var_type VAR_NAME;)*
356         ((<In | Out | InOut>) var_type VAR_NAME;) { initial_value_stmt })*
357     }) // end Parameters
358     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
359     (PreActions { ( < action_stmt | bypass_stmt )* } )
360     (Title STRING ; )
361     ( flownode_stmt )*
362     (PostActions { ( post_action_stmt)* } )
363     (PassActions { ( action_stmt)* } ) // end PassActions
364     (FailActions { ( action_stmt)* } ) // end FailActions
365 } // end FlowType
366
367 =====
368     ( Flows { ( flow_instance_stmt )* } ) // Instantiate named Flow(s) from FlowType(s)

```

```

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

```

```
425         // FailActions – use FailActions as defined in TestBase
426     } // end FlowType
427 } // end TestDefaults
428
```

For Multi-Site

```

429
430
431 // PROPOSED – How do we want to specify different test programs for different sites?
432 // Is this necessary?
433 ( SiteDefs (SITE_DEF_NAME) {
434
435     // Variables are global across all sites (one copy across all sites)
436     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
437
438     ( NumSites integer_expr ; )
439
440     // One statement per site. Only one can be default. If there are more sites than statements, then all
441     // unspecified sites will use the default program.
442     // It has been stated that we'd like to accommodate a scenario in which the same test program is
443     // running on all sites, but a different flow is running on each site. Using the syntax proposed here,
444     // it is possible do this by specifying different test programs for each site. The test programs are
445     // identical EXCEPT that the Flow specified by OnStart is different.
446     ( (Default) SITE_NAME TestProgram TEST_PROGRAM_NAME;)+
447 })
448
449 SPEC_NAME.NumCategories returns the number of categories in a spec block (as an integer)
450
451 SPEC_NAME.Category[I].Name returns the category name corresponding to the Ith category in a spec block
452 (first category is index 0)
453
454 SPEC_NAME.Category[I].NumVariables returns the number of variables in a given category in a spec
455 block
456
457 SPEC_NAME.Category[I].Variables[J].Name returns the variable name corresponding to the Jth variable in
458 the Ith category within a spec block.
459

```

```

460
461 STIL.4 mandatory requirements for TestBase
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 TestBase {
466   Parameters {
467     Out Const String TestID { InitialValue ""; } // Empty string
468     Out Integer execResult { InitialValue 0; } // Return value from execution
469                                           // execResult == 0 -> PASS, execResult != 0 -> FAIL
470                                           // 0 = PASS, non-zero = FAIL
471                                           // Define PASS = 0; Define FAIL = !PASS
472     In Bin failBin; // Initial Value = undefined – equivalent to no bin.
473   }
474   PreActions { } // No PreActions
475   PostActions { } // No PostActions
476   // Semantics: PassActions or FailActions are selected by the implied arbiter:
477   // if (execResult == PASS) { Do PassActions} Else { Do FailActions }
478   PassActions { } // No PassActions. Pass Actions are executed if execResult == PASS
479   FailActions { // Fail Actions are executed if execResult == FAIL
480     SetBin FailBin; // Perform binning based on the current value of FailBin.
481                     // If value of FailBin is undefined, then no binning takes place
482                     // This is the equivalent to:
483                     // If (FailBin != Undefined) { SetBin FailBin; }
484     // Vet this against industrial practice (ASAP, SmarTest, OTPL, Envision, Stylus)
485   } // end FailActions
486 } // end TestBase
487

```

488
489
490
491
492
493
494
495
496
497
498
499
500

STIL.4 definition of default FlowNode

```
FlowNode {
  PreActions { }
  TestExec <exec_object_name>;
  PostActions { }
  ExitPorts {
    Port 0 FAIL: <exec_object_name>.execResult == FAIL { SetBin <exec_object_name>.FailBin; Stop; }
    Port 1 PASS: <exec_object_name>.execResult == PASS { Next; }
  } // end ExitPorts
} // end FlowNode
```

501
502

STIL.4 definition of default FlowType

503 // The flowtype name **stil_dot_4_default** is arbitrary – but it MUST be defined by the
504 // standard, just as we’ve defined what TestBase will contain.

```
FlowType stil_dot_4_default {
  // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
  // Parameters – use TestBase Parameters
  // Variables – no local variables
  // PreActions – no PreActions
  // Flownodes – no flownodes specified in the type – will be provided at instantiation
  // PostActions – no PostActions
  // PassActions – no PassActions
  // FailActions – use FailActions as defined in TestBase
}
514 }
515
516
```

517
518

STIL.4 definition of TestType NoOpType and Test NoOp

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

524
525
526
527
528
529
530
531
532
533
534
535
536
537

```
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.FailFlag to the appropriate value (Pass or Fail)
  // PostActions – no PostActions
  // PassActions – no PassActions
  // FailActions – use FailActions as defined in TestBase
} // end TestType NoOp
```

538
539
540
541

```
Test NoOpType NoOp
```

```
542 Flow myflowtype myflowname;
543 Flow myflowtype myflowname2 { }
544 Flow myflowtype myflowname3 { }
545
546
547 Flows {
548     myflowtype myflowname;
549     myflowtype myflowname2 { }
550     myflowtype myflowname3 { }
551 }
```