

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.

### 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
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      | Bypass (GoTo < PORTLABEL | VAR_NAME > ( SkipActions ) );
100
101  bypass_stmt =
102      < bypass_actions | If boolean_expr { bypass_actions } >
103
104  exit_port_stmt =
105      < action_stmt
106      | ReturnState integer_expr; // default to last TestExec return value
107      | Next (FLOW_NODE_NAME); // only allowed in FlowNode
108      | Return (Retest (integer_expr) ); // terminate the Test or Flow; return to caller. Repeat test specified
109      // number of times upon fail. If Retest count is missing, use count = 1
110      | Stop; // terminate the initiating On* condition
111      | Reset; // terminate the initiating On* condition; generate an OnReset event
112      >

```

```

113
114 func_stmt =
115     < FUNC_NAME ; // name of a Function from FunctionDefs block
116     | FUNC_NAME { // name of a Function from FunctionDefs block
117         (VAR_NAME = expr)*
118         (VAR_NAME = [ expr (expr)+];)* // For arrays (?)
119     } // end func_stmt
120 >
121
122 test_elements_stmt =
123     < VAR_NAME = expr;
124     | VAR_NAME = [ expr (expr)+]; // For arrays (?)
125     | PreActions { ( < action_stmt | bypass_stmt)* } )
126     | FuncExec func_stmt | FuncExec { ( func_stmt )* }
127     | PostActions { ( action_stmt)* }
128     | PassActions { ( exit_port_stmt)* }
129     | FailActions { ( exit_port_stmt)* }
130 >
131
132 test_instance_stmt =
133     // create a named Test from a TestType, using the default elements for the TestType
134     TEST_TYPE TEST_INSTANCE_NAME;
135
136     // create a named Test from a TestType, overriding some or all of the default elements of the TestType.
137     // Any element specified in the instantiation (i.e., PreActions, FailActions) completely replaces that
138     // element as specified in the type definition.
139     | TEST_TYPE TEST_INSTANCE_NAME { ( test_elements_stmt )* }
140
141 flownode_stmt =
142     FlowNode (NODE_NAME) {
143         ( PreActions { ( < action_stmt | bypass_stmt)* } )
144         ( TestExec execute_stmt | TestExec { ( execute_stmt )* } )
145         ( PostActions { ( action_stmt)* } )
146
147         ( ExitPorts {
148             ( PORTLABEL: boolean_expr { ( exit_port_stmt)* } )*)
149         } ) // end ExitPorts
150
151     } // end FlowNode
152
153 flow_elements_stmt =
154     < VAR_NAME = expr;
155     | VAR_NAME = [ expr (expr)+]; // For arrays (?)
156     | PreActions { ( < action_stmt | bypass_stmt)* } )
157     | Title STRING ;
158     | ( flownode_stmt )*
159     | PostActions { ( action_stmt)* } )
160     | PassActions { ( exit_port_stmt)* } // end PassActions
161     | FailActions { ( exit_port_stmt)* } // end FailActions
162 >
163
164
165
166
167
168

```

```

169 flow_instance_stmt =
170     // create a named Flow from a FlowType, using the default elements for the FlowType
171     < (FLOW_TYPE) FLOW_INSTANCE_NAME ; // if FLOW_TYPE is not specified, then the
172     // STIL.4 default FLOW_TYPE is used
173
174     // create a named Flow from a FlowType, overriding some or all of the default elements of the FlowType.
175     // Any element specified in the instantiation (i.e., PreActions, flownodes, PassActions) completely
176     // replaces that element as specified in the type definition.
177     | (FLOW_TYPE) FLOW_INSTANCE_NAME { ( flow_elements_stmt )* } > // if FLOW_TYPE is not specified, then the
178     // STIL.4 default FLOW_TYPE is used
179
180 test_execute_stmt =
181     < TEST_NAME ; // execute a named Test
182     | TEST_TYPE ; // create and execute a temporary inline Test (named _INLINE_TEST)
183     // from TestType, using the type's default element value.
184
185     | TEST_TYPE { ( test_elements_stmt )* } > // Create and execute a temporary inline Test
186     // (named _INLINE_TEST) from TestType,
187     // overriding the type's default element values
188
189 flow_execute_stmt =
190     < FLOW_NAME ; // execute a named Flow
191     | FLOW_TYPE ; // create and execute a temporary inline Flow (named _INLINE_FLOW)
192     // from FlowType, using the type's default element values
193     | FLOW_TYPE { ( flow_elements_stmt )* } > // Create and execute an a temporary inline Flow
194     // (named _INLINE_FLOW) from FlowType,
195     // overriding the type's default element values
196
197 execute_stmt = < test_execute_stmt | flow_execute_stmt >
198
199 var_elements_stmt =
200     // Do we need an Operator attribute? Intended to identify variables whose values are set by a
201     // query to an operator and the operator's response (i.e., lot ID)
202     < var_type VAR_NAME ;
203     | var_type VAR_NAME {
204         ( Length integer; ) // array of var_type HOW TO INDEX????
205         ( initial_value_stmt )
206     }
207
208 =====
209 STIL 1.0 {
210     ( Flow 2007; )+
211 }
212 =====
213 Variables ( VAR_DOMAIN ) { // extensions to 1450.1
214     var_elements_stmt*
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 -----
269 Bin Property access syntax
270 counter_reset_event =
271 <
272   OnLoad |
273   OnLotStart |
274   OnRetest |
275   OnSiteStart | // What’s the difference between OnSiteStart
276   OnStart | // and OnStart ?
277   OnWaferStart
278 >
279 bin_property =
280 <

```

```

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

```

```

311 =====
312 TestBase {
313     (Parameters {
314         ((<In | Out | InOut>) var_type VAR_NAME;)*
315         ((<In | Out | InOut>) var_type VAR_NAME;){ initial_value_stmt })*
316     }) // end Parameters
317     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
318     (PreActions { (< action_stmt | bypass_stmt )* })
319     (PostActions { ( action_stmt)* })
320     (PassActions { ( exit_port_stmt)* }) // end PassActions
321     (FailActions { ( exit_port_stmt)* }) // end FailActions
322 } // end TestBase
323
324 =====
325 TestType TEST_TYPE_NAME {
326     // If not inheriting from a previously-defined TEST_TYPE_NAME (either vendor-defined or user-defined),
327     // a TestType will inherit from TestBase. The “Inherit TestBase” statement is not required, but is
328     // recommended. If no Inherit statement is present, the behavior is equivalent to “Inherit TestBase”
329     (Inherit TestBase; | Inherit TEST_TYPE_NAME ; )
330     (Parameters {
331         ((<In | Out | InOut>) var_type VAR_NAME;)*
332         ((<In | Out | InOut>) var_type VAR_NAME;){ initial_value_stmt })*
333     }) // end Parameters
334     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
335     (PreActions { (< action_stmt | bypass_stmt )* })
336     (TestExec; | FuncExec func_stmt | FuncExec { ( func_stmt )* }
337     | flownode_stmt+ ) // Allows inline instantiation of a flow in a Test. That flow can't be reused.
338     (PostActions { ( action_stmt)* })
339     (PassActions { ( exit_port_stmt)* }) // end PassActions
340     (FailActions { ( exit_port_stmt)* }) // end FailActions
341 } // end TestType
342
343 =====
344     (Tests { ( test_instance_stmt )* }) // Instantiate named Test(s) from TestType(s)
345
346 =====
347 FlowType FLOW_TYPE_NAME {
348     // If not inheriting from a previously-defined FLOW_TYPE_NAME (either vendor-defined or user-defined),
349     // a FlowType will inherit from TestBase. The “Inherit TestBase” statement is not required, but is
350     // recommended. If no Inherit statement is present, the behavior is equivalent to “Inherit TestBase”
351     (Inherit TestBase; | Inherit FLOW_TYPE_NAME ; )
352     (Parameters {
353         ((<In | Out | InOut>) var_type VAR_NAME;)*
354         ((<In | Out | InOut>) var_type VAR_NAME;){ initial_value_stmt })*
355     }) // end Parameters
356     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
357     (PreActions { (< action_stmt | bypass_stmt )* })
358     (Title STRING ;)
359     (flownode_stmt)*
360     (PostActions { ( post_action_stmt)* })
361     (PassActions { ( exit_port_stmt)* }) // end PassActions
362     (FailActions { ( exit_port_stmt)* }) // end FailActions
363 } // end FlowType
364
365 =====
366     (Flows { ( flow_instance_stmt )* }) // Instantiate named Flow(s) from FlowType(s)

```

```

367
368 =====
369 TestProgram TEST_PROGRAM_NAME {
370     DUTType "DUT NAME STRING";
371     SocketDef "SOCKET NAME STRING";
372     // Variables are global to test program and below; local to site (a copy for each site)
373     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
374     (BinDefs BIN_DEF_NAME; )
375     (BinMap BIN_MAP_NAME; )
376
377     EntryPoints {
378         (OnException execute_stmt)
379         (OnFinish execute_stmt )
380         (OnLoad execute_stmt )
381         (OnLotEnd execute_stmt )
382         (OnLotStart execute_stmt )
383         (OnMultiSiteDisable execute_stmt )
384         (OnMultiSiteEnable execute_stmt )
385         (OnPatternLoad execute_stmt )
386         (OnPowerDown execute_stmt )
387         (OnReset execute_stmt )
388         (OnSiteEnd execute_stmt )
389         (OnSiteStart execute_stmt )
390         (OnStart execute_stmt )
391         (OnUnload execute_stmt )
392         (OnWaferEnd execute_stmt )
393         (OnWaferStart execute_stmt )
394     } // End Entry Points
395 } // end TestProgram
396
397 =====
398 // The TestDefaults block is a container within which the default definitions for TestBase, FlowType, and
399 // FlowNode
400 TestDefaults {
401     // The STIL.4-mandated contents for TestBase are shown below. If a user provides an alternate definition,
402     // that definition MUST include all the elements shown below. The purpose of TestBase is to provide a
403     // common set of elements for all TestType and FlowType definitions.
404     TestBase {
405         // See "STIL.4 mandatory requirements for TestBase" below
406     } // end TestBase
407
408     // STIL.4 default FlowNode definition.
409     FlowNode {
410         // See "STIL.4 definition of default FlowNode" below
411     } // end FlowNode
412
413     // The flowtype name stil_dot_4_default is arbitrary – but it MUST be defined by the
414     // standard, just as we've defined what TestBase will contain.
415     FlowType stil_dot_4_default {
416         // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
417         // Parameters – use TestBase Parameters
418         // Variables – no local variables
419         // PreActions – no PreActions
420         // Flownodes – no flownodes specified in the type – will be provided at instantiation
421         // PostActions – no PostActions
422         // PassActions – no PassActions

```

```
423         // FailActions – use FailActions as defined in TestBase
424     } // end FlowType
425 } // end TestDefaults
426
```

## For Multi-Site

```

427
428
429 // PROPOSED – How do we want to specify different test programs for different sites?
430 // Is this necessary?
431 ( SiteDefs (SITE_DEF_NAME) {
432
433     // Variables are global across all sites (one copy across all sites)
434     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
435
436     ( NumSites integer_expr ; )
437
438     // One statement per site. Only one can be default. If there are more sites than statements, then all
439     // unspecified sites will use the default program.
440     // It has been stated that we'd like to accommodate a scenario in which the same test program is
441     // running on all sites, but a different flow is running on each site. Using the syntax proposed here,
442     // it is possible do this by specifying different test programs for each site. The test programs are
443     // identical EXCEPT that the Flow specified by OnStart is different.
444     ( (Default) SITE_NAME TestProgram TEST_PROGRAM_NAME;)+
445 })
446
447 SPEC_NAME.NumCategories returns the number of categories in a spec block (as an integer)
448
449 SPEC_NAME.Category[I].Name returns the category name corresponding to the Ith category in a spec block
450 (first category is index 0)
451
452 SPEC_NAME.Category[I].NumVariables returns the number of variables in a given category in a spec
453 block
454
455 SPEC_NAME.Category[I].Variables[J].Name returns the variable name corresponding to the Jth variable in
456 the Ith category within a spec block.
457

```

```

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

```

**STIL.4 definition of default FlowNode**

```

488
489 FlowNode {
490     PreActions { }
491     TestExec <exec_object_name>;
492     PostActions { }
493     ExitPorts {
494         Port 0 FAIL: <exec_object_name>.execResult == FAIL    { SetBin <exec_object_name>.FailBin; Stop; }
495         Port 1 PASS: <exec_object_name>.execResult == PASS { Next; }
496     } // end ExitPorts
497 } // end FlowNode
498
499
500

```

**STIL.4 definition of default FlowType**

```

501
502
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.
505 FlowType stil_dot_4_default {
506     // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
507     // Parameters – use TestBase Parameters
508     // Variables – no local variables
509     // PreActions – no PreActions
510     // Flownodes – no flownodes specified in the type – will be provided at instantiation
511     // PostActions – no PostActions
512     // PassActions – no PassActions
513     // FailActions – use FailActions as defined in TestBase
514 }
515
516

```

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

517  
518  
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 TestType NoOpType {
526     // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
527     // Parameters – use TestBase Parameters
528     // Variables – no local variables
529     // PreActions – no PreActions
530     // No TestExec. No operation performed. Or – use TestExec; (with no tokens).
531     // Test library must contain a Test named NoOp, which sets the
532     // value of NoOp.FailFlag to the appropriate value (Pass or Fail)
533     // PostActions – no PostActions
534     // PassActions – no PassActions
535     // FailActions – use FailActions as defined in TestBase
536 } // end TestType NoOp
537
538 Test NoOpType NoOp
539
540
541

```

```
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 }
```