

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 TestBase”. 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> statement definition. 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, replaced <code><exec_object_name></code> with <code>EXEC_OBJECT_NAME</code> to adhere to the <code>STIL BNF</code> 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> ?
8/14/08	D0.23	jo	<ul style="list-style-type: none"> In <code>initial_value_stmt</code>, updated syntax so that <code>test_element_stmt</code> and <code>flow_element_stmt</code> occur after <code>InitialValue</code> keyword, to bring <code>TestType</code> and <code>FlowType</code> initial value statements in line with those of all other types. Note the inclusion of open/close braces (<code>{ }</code>) to enclose those statements. Added keyword Port preceding <code>PORTLABEL</code> in <code>ExitPorts</code> statement; removed colon (<code>:</code>) after <code>PORTLABEL</code> in <code>exit_port_stmt</code>. Addition of <code>Port</code> keyword makes the colon unnecessary.

			<ul style="list-style-type: none"> • Modified InitialValue syntax (for variables) to include array initialization (using a comma-separated list). • Added keyword Return <i>integer_expr</i> to <i>exit_port_actions</i>. This causes an immediate return to the caller (either a Test or a Flow). The integer return value sets the FlowNode variable Result. Result is a predefined keyword which can be used only in the FlowNode PostActions or ExitPort clauses. • Updated semantics of Stop statement. Stop now returns immediately to EntryPoint initiator, with appropriate pass/fail result. See text for more details. • Added the option (<i>flownode_stmt</i>)* to <i>test_elements_stmt</i>. This will allow tests to specify a sequence of flownodes for each instance created, as well as in the type definition. • Allow a TestType to derive from a FlowType. This is to allow Flows to be turned into Tests (and thus, presumably, hiding the internals of that flow from graphical Flow Editor tools). • Removed TestDefaults block. Definitions of TestBase and the default flow type stil_dot_4_default need no container; the STIL.4 default flownode definition must occur outside the scope of a Flow or Test. If more than one default definition occurs, the last one listed is used. • Modified the syntax of FlowNode to include the SetResult statement, the introduction of the FlowNode variable Result, and the keyword CurrentExec, which is an alias for the actual Test or Flow named in the TestExec statement. CurrentExec can be used ONLY in the SetResult, the <i>exit_port_expr</i>, or <i>exit_port_stmt</i> statements. • Change STIL 1.0 Flow extension date from 2007 to 2008. • Added angle brackets (< >) to syntax notation for <i>softbin_definition</i>. Expanded SetBin and SetBinStop syntax to include BIN_VAR_NAME, as well as explicit bin name. Added <i>bin_expr</i> notation. • Modified FlowType syntax to allow FLOW_TYPE_NAME to be optional. This allows an unnamed FlowType block which can be used as a default FlowType. See section 6.9 of IEEE_1450_1999_0 (dot0) for additional information on domain names. • Updated STIL.4 default definitions of TestBase, FlowNode, and FlowType. • Change Test and Flow instantiation syntax from Flows { } / Tests { } to Flow FLOWTYPE FLOWNAME; / Test TEST_TYPE TEST_NAME; • Added clarifications about integer values for Pass, Fail, True, and False.
9/04/08	D0.24	jo	<ul style="list-style-type: none"> • Removed SetReturn statement and FlowNode variable Result (per discussion at WG meeting of 09/04/08). • Clarified semantics of Stop action statement. • Added Exit and Return statements. Clarified semantics of those statements. • Updated default FlowNode definition so that FlowNode FAIL

			<p>ExitPort actions sets the execResult of the Test or Flow containing the FlowNode.</p> <ul style="list-style-type: none"> • Updated semantics of default FlowNode. Not required, but if not provided by the user, then substitution of TestExec statements for FlowNodes is NOT allowed. • Updated TestBase definition. • Changed spelling of TestBase mandatory fields from execResult to ExecResult and failBin to FailBin. • Added NullBlock keyword to <i>initial_val_elements</i> metatype. Used for setting <i>stil_block_type</i> Variables or Parameters to a Null block. For TestType or FlowType parameters, this is treated as an optional parameter (the user can, but is not required to, provide a valid override value). If the user does NOT provide a valid override value, the parameter will not be used. If the user does provide a valid overrided value, the parameter will be used.
9/18/08	D0.25	jo	<ul style="list-style-type: none"> • Added semantics regarding user definition of TestBase and default FlowNode (each can be defined or redefined by the user any number of times UNTIL first use. After first use, redefinition is NOT allowed). • Added MultiSiteSerial field to TestBase. Normally initialized to False, meaning that when this test is used in a flows which are running on multiple sites, all are executed in parallel. If set to True, then all sites executes this test (and its containing FlowNode) serially (one after the other; order of execution of sites is not specified). • Updated semantics for Next <i>exit_port_stmt</i>. For the last FlowNode in a sequence, Next; is the same as Return;. This does NOT extend to the “Next FLOW_NODE_NAME;” form, however.
10/02/08	D0.26	jo	<ul style="list-style-type: none"> • Removed Test and Flow from <i>stil_block_type</i>; removed <i>test_elements_stmt</i> and <i>flow_elements_stmt</i> from <i>initial_value_elements</i> metatype. • Modified Parameter syntax for FunctionDefs so that direction (In, Out, InOut) is listed first, prior to the type and name definition, rather than after the type and name definition. This syntax now mirrors that of TestBase, TestType, and FlowType. • Restructured BNF for <i>initial_value_stmt</i> to be more concise. • Restructured BNF for Parameter blocks to use <i>var_elements_stmt</i>. • Updated <i>var_elements_stmt</i> syntax used for specifying arrays, and for initializing scalar or array variables. Changed metatype name for initial value elements from <i>initial_value_elements</i> to <i>initial_value_element</i>. • Added keyword None to <i>initial_value_element</i>. Indicates an optional parameter when used in Parameter blocks of TestTypes, Tests, FlowTypes, or Flows (i.e., does not need to be specified by the user, since it has an initial value, but is ignored by runtime code). Keyword None is not allowed for initial value assignments in Variables blocks.

10/09/08	D0.26	jo	<ul style="list-style-type: none"> • Modified <code>flow_node_stmt</code> to include bare <code>TestExec</code>, which implies that the default <code>FlowNode</code> is used around the <code>TestExec</code>. • Update <code>TestBase</code> default definition to use new “initial value” syntax
10/16/08	D0.26	jo	<ul style="list-style-type: none"> • Removed multiple <code>execute_stmt</code> form of <code>TestExec</code> from <code>flownode_stmt</code> (TestExec { (<code>execute_stmt</code>)* }). This capability was determined to be redundant, and therefore not needed. Changed TestExec NullExec; statement to TestExec; (<code>NullExec</code> keyword added no additional value). • Removed Title <code>STRING</code>; statement from <code>flow_elements_stmt</code> and <code>FlowType</code>. Again, not needed, since each object will have its own name (ID), and if a different title is needed, parameters or variables can serve this purpose.
03/04/09	D0.27	jo	<ul style="list-style-type: none"> • Added semantics description for <code>func_exec</code> call. • Updated <code>bin_properties</code> per Ernie’s latest binning document. <ul style="list-style-type: none"> ○ Removed <code>counter_reset_event</code> from <code>isSet</code> property (not needed). ○ Removed <code>ContinueOnFail</code> bin property. Deferred to phase 2. ○ Added <code>IsFailBin</code> property. • Added variable attributes per discussions on 11/20/08 and 12/04/08. Added .Size operator for arrays. • Added property queries for <code>Specs/Categories</code> and <code>Bin/BinAxis</code> hierarchies. • Added IncludeOnce keyword and semantics. • Added scoping rules for variables, specs, etc.
03/11/09	D0.27	jo	<ul style="list-style-type: none"> • Fixed typo in <code>IncludeOnce</code> statement (reference to <code>MSoft #pragma once</code>, instead of <code>IncludeOnce</code>) • In <code>Spec/Category</code> hierarchy access syntax, make <code>SPEC_NAME</code> optional. • Removed <code>OnSiteStart/OnSiteEnd</code> <code>EntryPoint</code> keywords. Can’t figure out how these might differ from <code>OnStart</code> (do we need an <code>OnEnd</code> – or maybe <code>OnTestStart/OnTestEnd</code>?).
03/19/09	D0.27	jo	<ul style="list-style-type: none"> • In <code>Spec/Category</code> hierarchy access syntax, update access syntax and semantics. Will need to do the same for binning.
05/20/09	D0.27	jo	<ul style="list-style-type: none"> • Updated <code>binmap</code> syntax per latest discussions and Ernie’s binning document (see lines 325-350). • Updated access syntax (lines 360-414) for groups (<code>Pass Fail</code>), <code>Axes</code>, and <code>Bins</code> per latest discussions and Ernie’s binning document. • Changed all references to “Unsigned (integer)” to <code>Integer</code>, specifying value range limitation <code>>=0</code>
05/26/09	D0.27	jo	<ul style="list-style-type: none"> • Changed <code>bindefs</code> syntax to require at least one bin per group (* was changed to + at lines 312, 314, 319, 321)

06/16/09	D0.27	jo	<ul style="list-style-type: none"> • Updated Bindefs/Binmap definitions and usage per recent WG discussions. Changes are between lines <ul style="list-style-type: none"> ○ In TestProgram block, replaced Bindefs BIN_DEF_NAME with SoftBinDefs BIN_DEF_NAME/HardBinDefs BIN_DEF_NAME (lines 544, 545). ○ Update bin access syntax to include BIN_NAME or BIN_NUMBER, as well as the full hierarchical syntax. Changes to binning are between lines 295 and 448. ○ Add ClearBin action to <i>actions</i> metatype (line 57-85). Added keyword All to both SetBin and ClearBin actions, to support setting or clearing all bins in all axes of all groups of selected SoftBin definition. Setting of hard bins (only one axis allowed) is done at end of test, at bin mapping time.
06/23/09	D0.27	jo	<ul style="list-style-type: none"> • Corrected keyword HardBin to HardBins (line 545) • Corrected syntax specifications at lines 414, 445/446 (changed integer to either AXIS_INDEX or BIN_INDEX, as appropriate)
07/08/09	D0.27	jo	<ul style="list-style-type: none"> • In the STIL 1.0 statement, change Flow 2008 to Flow 2009 (sigh) • Move selection of SoftBins and HardBins blocks from TestProgram block to BinMap. In TestProgram block, allow specification of either a BinMap (with its associated SoftBins and HardBins blocks), or of only a SoftBin block (if you only want to bin with soft bins for record-keeping purposes, and don't need the hard bins for handling equipment). • Removed specification of BINDEFS_NAME from access syntax for groups (line 406), axes (line 418), and bins (lines 451/452). The currently bindef as specified by the TestProgram block will be used to resolve any names specified by the group/axis/bins properties access syntax. • Remove HighestSetBin/LowestSetBin axis properties (lines 412-415). Replace with MapHighestBin/MapLowestBin statements in Axis definition, and MapType axis property. If multiple bins are set on an axis, then the MapHighestBin or MapLowestBin will be used to determine which bin to use in mapping softbins to hardbins. If neither is specified, then MapLowestBin will be used as a default. The MapType axis property will return either MapHighestBin or MapLowestBin (NOTE: this property is optional – we could drop it if we don't think it will be needed!) • Change EntryPoints keywords from On<something> to On <something>. This allows for the use of user-defined keywords as (non-portable) entry points.

3 Syntax summary for 1450.4

```

4   stil_block_type =
5       < Category
6         | DCLevels
7         | DCSequence
8         | DCSets
9         | Environment
10        | MacroDefs
11        | Pattern
12        | PatternBurst
13        | PatternExec
14        | Procedures
15        | ScanStructures
16        | Selector
17        | SignalGroups
18        | Signals
19        | Timing >
20
21   real_var_type =
22       < Capacitance // F (Farads)
23         | Compound // combinations of types
24         | Current // A (Amperes)
25         | Double
26         | Frequency // Hz (Herz)
27         | Gain // db (Decibels)
28         | Inductance // H (Henries)
29         | Length // m (Meters)
30         | Power // W (Watts)
31         | Real
32         | Resistance // Ohm (Ohms)
33         | Temperature // Cel (Degrees Celsius)
34         | Time // s (Seconds)
35         | Voltage // V (Volts) >
36
37   var_type =
38       // Boolean values: True/False or Pass/Fail
39       (Const) < Boolean | Integer | SignalRef | SignalVariable | String | real_var_type | stil_block_type >
40
41   initial_value_element =
42       < integer_expr // allow if var of type integer_expr
43         | sig_ref_expr // allow if var of type sig_ref_expr
44         | < True | False > // allow if var of type Boolean – from dot1, 0 = TRUE, 1 = FALSE.
45         | string // allow if var of type String
46         | wfc_string // allow if var of type SignalVariable
47         | real_expr // allow if var of type real_var_type
48         | BLOCK_NAME // allow if var of type stil_block_type - assign from predefined block
49         | None // allow for all types in Parameter blocks ONLY. Not allowed for
50           // initial values in variables blocks. If parameter is initialized to None
51           // and not overridden to a value other than None during instantiation,
52           // it is ignored during execution.
53       >
54
55   initial_value_list = initial_value_element \ initial_value_list, initial_value_element
56

```

```

57  action =
58      < VAR_NAME = expr; // Scalar variables
59      | VAR_NAME[integer_expr] = expr; // Array variables
60
61      // In the following SetBin, ClearBin, or SetBinStop statements, the bin must be selected from the SoftBin
62      // definitions. Mapping of the various soft bins to hard bins is done at the end of test at bin mapping time.
63      // All keyword in place of soft_bin_expr will set (or clear) all bins on all axes of all groups of selected
64      // SoftBin definition (to support disqualify binning and other such binning strategies). Setting of hard bins
65      // (only one axis allowed) is done at end of test at bin mapping time.
66      | SetBin (soft_bin_expr | All);
67      | ClearBin (soft_bin_expr | All);
68
69      // Return to the caller, "bubbling up" through the levels of FlowNodes, Flows and Tests, until the
70      // initiating EntryPoint is reached. At each level, execute the PostActions and PassActions or
71      // FailActions for Tests and Flows, but SKIP execution of FlowNodes (including any FlowNode
72      // PostActions or ExitPort actions).
73      | Stop; // terminate the initiating On* condition.
74
75      // SetBinStop is equivalent, semantically, to:
76      If (<bin_name> != NoBin) { SetBin <bin_name>; Stop;} Else { // No Action }
77      | SetBinStop soft_bin_expr;
78
79      // Exit – Immediately stop execution of current block (test, flow, or flow node), and jump directly to the
80      // initiating EntryPoint (On* condition) for end-of-test processing. The pass/fail result returned to the
81      // EntryPoint initiator (which would normally be the ExecResult of the Test or Flow specified by the
82      // EntryPoint), is specified by the integer_expr token of the "Exit <integer_expr>;" statement. This
83      // statement is generally intended to be used for exceptions, such as hardware failures, which require
84      // immediate termination of test execution.
85      | Exit integer_expr; // terminate the initiating On* condition.
86
87  action_stmt =
88      < If boolean_expr | Else If boolean_expr | Else { // Usual rules for If/Else If/Else apply
89          ( action )*
90      }
91      | ( action )* >
92
93  bypass_actions =
94      <
95      // For FlowNodes, Flows, and Tests
96      // Skip only Test execution or FlowNode sequence execution, resume at entry to PostActions Block
97      // Normal processing continues from there. Execute PostActions, and PassActions/FailActions (as
98      // determined by execResult of Test or Flow), or ExitPort selection Actions (as determined by flownode
99      // exit port selector). Selection of Pass/Fail path (for Tests and Flows) is controlled by execResult (which
100     // can be forced to a desired state prior to Bypass statement). Selection of ExitPort path (for flownodes)
101     // is also controlled by boolean expressions – typically, the return status of the test executed by the
102     // FlowNode. This return state (or value of any other boolean expression) can be forced to a desired state
103     // prior to the Bypass statement.
104     Bypass;
105
106     // For Flows and Tests only.
107     // Skip Test and PostActions, resume execution at entry to PassActions/FailActions.
108     // If Pass or Fail specified, follow that path. Otherwise, VAR_NAME is a string variable which specifies
109     // either PASS or FAIL. If using VAR_NAME, and it's an empty string, no bypass action occurs
110     // If SkipActions specified, don't execute PassActions/FailActions (equivalent to a return,
111     // since there's no branching from a Test or Flow)
112     | Bypass (GoTo <Pass | Fail | VAR_NAME > ( SkipActions ) );

```

```

113
114 // For FlowNodes only.
115 // Skip Test and PostActions, resume execution at entry to specified ExitPort.
116 // If SkipActions specified, don't execute don't execute the exit port actions.
117 // PORTLABEL is a string specifying a port label. VAR_NAME is either a string variable or an integer variable.
118 // If a string variable, it specifies the bypass exit port by label. If using VAR_NAME, and it's an empty string,
119 // no bypass action occurs. If an integer variable, it specifies the bypass exit port by index (based on the
120 // ordinal order of the exit ports, from top of list to bottom, with the first index being 0.
121 | Bypass (GoTo < PORTLABEL | VAR_NAME > ( SkipActions ) );
122
123 bypass_stmt =
124   < bypass_actions | If boolean_expr { bypass_actions } >
125
126 exit_port_stmt =
127   < action_stmt
128     | Next (FLOW_NODE_NAME); // For last FlowNode in a sequence, "Next;" == "Return;"
129     | Return; // Stop execution of FlowNodes. Return to execution chain of containing Test or
130               // Flow (PostActions, PassActions, or FailActions). ExecResult of containing Test
131               // or Flow can be set either by the FlowNode PostActions or ExitPort actions
132               // (typically, based on the results of the most recent TestExec object), or by the
133               // PostActions of the containing Test or Flow (a typical use here might be to set the
134               // ExecResult of the containing Test or Flow based on some combination of
135               // previously-executed tests). Typically used to construct subflows which can be used
136               // in place of Tests.
137   >
138
139 func_stmt =
140   < FUNC_NAME ; // name of a Function from FunctionDefs block
141   | FUNC_NAME { // name of a Function from FunctionDefs block
142     // VAR_NAME is the name of a formal parameter as defined for FUNC_NAME in FunctionDefs block.
143     // expr is a value or variable accessible in the scope from which func_name is called.
144     // For In parameter types, expr can be a variable or a value. The value of expr is passed to the formal
145     // parameter during the function call.
146     // For InOut parameter types, expr must be a variable accessible from the calling scope; the value of expr is
147     // assigned to the formal parameter, and if modified inside the function call, the variable used for expr is
148     // updated; the updated value is available in the calling scope.
149     // For Out parameter types, expr must be a variable. Upon returning to the calling scope, the variable used
150     // for expr contains the value assigned inside the function call.
151     (VAR_NAME = expr);*
152     (VAR_NAME = [ expr (expr)+];)* // For arrays (?)
153   } // end func_stmt
154   >
155
156 test_elements_stmt =
157   < VAR_NAME = expr;
158   | VAR_NAME = [ expr (expr)+]; // For arrays (?)
159   | PreActions { ( < action_stmt | bypass_stmt )* } )
160   | FuncExec func_stmt | FuncExec { ( func_stmt )* } | ( flownode_stmt )*
161   | PostActions { ( action_stmt )* }
162   | PassActions { ( action_stmt )* }
163   | FailActions { ( action_stmt )* }
164   >
165
166 test_instance_stmt =
167   // create a named Test from a TestType, using the default elements for the TestType
168   TEST_TYPE TEST_INSTANCE_NAME;

```

```

169
170 // create a named Test from a TestType, overriding some or all of the default elements of the TestType.
171 // Any element specified in the instantiation (i.e., PreActions, FailActions) completely replaces that
172 // element as specified in the type definition.
173 | TEST_TYPE TEST_INSTANCE_NAME { ( test_elements_stmt )* }
174
175 run_result_item = integer | integer:integer
176
177 run_result_list = run_result_item | run_result_list, run_result_item
178
179 // Note that exit_port_expr can use single values or run_result_lists on the RHS of an equality expression
180 // i.e., "ExecResult == 0;" or " ExecResult == 0,1,5,7;" or "ExecResult == -3:-1,1,4:6;"
181 exit_port_expr = <integer_expr == run_result_list | boolean_expr >
182
183 flownode_stmt =
184     FlowNode (NODE_NAME) {
185         ( PreActions { ( < action_stmt | bypass_stmt )* } )
186         // NullExec allowed in TestExec statement of FlowNode ONLY in default FlowNode
187         // definition (a FlowNode defined OUTSIDE the scope of a Test or Flow).
188         ( TestExec execute_stmt | TestExec;)
189         ( PostActions { ( action_stmt)* } )
190         ( ExitPorts {
191             ( Port PORTLABEL boolean_expr { (exit_port_stmt)* } )*
192         } ) // end ExitPorts
193     } // end FlowNode
194 | TestExec execute_stmt // Default FlowNode is used around a bare TestExec statement in this context.
195
196 flow_elements_stmt =
197     < VAR_NAME = expr;
198     | VAR_NAME = [ expr (expr)+]; // For arrays (?)
199     | PreActions { ( < action_stmt | bypass_stmt )* } )
200     | ( flownode_stmt )*
201     | PostActions { ( action_stmt)* } )
202     | PassActions { (action_stmt)* } // end PassActions
203     | FailActions { (action_stmt)* } // end FailActions
204     >
205
206 flow_instance_stmt =
207     // create a named Flow from a FlowType, using the default elements for the FlowType
208     < (FLOW_TYPE) FLOW_INSTANCE_NAME ; // if FLOW_TYPE is not specified, then the
209     // STIL.4 default FLOW_TYPE is used
210     // create a named Flow from a FlowType, overriding some or all of the default elements of the FlowType.
211     // Any element specified in the instantiation (i.e., PreActions, flownodes, PassActions) completely
212     // replaces that element as specified in the type definition.
213     | (FLOW_TYPE) FLOW_INSTANCE_NAME { ( flow_elements_stmt )* } > // if FLOW_TYPE is not specified, then the
214     // STIL.4 default FlowType (the unnamed
215     // FlowType) is used.
216
217 test_execute_stmt =
218     < TEST_NAME ; // execute a named Test
219     | TEST_TYPE; // create and execute a temporary inline Test (named _INLINE_TEST)
220     // from TestType, using the type's default element value.
221     | TEST_TYPE { ( test_elements_stmt )* } > // Create and execute a temporary inline Test
222     // (named _INLINE_TEST) from TestType,
223     // overriding the type's default element values
224

```

```

225 flow_execute_stmt =
226     < FLOW_NAME; // execute a named Flow
227     | FLOW_TYPE; // create and execute a temporary inline Flow(named _INLINE_FLOW)
228                 // from FlowType, using the type's default element values
229     | FLOW_TYPE { (flow_elements_stmt)* } > // Create and execute an a temporary inline Flow
230                                             // (named _INLINE_FLOW) from FlowType,
231                                             // overriding the type's default element values
232
233 execute_stmt =< test_execute_stmt | flow_execute_stmt >
234
235 var_attributes =
236 < Permissions RW | RRW | RO; // Default value if not specified is RW (what do each of these mean?)
237 | Owner System | User; // Default value if not specified is User
238 | Description <string>; // Default value if not specified is empty string ("").
239 >
240
241 // If variable is an array type, the number of elements in that array is given by VAR_NAME.Size. Arrays are
242 // indexed from 0 to<VAR_NAME.Size-1>. .Size will return the number of elements in an array, not the total
243 // memory required by each element .Size is illegal for non-array variables.
244 var_elements_stmt =
245     // Do we need an Operator attribute? Intended to identify variables whose values are set by a
246     // query to an operator and the operator's response (i.e., lot ID)
247
248     // Uninitialized scalar variable
249     < var_type VAR_NAME;
250     | var_type VAR_NAME { (var_attributes)* }
251
252     // Uninitialized array variable (length integer_expr)
253     | var_type VAR_NAME[integer_expr];
254     | var_type VAR_NAME[integer_expr] { (var_attributes)* }
255
256     // Initialized scalar variable.
257     | var_type VAR_NAME = initial_value_element;
258     | var_type VAR_NAME = initial_value_element { (var_attributes)* }
259
260     // Initialize array elements to distinct values
261     | var_type VAR_NAME[(integer_expr)] = [initial_value_list];
262     | var_type VAR_NAME[(integer_expr)] = [initial_value_list] { (var_attributes)* }
263
264     // Initialize all array element to same value
265     var_type VAR_NAME[integer_expr] = initial_value_element;
266     | var_type VAR_NAME[integer_expr] = initial_value_element { (var_attributes)* }
267 >
268
269 =====
270 STIL 1.0 {
271     ( Flow 2009; )+
272 }
273
274 =====
275 Variables ( VAR_DOMAIN ) {
276     // If "Design 2005" is specified in the STIL 1.0 statement (above),
277     // all 1450.1 syntax will also accepted.
278     var_elements_stmt*
279 }
280

```

```

281 =====
282 IncludeOnce; // A statement to be used in files to be included. If present in a file, and that file is included
283 // into another file (using the Include "FILE_NAME"; statement - dot0, section 10.1) more than
284 // once, subsequent Include are ignored. Draws from Microsoft's #pragma once.
285 =====
286 FunctionsDefs {
287     ( FUNCTION_NAME {
288         ( Parameters {
289             ( (<In | Out | InOut>) var_elements_stmt ) *
290         } ) // end Parameters
291     } ) * // end function_name
292 } // end FunctionDefs
293
294 =====
295 bin_attribute =
296 <
297     Color <String>; | // Hex, RGB, or name
298     Number <Integer>; | // Integer must be > 0; Bin numbers MUST be unique within a ... BinDef? Axis? Group?
299     Retest <Integer>; | // Integer must be >= 0
300     Terse <String>; |
301     Verbose <String>; |
302     WafermapChar <simple_character>; // From P1450.1999 BNF
303 >
304
305 bin_definition =
306 Bin < BIN_NAME ;
307     | BIN_NAME { (bin_attribute)* }
308 >
309
310 axis_definition =
311 Axis BIN_AXIS_NAME {
312     (MapLowestBin | MapHighestBin) // Default is to use MapLowestBin
313     (bin_definition)+
314 }
315
316 // At least one bin definition is required in each of the Pass and Fail groups
317 SoftBinDefs (SOFT_BIN_DEF_NAME) { // soft bin definitions
318     Pass { // Increment Pass-bin if ReturnState is Pass
319         (Color <String>;) // Contains default color for all Pass bins, "green" if unspecified
320         (bin_definition)+ | (axis_definition)+
321     } // end Pass
322     Fail { // Increment Fail-bin if ReturnState is Fail
323         (Color <String>;) // Contains default color for all Fail bins, "red" if unspecified
324         (bin_definition)+ | (axis_definition)+
325     } // end Fail
326 }
327 HardBinDefs (HARD_BIN_DEF_NAME) { // hard bin definitions
328     Pass { // Increment Pass-bin if ReturnState is Pass
329         (Color <String>;) // Contains default color for all Pass bins, "green" if unspecified
330         (bin_definition)+ | (axis_definition) // hardbin defs can have only one axis – unnamed or named
331     } // end Pass
332     Fail { // Increment Fail-bin if ReturnState is Fail
333         (Color <String>;) // Contains default color for all Fail bins, "red" if unspecified
334         (bin_definition)+ | (axis_definition) // hardbin defs can have only one axis – unnamed or named
335     } // end Fail
336 }

```

```

337 // In the syntax below, BIN_INDEX is the (0-based) index of the Axis or Bin.
338 //
339 // If using BIN_NAME as an index, and BIN_NAME names are unique across bin groups (Pass or Fail), there is no
340 // need to specify the Pass or Fail group. Since each of the Pass and Fail groups (and Bin Axes, if used) have
341 // BIN_INDEX numbers which start from 0, they are by definition reused, and not unique. Therefore, if using
342 // BIN_INDEX rather than BIN_NAME to index the Bins[ ] arrays, the bin group (Pass or Fail) MUST be specified.
343 //
344 // If BIN_NAME names or BIN_NUMBER numbers are unique within a bindef, they can be used as shorthand
345 // notation for bin_expr in place of the full hierarchical access syntax.
346 //
347 // Multiple axes not allowed for hardbins – but can have single named or unnamed hardbin axis.
348 // softbins can have one unnamed axis, or one or more named axes.
349 bin_expr = (Pass | Fail.)Bins[BIN_NAME | BIN_INDEX ] |
350           (Pass | Fail.)Axes[BIN_AXIS_NAME | AXIS_INDEX ].Bins[BIN_NAME | BIN_INDEX ] |
351           BIN_NAME | BIN_NUMBER | BIN_VAR_NAME
352
353 soft_bin_expr = bin_expr;
354 hard_bin_expr = bin_expr;
355
356 // One or more softbins (one from each axis to be mapped) will map to a single hardbin
357 // The final bin_expr in the map statement must be from a bindef specified as a hardbin Bindef in the
358 // TestProgram block. All bin_exprs but the final one must be from a bindef specified as a softbin Bindef
359 // in the TestProgram block.
360
361 // If more than one softbin expression is present, white space is used to separate the list of softbin expressions
362 // It is illegal to map a Fail softbin to a Pass hardbin, or a Pass softbin to a Fail hardbin.
363 bin_map_stmt =
364     Map soft_bin_expr+ hard_bin_expr;
365
366 // When indexing Bin or Axis arrays, either the softbin or bin axis name can be used, or an integer >= 0 can be used.
367 // If bin names or bin numbers are unique across
368
369
370 // Binmap entries showing softbin to hardbin mapping when bin axes are used follow below:
371 // In the four examples below, the number “1” and the number “5” is the hardbin number, not the bin index.
372 Map Pass.Axes[CacheSize].Bins["8Mb"] Pass.Axes[ClockSpeed].Bins["3.00GHz"] 1;
373 Map Pass.Bins["8Mb"] 1;
374
375 // Binmap entries showing softbin to hardbin mapping when no bin axes are used (i.e., an anonymous axis) are
376 // shown below. When no axes are used, there is a single unnamed axis which can be accessed as “Axes[0].Bins[ ]”,
377 // as shown in the second example. However, in that case, the preferred syntax is to skip the axis specifier, and use
378 // only the group and bins specifier, as shown in the first example below.
379 Map Fail.Bins[ContactOpens] 5; // Without axis specifier
380 Map Fail.Axes[0].Bins[ContactOpens] 5; // With axis specifier – must use index 0, as there is only 1 axis
381
382
383 =====
384 BinMap BIN_MAP_NAME { // soft to hard bin mapping
385     // Specify the soft bin and hard bin defs to be used in this bin mapping.
386     SoftBins SOFT_BIN_DEF_NAME;
387     HardBins HARD_BIN_DEF_NAME;
388     bin_map_stmt*
389 }
390 =====
391
392

```

```

393 // Group Property Access Syntax
394 group = < Pass|Fail>
395
396 group_property =
397 <
398     Color | // String (hexadecimal RGB or name, e.g., red)
399     isAnyBinSet | // Boolean
400     Axes.Size // Integer – number of axes. MUST BE >= 1 – no named axis implies anonymous unnamed axis,
401             // so Size will always return value >=1.
402 >
403
404 // Any elements specified by this syntax must be found in the binmap (with its associated
405 // softbin and hardbin definitions) or softbin definitions specified by the test program.
406 group.group_property
407
408 // Axis Property Access Syntax
409 axis_property =
410 <
411     Bins.Size | // Unsigned
412     Name | // String
413     MapType | // Returns MapLowestBin or MapHighestBin
414 >
415
416 // Any elements specified by this syntax must be found in the binmap (with its associated
417 // softbin and hardbin definitions) or softbin definitions specified by the test program.
418 group.Axes[AXIS_INDEX |AXIS_NAME].axis_property
419
420 // Bin Property access syntax
421 counter_reset_event =
422 <
423     OnLoad |
424     OnLotStart |
425     OnRetest |
426     OnStart |
427     OnWaferStart
428 >
429
430 bin_property =
431 <
432     Color | // String (hexadecimal RGB or name, e.g., red)
433     counter.counter_reset_event | // Unsigned
434     Enabled | // Boolean
435     Index | // Unsigned
436     isFailBin | // Boolean
437     isSet | // Boolean
438     Name | // String
439     Number | // Integer
440     retest.(current|Original) | // Unsigned
441     Terse | // String
442     Verbose | // String
443     WafermapChar // Character
444 >
445
446
447
448

```

```
449 // Bidefs access syntax. Any elements specified by this syntax must be found in the binmap (with its associated
450 // softbin and hardbin definitions) or softbin definitions specified by the test program.
451 group.Bins[BIN_INDEX | BIN_NAME ].bin_property |
452 group.Axes [AXIS_INDEX | AXIS_NAME].Bins[BIN_INDEX | BIN_NAME ].bin_property
453 Pass.Axes[CacheSize].Bins["8Mb"] Pass.Axes[ClockSpeed].Bins["3.00GHz"] 1; // UPDATE THIS EXAMPLE!!
```

454 // *Spec and Category access syntax. If spec blocks are specified using Specs/Variables/Categories notation rather*
455 *than Spec/Categories/Variables notation, same notation still applies (since in STIL dot0, spec variables are global*
456 *in scope with the combination of category name + variable name being unique).*
457 SPEC_NAME.**Size** returns the number of categories in a spec block (as an integer)
458
459 (SPEC_NAME.)**Category[I]** returns the Ith category in a spec block (first category is index 0), which can then be
460 further indexed, as shown below.
461
462 (SPEC_NAME.)**Category[I].Name** returns a string containing the category name corresponding to the Ith category in
463 a spec block (first category is index 0)
464
465 (SPEC_NAME.)**Category[I].Size** returns the number of variables in the Ith category (first category is index 0) in a
466 spec block
467
468 (SPEC_NAME.)**Category[I].Variables[J]** returns the Jth variable in the Ith category within a spec block, suitable for
469 further indexing, as shown below. The value of the variable is determined by the currently-active selector, unless
470 that selection is overridden explicitly, as shown below by the **.[Min | Typ | Max | Meas]** syntax.
471
472 (SPEC_NAME.)**Category[I].Variables[J].Name** returns a string containing the variable name of the Jth variable in
473 the Ith category within a spec block.
474
475 (SPEC_NAME.)**Category[I].Variables[J].[Min | Typ | Max | Meas]** returns the Jth variable in the Ith category within
476 a spec block. The value of the variable is determined by the specification of **.[Min | Typ | Max | Meas]**.
477
478 In the above syntax, the explicit category names or variable names can be substituted for Category[I] and
479 Variable[J]. (example to follow).
480
481 =====
482 **TestBase** {
483 (**Parameters** {
484 ((<In | Out | **InOut**>) var_elements_stmt) *
485 }) // end Parameters
486 (**Variables** { var_elements_stmt* } | **Variables** VAR_DOMAIN) *
487 (**PreActions** { (< action_stmt | bypass_stmt) * })
488 // Allows implementers to include a dummy TestExec in TestBase. The dummy TestExec can,
489 // for instance, display a warning or error that an actual Test did not include its own TestExec or
490 // FuncExec, or that an actual Flow did not include at least one FlowNode (which can be empty).
491 (**TestExec** ;)
492 (**PostActions** { (action_stmt) * })
493 (**PassActions** { (action_stmt) * }) // end PassActions
494 (**FailActions** { (action_stmt) * }) // end FailActions
495 } // end TestBase
496
497 =====
498 **TestType** TEST_TYPE_NAME {
499 // If not inheriting from a previously-defined TEST_TYPE_NAME (either vendor-defined or user-defined),
500 // a TestType will inherit from TestBase. The “Inherit TestBase” statement is not required, but is
501 // recommended. If no Inherit statement is present, the behavior is equivalent to “Inherit TestBase”
502 (**Inherit TestBase** ; | **Inherit** TEST_TYPE_NAME ; | **Inherit** FLOW_TYPE_NAME ;)
503 (**Parameters** {
504 ((<In | Out | **InOut**>) var_elements_stmt) *
505 }) // end Parameters
506 (**Variables** { var_elements_stmt* } | **Variables** VAR_DOMAIN) *
507 (**PreActions** { (< action_stmt | bypass_stmt) * })
508 (**TestExec** ; | **FuncExec** func_stmt | **FuncExec** { (func_stmt) * }
509 | flownode_stmt+) // Allows inline instantiation of a flow in a Test. That flow can't be reused.

```

510     ( PostActions { ( action_stmt)* } )
511     ( PassActions { ( action_stmt)* } ) // end PassActions
512     ( FailActions { ( action_stmt)* } ) // end FailActions
513 } // end TestType
514
515 =====
516     ( Test test_instance_stmt )* // Instantiate named Test from TestType
517
518
519
520
521
522
523 =====
524 // A FlowType can be named or unnamed. Only one unnamed FlowType is allowed. If an unnamed
525 // FlowType is not provided by the user, then a default unnamed FlowType must be provided by the
526 // environment or toolset. The contents of this default unnamed FlowType are specified below.
527 FlowType (FLOW_TYPE_NAME) {
528     // If not inheriting from a previously-defined FLOW_TYPE_NAME (either vendor-defined or user-defined),
529     // a FlowType will inherit from TestBase. The “Inherit TestBase” statement is not required, but is
530     // recommended. If no Inherit statement is present, the behavior is equivalent to “Inherit TestBase”
531     ( Inherit TestBase; | Inherit FLOW_TYPE_NAME ; )
532     ( Parameters {
533         ( (<In | Out | InOut>) var_elements_stmt )*
534     } ) // end Parameters
535     ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN )*
536     ( PreActions { ( < action_stmt | bypass_stmt )* } )
537     ( flownode_stmt )*
538     ( PostActions { ( post_action_stmt)* } )
539     ( PassActions { ( action_stmt)* } ) // end PassActions
540     ( FailActions { ( action_stmt)* } ) // end FailActions
541 } // end FlowType
542 =====
543     ( Flow flow_instance_stmt )* // Instantiate named Flow from FlowType
544
545 =====
546 TestProgram TEST_PROGRAM_NAME {
547     SocketDef "SOCKET NAME STRING";
548     // Variables are global to test program and below; local to site (a copy for each site)
549     ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN )*
550     ( BinMap BIN_MAP_NAME; ) | ( SoftBins SOFT_BIN_DEF_NAME; )
551
552     EntryPoints {
553         ( On Exception execute_stmt )
554         ( On Finish execute_stmt )
555         ( On Load execute_stmt )
556         ( On LotEnd execute_stmt )
557         ( On LotStart execute_stmt )
558         ( On MultiSiteDisable execute_stmt )
559         ( On MultiSiteEnable execute_stmt )
560         ( On PatternLoad execute_stmt )
561         ( On PowerDown execute_stmt )
562         ( On Reset execute_stmt )
563         ( On Start execute_stmt )
564         ( On Unload execute_stmt )
565         ( On WaferEnd execute_stmt )

```

```
566         ( OnWaferStart execute_stmt )  
567         } // End Entry Points  
568     } // end TestProgram  
569  
570 =====  
571  
572
```

573 For Multi-Site

```

574
575 // PROPOSED – How do we want to specify different test programs for different sites?
576 // Is this necessary?
577 ( SiteDefs (SITE_DEF_NAME) {
578
579     // Variables are global across all sites (one copy across all sites)
580     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
581
582     ( NumSites integer_expr ; )
583
584     // One statement per site. Only one can be default. If there are more sites than statements, then all
585     // unspecified sites will use the default program.
586     // It has been stated that we'd like to accommodate a scenario in which the same test program is
587     // running on all sites, but a different flow is running on each site. Using the syntax proposed here,
588     // it is possible do this by specifying different test programs for each site. The test programs are
589     // identical EXCEPT that the Flow specified by OnStart is different.
590     ( (Default) SITE_NAME TestProgram TEST_PROGRAM_NAME;)+
591 })
592

```

593 **Variable scoping rules.**

594 Within a test or flow, local scope consists of local variables and parameters. Each name must be unique in that
595 context.

596 Variables, constants, and parameters (including spec variables) declared in the local scope will hide any variables,
597 constants, or parameters of the same name which are declared in the global scope. Global scope includes specs
598 declared in Spec blocks,
599 and variables declared in an unnamed variables block.

600
601 To access the variables in the upper level scope, the complete variable or spec access syntax can be used.

```

602 STIL.4 mandatory requirements for TestBase
603 // The STIL.4-mandated contents for TestBase are shown below. If a user provides an alternate
604 // definition, that definition MUST include all the elements shown below. The purpose of
605 // TestBase is to provide a common set of elements for all TestType and FlowType definitions.
606
607 // The user can define (and redefine) TestBase any number of times prior to first use. After first use,
608 // redefinition of TestBase is NOT allowed. First use of TestBase is defined as the first user definition of a
609 // TestType or FlowType, or first instantiation of a Flow using default FlowType.
610 // It is HIGHLY recommended that a user definition of TestBase occur prior to definition of any
611 // TestTypes, Tests, FlowTypes, Flows, or TestPrograms.
612 TestBase {
613     Parameters {
614         Out Const String TestID = ""; // Empty string – set when test or flow is instantiated.
615         Out Integer ExecResult = 0; // Return value from execution
616                                     // ExecResult == 0 -> Pass, execResult != 0 -> Fail
617                                     // 0 = Pass, non-zero = Fail
618                                     // Define Pass = 0; Define Fail = !Pass
619                                     // For the normal case of only one fail mode, ExecResult = 1 on fail.
620         In Bin FailBin = NoBin; // For component values, see section 3.2.5 of Ernie’s bin doc.
621         In Integer MultiSiteSerial = False; // From dot1, p15, table 5, False = 1;
622     }
623     PreActions { } // No PreActions
624     // TestExec;
625     PostActions { } // No PostActions
626     // Semantics: PassActions or FailActions are selected by the implied arbiter:
627     // if (ExecResult == Pass) { Do PassActions } Else { Do FailActions }
628     PassActions { } // No PassActions. Pass Actions are executed if execResult == Pass
629     FailActions { // Fail Actions are executed if execResult == Fail
630         SetBinStop FailBin; // Perform binning based on the current value of FailBin, and stop
631                             // If value of FailBin is undefined, then no action (binning and stopping)
632                             // takes place.
633                             // This is the equivalent to:
634                             // If (FailBin != NoBin) { SetBin FailBin; Stop }
635     } // end FailActions
636 } // end TestBase
637
638

```

639

STIL.4 definition of default FlowNode

640

// When an unnamed (global) FlowNode definition occurs OUTSIDE of any other scope (i.e., outside

641

// the scope of Tests or Flows, which is the only other place where it's allowed to define a FlowNode),

642

// that FlowNode definition becomes the default FlowNode definition. If the user does NOT define a

643

// default FlowNode, the following definition is used.

644

645

// The user can define (and redefine) the default FlowNode definition any number of times prior to first use.

646

// After first use, redefinition of the default FlowNode is NOT allowed. First use of the default FlowNode

647

// is defined as the first user definition of a FlowType, or first instantiation of a Flow using default

648

// FlowType.

649

// **It is HIGHLY recommended that a user definition of the default FlowNode occur prior to**

650

// **definition of any TestTypes, Tests, FlowTypes, Flows, or TestPrograms.**

651

652

FlowNode {

653

PreActions { }

654

TestExec NullExec; // In actual use, an EXEC_OBJECT_NAME is substituted for NullExec.

655

PostActions { }

656

ExitPorts {

657

Port FAIL CurrentExec.ExecResult == Fail {

658

// Set the ExecResult of the Test or Flow containing

659

// this FlowNode. Either the hard-coded value Fail

660

// or the ExecResult of the Test or Flow which this

661

// FlowNode executed can be used. CurrentExec is an

662

// alias for EXEC_OBJECT_NAME

663

ExecResult = Fail;

664

// or

665

// ExecResult = CurrentExec.ExecResult;

666

Stop; // Stop execution of FlowNodes – return control

667

// to containing Test or Flow, and execute any

668

// PostActions and PassActions/FailActions of

669

// Test or Flow.

670

}

671

Port PASS CurrentExec.ExecResult == Pass { Next; }

672

} // end ExitPorts

673

} // end FlowNode

674

675

676

STIL.4 definition of default FlowType

677

678

// If not provided by the user, the STIL.4 environment or toolset MUST provide for an unnamed default

679

// FlowType as defined below.

680

FlowType {

681

// No Inherit keyword – so this FlowType includes all elements defined by TestBase above

682

// Parameters – use TestBase Parameters

683

// Variables – no local variables

684

// PreActions – no PreActions

685

// Flownodes – no flownodes specified in the type – will be provided at instantiation

686

// setResult CurrentExec.execResult;

687

// PostActions – no PostActions – use TestBase PostActions

688

// PassActions – no PassActions – use TestBase PassActions

689

FailActions {

690

Stop; // Stop, but do no binning here. Binning is done either at the FlowNode or the Test level.

691

}

692

}

693

STIL.4 definition of TestType NoOpType and Test NoOp

```
694
695
696 // In order to help demonstrate concepts in STIL.4, it's necessary to define a TestType, and show how a
697 // Test can be instantiated using this type. Therefore, we define an example TestType called NoOpType,
698 // and instantiate a Test called NoOp from it. This test can be used to illustrate any of the flow concepts
699 // without getting bogged down in the definition of the many actual TestTypes that might be needed on any
700 // specific tester.
701
702 TestType NoOpType {
703     // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
704     // Parameters – use TestBase Parameters
705     // Variables – no local variables
706     // PreActions – no PreActions
707     // No TestExec. No operation performed. Or – use TestExec; (with no tokens).
708     //         Test library must contain a Test named NoOp, which sets the value of
709     //         NoOp.ExecResult to the appropriate value (PASS (0) or FAIL (non-zero))
710     // PostActions – no PostActions
711     // PassActions – no PassActions
712     // FailActions – use FailActions as defined in TestBase
713 } // end TestType NoOp
714
715 Test NoOpType NoOp
```