

1

Revision History

2

Date	Rev	Init	Change
			NOTE: Any line numbers used in a change descriptions for a specific date apply ONLY to the line numbers of the document with that date.
02/29/08	D0.18	jo	Modified <code>bypass_stmt</code> - removed <code>portindex</code> as option in and updated semantics if <code>VAR_NAME</code> is type integer. Modified <code>ExitPort</code> statement to remove <code>portindex</code> (used as a bypass option). Ordinal index order (starting with 0) is now used instead. <code>PORTLABEL</code> is now required, not optional.
3/24/08	D0.18	jo	Modified <code>TestType</code> , <code>FlowType</code> syntax. Changed “UseDefaults” to “Inherit <code>TestBase</code> ”. Clarified semantics if “Inherit” statement is absent.
3/27/08	D0.18	jo	Added revision table to keep track of changes.
5/15/08	D0.19	jo	Added <code>softbin</code> attributes and <code>bin</code> property access syntax (per Ernie’s recent writeups).
5/22/08	D0.20	jo	Added <code>(Const)</code> attribute to <code>var_type</code> ; Removed <code>Operator</code> attribute from <code>var_elements_stmt</code> (may be added back if/when we determine it’s needed). Updated <code>TestBase</code> definition per discussions with Doug and Ernie. Removed <code>ReturnState</code> keyword as option in <code>exit_port_stmt</code> . Modified <code>Return</code> keyword to remove optional (<code>integer_expr</code>) token. <code>ExecResult</code> from <code>TestBase</code> (or derivatives) is used instead.
5/23/08	D0.21	jo	Removed <code>Return</code> keyword from <code>exit_port_stmt</code> . Moved <code>Stop</code> keyword from <code>exit_port_stmt</code> to <code>action_stmt</code> . In <code>PassActions/FailActions</code> of <code>test_elements_stmt</code> and <code>flow_elements_stmt</code> , and in <code>TestBase</code> , <code>TestType</code> , and <code>FlowType</code> syntax, replaced <code>exit_port_stmt</code> with <code>action_stmt</code> .
7/02/08	D0.22	jo	Added optional <code>TestExec</code> statement to <code>TestBase</code> syntax. <code>STIL .4</code> definition definition of <code>TestBase</code> will not include <code>TestExec</code> . Added <code>SetBinStop</code> alternative to <code>action</code> 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 ({ }) to enclose those statements. • Added keyword Port preceding <code>PORTLABEL</code> in <code>ExitPorts</code> statement;

			<p>removed colon (:) after PORTLABEL in <i>exit_port_stmt</i>. Addition of Port keyword makes the colon unnecessary.</p> <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

			<p>statements.</p> <ul style="list-style-type: none"> • Updated default FlowNode definition so that FlowNode FAIL ExitPort actions sets the execResult of the Test or Flow containing the FlowNode. • 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>stl_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 overridd 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>stl_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 TestExec, which implies that the default FlowNode is used around the TestExec. Update TestBase 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 TestExec 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; (NullExec keyword added no additional value). Removed Title STRING; statement from <code>flow_elements_stmt</code> and FlowType. 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 isSet property (not needed). Removed ContinueOnFail bin property. Deferred to phase 2. Added IsFailBin property. Added variable attributes per discussions on 11/20/08 and 12/04/08. Added .Size operator for arrays. Added property queries for Specs/Categories and Bin/BinAxis 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 IncludeOnce statement (reference to MSoft #pragma once, instead of IncludeOnce) In Spec/Category hierarchy access syntax, make SPEC_NAME optional. Removed OnSiteStart/OnSiteEnd EntryPoint keywords. Can’t figure out how these might differ from OnStart (do we need an OnEnd – or maybe OnTestStart/OnTestEnd?).
03/19/09	D0.27	jo	<ul style="list-style-type: none"> In Spec/Category 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 (Pass Fail), Axes, and Bins per latest discussions and Ernie’s binning document. Changed all references to “Unsigned (integer)” to Integer, specifying value range limitation ≥ 0
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 MapBinHighest/MapBinLowest statements in Axis definition. If multiple bins are set on an axis, then the MapBinHighest or MapBinLowest will be used to determine which bin to use in mapping softbins to hardbins. If neither is specified, then MapBinLowest will be used as a default. • Change EntryPoints keywords from On<something> to On <something>. This allows for the use of user-defined keywords as (non-portable) entry points.
07/15/09	D0.27	jo	<ul style="list-style-type: none"> • Modify list of allowable EntryPoints to match those agreed to in last several meetings (and specified in Ernie's binning document dated 7/9/09). Removed: On Exception, On Finish, On MultiSiteDisable, On MultiSiteEnable, OnPatternLoad, On PowerDown, On Unload. Added: On <USER_KEYWORD>. • Added: <i>unary_bin_expr</i>, <i>unary_axis_expr</i>, <i>multi_bin_expr</i> metatypes from Ernie's binning document. Updated SetBin, SetBinStop, and ClearBin statements to use these metatypes for single or multiple bins.

07/15/09	D0.27	jo	<ul style="list-style-type: none"> • Reworked organization of various actions statements (pre, post, pass, fail, exit ports). Restructured definitions (starting on p8, lines 57-139) of these elements, and their usage (lines 158-166, 185-206, and 534-588). Moved binning action statements to binning section of syntax (lines 297-505). Pre-actions now include only variable assignment and bypass actions – binning, stop, or exit actions are now available only in PostAction, PassAction, FailAction, or exit ports action blocks. Renamed <i>bypass_actions</i> and <i>bypass_stmt</i> metatypes to <i>pre_actions</i> and <i>pre_action_stmt</i>; renamed <i>actions</i> and <i>action_stmt</i> metatypes to <i>post_actions</i> and <i>post_action_stmt</i>. Separated binning actions into their own metatype (<i>bin_action</i>, defined in binning section and referenced in <i>post_action</i> metatype. Note that the SetBin, SetBinStop, and ClearBin statements can take a BIN_VAR_NAME (a variable which can contain a <i>soft_bin_expr</i>), as well as a <i>unary_bin_expr</i> (which must be a <i>soft_bin_expr</i>) or a <i>multi_bin_expr</i>. Added <i>color_string</i> metatype, to be used in Color statements of the various binning constructs. <i>color_string</i> can be either a 6-digit hex number representing an RGB color (i.e., #0FC9FF for this color), or a string such as “red” or “green”.
07/22/09	D0.27	jo	<ul style="list-style-type: none"> • Updated Port statement in ExitPorts block (lines 192-194) to use <i>exit_port_expr</i> metatype rather than <i>boolean_expr</i>. This change should have been made at D0.23, 08/14/08. • In Spec/Category access syntax (lines 506-531), change Category (singular) to Categories (plural) for consistency with Variables (in this section) and Axis/Bin access syntax • Update binning group, axis, and bin property access syntax to match current binning document. • Added ReInitializeAt <i>reset_event</i> attribute for variables. Default if not specified is Load (variables are set to their initial value at program load time only).
09/23/09	D0.28	jo	<ul style="list-style-type: none"> • Split <i>bin_attribute</i> and <i>bin_definition</i> metatypes into separate metatypes for soft and hard bins (<i>soft_bin_attribute</i>, <i>soft_bin_definition</i>, <i>hard_bin_attribute</i>, <i>hard_bin_definition</i>); changed references of <i>bin_definition</i> in SoftBinDefs and HardBinDefs to <i>soft_bin_definition</i> and <i>hard_bin_definition</i>, respectively. This change was done to restrict the Retest attribute to HardBins only • Removed LotEnd, WaferEnd <i>reset_event</i> keywords. • Update TestBase minimum requirements. Removed multi-site serial; added TestType, Failed, TestNum. Changed type of ExecResult from integer to TestStatus. Allowed enum syntax per C/C++ rules; added TestStatus definition using that syntax. • Update BinMap syntax. Add “->” token between list of soft bins, and hard bin (makes the parser’s job easier). • Modified default Bin numbers (if not specified) in lines 364-388 to be <last_used_bin_number> + 1. • Added section describing Retest behavior (lines 789-811)

09/23/09	D0.28	jo	<ul style="list-style-type: none">• Added StartBinNumber <Integer>, BinNumberIncrement <Integer> statements to SoftBinDefs and HardBinDefs blocks. See lines 403-404 and 415-416.• Modified behavior of user-assigned vs. default bin numbers. Within a Bindef, if any bin numbers are specified, all must be specified. If no bin numbers are specified, numbering starts at value specified by StartBinNumber value, and each subsequent bin statement will use a bin number = <last_used_bin_number> + BinNumberIncrement. See comments at lines 366-375.
----------	-------	----	---

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  var_assignment_stmt =
58  <
59      VAR_NAME = expr; // Scalar variables
60      | VAR_NAME[integer_expr] = expr; // Array variables
61  >
62
63  pre_action =
64  <
65      var_assignment_stmt
66
67      // For FlowNodes, Flows, and Tests
68      // Skip only Test execution or FlowNode sequence execution, resume at entry to PostActions Block
69      // Normal processing continues from there. Execute PostActions, and PassActions/FailActions (as
70      // determined by execResult of Test or Flow), or ExitPort selection Actions (as determined by flownode
71      // exit port selector). Selection of Pass/Fail path (for Tests and Flows) is controlled by execResult (which
72      // can be forced to a desired state prior to Bypass statement). Selection of ExitPort path (for flownodes)
73      // is also controlled by boolean expressions – typically, the return status of the test executed by the
74      // FlowNode. This return state (or value of any other boolean expression) can be forced to a desired state
75      // prior to the Bypass statement.
76      | Bypass;
77
78      // For Flows and Tests only.
79      // Skip Test and PostActions, resume execution at entry to PassActions/FailActions.
80      // If Pass or Fail specified, follow that path. Otherwise, VAR_NAME is a string variable which specifies
81      // either PASS or FAIL. If using VAR_NAME, and it's an empty string, no bypass action occurs
82      // If SkipActions specified, don't execute PassActions/FailActions (equivalent to a return,
83      // since there's no branching from a Test or Flow)
84      | Bypass (GoTo <Pass | Fail | VAR_NAME > ( SkipActions ) );
85
86      // For FlowNodes only.
87      // Skip Test and PostActions, resume execution at entry to specified ExitPort.
88      // If SkipActions specified, don't execute don't execute the exit port actions.
89      // PORTLABEL is a string specifying a port label. VAR_NAME is either a string variable or an integer variable.
90      // If a string variable, it specifies the bypass exit port by label. If using VAR_NAME, and it's an empty string,
91      // no bypass action occurs. If an integer variable, it specifies the bypass exit port by index (based on the
92      // ordinal order of the exit ports, from top of list to bottom, with the first index being 0.
93      | Bypass (GoTo < PORTLABEL | VAR_NAME > ( SkipActions ) );
94  >
95
96  pre_action_stmt =
97      < If boolean_expr { pre_action } | pre_action >
98
99  post_action =
100 <
101     var_assignment_stmt
102     | bin_action // See binning section for definition of bin_action
103
104     // Return to the caller, "bubbling up" through the levels of FlowNodes, Flows and Tests, until the
105     // initiating EntryPoint is reached. At each level, execute the PostActions and PassActions or
106     // FailActions for Tests and Flows, but SKIP execution of FlowNodes (including any FlowNode
107     // PostActions or ExitPort actions).
108     | Stop; // terminate the initiating On* condition.
109
110     // Exit – Immediately stop execution of current block (test, flow, or flow node), and jump directly to the
111     // initiating EntryPoint (On* condition) for end-of-test processing. The pass/fail result returned to the
112     // EntryPoint initiator (which would normally be the ExecResult of the Test or Flow specified by the

```

```

113      // EntryPoint), is specified by the integer_expr token of the “Exit <integer_expr>;” statement. This
114      // statement is generally intended to be used for exceptions, such as hardware failures, which require
115      // immediate termination of test execution.
116      | Exit integer_expr; // terminate the initiating On* condition.
117  >
118
119  post_action_stmt =
120  <
121      If boolean_expr | Else If boolean_expr | Else { // Usual rules for If/Else If/Else apply
122          ( post_action )*
123      }
124      | ( post_action )*
125  >
126
127  exit_port_stmt =
128  <
129      post_action_stmt
130      | Next (FLOW_NODE_NAME); // For last FlowNode in a sequence, “Next;” == “Return;”
131      | Return; // Stop execution of FlowNodes. Return to execution chain of containing Test
132                // or Flow (PostActions, PassActions, or FailActions). ExecResult of containing Test
133                // or Flow can be set either by the FlowNode PostActions or ExitPort actions
134                // (typically, based on the results of the most recent TestExec object), or by the
135                // PostActions of the containing Test or Flow (a typical use here might be to set the
136                // ExecResult of the containing Test or Flow based on some combination of
137                // previously-executed tests). Typically used to construct subflows which can be used
138                // in place of Tests.
139  >
140
141  func_stmt =
142  < FUNC_NAME; // name of a Function from FunctionDefs block
143  | FUNC_NAME { // name of a Function from FunctionDefs block
144      // VAR_NAME is the name of a formal parameter as defined for FUNC_NAME in FunctionDefs block.
145      // expr is a value or variable accessible in the scope from which func_name is called.
146      // For In parameter types, expr can be a variable or a value. The value of expr is passed to the formal
147      // parameter during the function call.
148      // For InOut parameter types, expr must be a variable accessible from the calling scope; the value of expr is
149      // assigned to the formal parameter, and if modified inside the function call, the variable used for expr is
150      // updated; the updated value is available in the calling scope.
151      // For Out parameter types, expr must be a variable. Upon returning to the calling scope, the variable used
152      // for expr contains the value assigned inside the function call.
153          (VAR_NAME = expr;)*
154          (VAR_NAME = [ expr (expr)+];)* // For arrays (?)
155      } // end func_stmt
156  >
157
158  test_elements_stmt =
159  < VAR_NAME = expr;
160  | VAR_NAME = [ expr (expr)+]; // For arrays (?)
161  | PreActions { ( < pre_action_stmt )* } )
162  | FuncExec func_stmt | FuncExec { ( func_stmt )* } | ( flownode_stmt )*
163  | PostActions { ( post_action_stmt )* }
164  | PassActions { ( post_action_stmt )* }
165  | FailActions { ( post_action_stmt )* }
166  >
167
168

```

```

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

```

```

225
226 flow_execute_stmt =
227     < FLOW_NAME; // execute a named Flow
228     | FLOW_TYPE; // create and execute a temporary inline Flow(named _INLINE_FLOW)
229     // from FlowType, using the type's default element values
230     | FLOW_TYPE { (flow_elements_stmt)* } > // Create and execute an a temporary inline Flow
231     // (named _INLINE_FLOW) from FlowType,
232     // overriding the type's default element values
233
234 execute_stmt = < test_execute_stmt | flow_execute_stmt >
235
236 var_attributes =
237 < Permissions RW | RRW | RO; // Default value if not specified is RW (what do each of these mean?)
238 | Owner System | User; // Default value if not specified is User
239 | Description <string>; // Default value if not specified is empty string (“”).
240 | ReInitializeAt reset_event; // See binning section for definition of reset_event. Default is Load
241 >
242
243 // If variable is an array type, the number of elements in that array is given by VAR_NAME.Size. Arrays are
244 // indexed from 0 to <VAR_NAME.Size-1>. .Size will return the number of elements in an array, not the total
245 // memory required by each element .Size is illegal for non-array variables.
246 var_elements_stmt =
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 FunctionDefs {
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
296

```

```

297 =====
298 // Bins are automatically cleared by the OnStart event handler before any Test or Flow is executed. When a bin is set,
299 // it remains set until cleared implicitly by the next OnStart event, or explicitly by a ClearBin statement. Bins may be
300 // set or cleared by the user only in FlowNode, Test, and/or Flow PostActions, PassActions or FailActions (for Tests
301 // or Flows), or ExitPorts actions (for FlowNodes) using the SetBin, SetBinStop, or ClearBin statements.
302
303 // In the syntax below, AXIS_INDEX and BIN_INDEX are the (0-based) indices of the Axis or Bin.
304
305 // If using only a single axis (named or unnamed), then the axis specifier (Axes[ BIN_AXIS_NAME | AXIS_INDEX ])
306 // can be omitted, allowing a syntax similar to that used existing systems which currently do not use the axis concept.
307
308 // If BIN_NAME names are unique across bin groups (Pass or Fail), and it is desired to use them as symbolic indices,
309 // the group specifier (Pass or Fail) can be omitted . Since the BIN_INDEX numbers for each of the Pass and Fail
310 // groups (and axes, if used) start from 0, they are by definition reused, and not unique. Therefore, if using
311 // BIN_INDEX rather than BIN_NAME to index the Bins[ ] arrays, the bin group (Pass or Fail) MUST be specified.
312
313 // If BIN_NAME names or BIN_NUMBER numbers are unique within a bindef, they can be used as shorthand
314 // notation for unary_bin_expr in place of the full hierarchical access syntax. This is typically used for hardbins,
315 // which on many systems have simply been integer numbers. It can also be used for softbins, though, if desired, and
316 // if the uniqueness constraint is met.
317
318 // Multiple axes not allowed for hardbins – but can have single named or unnamed hardbin axis.
319 // Softbins can have one unnamed axis, or one or more named axes.
320
321 unary_bin_expr =
322 <
323     (Pass | Fail).(Axes[ BIN_AXIS_NAME | AXIS_INDEX ].)Bins[ BIN_NAME | BIN_INDEX ] |
324     BIN_NAME | BIN_NUMBER
325 >
326
327 soft_bin_expr = unary_bin_expr;           // Must be from a SoftBinDefs block
328 hard_bin_expr = unary_bin_expr;         // Must be from a HardBinDefs block
329
330 unary_axis_expr =
331 <
332     (Pass | Fail).Axes[BIN_AXIS_NAME | AXIS_INDEX] | BIN_AXIS_NAME
333 >
334 multi_bin_expr =
335 <
336     All                               // All Bins in currently selected SoftBinDefs
337     | All <Pass | Fail>               // All Bins in specified group of currently selected SoftBinDefs
338     | All unary_axis_expr             // All Bins in specified Axis of currently selected SoftBinDefs
339 >
340
341 // color_string can be a hex color specifier (ex: #0FC9FF for this color), or a string (ex: "red")
342 color_string = # <hex_char><hex_char><hex_char><hex_char><hex_char><hex_char> | <String>
343
344 bin_action =
345 <
346     // In the following SetBin, ClearBin, or SetBinStop statements, the bin must be selected from the SoftBin
347     // definitions. Mapping of the various soft bins to hard bins is done at the end of test at bin mapping time.
348     // All keyword in place of soft_bin_expr will set (or clear) all bins on all axes of all groups of selected
349     // SoftBin definition (to support disqualify binning and other such binning strategies). Setting of hard bins
350     // (only one axis allowed) is done at end of test at bin mapping time
351     SetBin <BIN_VAR_NAME | unary_bin_expr | multi_bin_expr>;
352     | ClearBin <BIN_VAR_NAME | unary_bin_expr | multi_bin_expr >;

```

```

353 // SetBinStop is equivalent, semantically, to:
354   If (<bin_name> != NoBin) { SetBin <bin_name>; Stop;} Else { // No Action }
355 | SetBinStop < BIN_VAR_NAME | unary_bin_expr | multi_bin_expr >;
356 // Return to the caller, “bubbling up” through the levels of FlowNodes, Flows and Tests, until the initiating
357 // EntryPoint is reached. At each level, execute the PostActions and PassActions or FailActions for Tests
358 // and Flows, but SKIP execution of FlowNodes (including any FlowNode PostActions or ExitPort actions
359 | Stop; // Terminate the initiating On* condition.
360 // No binning is done directly, binning will be determined by previous SetBin/ClearBin statements.
361 >
362
363 soft_bin_attribute =
364 <
365   Color color_string; // Hex, RGB, or name
366   | Number < Integer>; // Bin number must be > 0; Bin numbers MUST be unique within an axis.
367 // Bin numbers can be made unique across a group or even a bindef.
368 // Within a Bindef, if any bin numbers are explicitly specified, all must be
369 // specified. If no bin numbers are specified, the system automatically assigns
370 // bin numbers as follows: the first Bin statement is assigned Bin number from
371 // the StartBinNumber property of the bindef; each subsequent Bin statement
372 // (across both the Pass and Fail groups, and multiple axes, if used, within each
373 // group) is assigned a Bin number greater by BinNumberIncrement than the
374 // Number used by the previous Bin statement. Note that the defaults for both
375 // StartBinNumber and BinNumberIncrement are both 1.
376   | Terse <String>;
377   | Verbose <String>;
378   | WafermapChar <simple_character>; // From P1450.1999 BNF
379 >
380
381 soft_bin_definition =
382   Bin < BIN_NAME ;
383   | BIN_NAME { (soft_bin_attribute)* }
384 >
385
386 hard_bin_attribute = soft_bin_attribute | Retest <Integer>; // Integer must be >= 0.
387
388 hard_bin_definition =
389   Bin < BIN_NAME ;
390   | BIN_NAME { (hard_bin_attribute)* }
391 >
392
393 axis_definition =
394   Axis BIN_AXIS_NAME {
395     (MapBinLowest | MapBinHighest) // When mapping soft bins to hard bins in the BinMap, if two or
396 // more bins are set on a given axis, choose the soft bin with the
397 // lowest or highest index. Default is to use MapBinLowest
398     (bin_definition)+
399   }
400
401 // At least one bin definition is required in each of the Pass and Fail groups
402 SoftBinDefs (SOFT_BIN_DEF_NAME) { // soft bin definitions
403   (StartBinNumber <Integer>;) // Integer value must be > 0; Default if not specified is 1
404   (BinNumberIncrement <Integer>;) // Integer value must be > 0; Default if not specified is 1
405   Pass { // Increment Pass-bin if ReturnState is Pass
406     (Color color_string;) // Contains default color (name or hex RGB) for all Pass bins, “green” if unspecified
407     (soft_bin_definition)+ | (axis_definition)+
408   } // end Pass

```

```

409     Fail {           // Increment Fail-bin if ReturnState is Fail
410         (Color color_string;) // Contains default color (name or hex RGB) for all Fail bins, “red” if unspecified
411         (soft_bin_definition)+ | (axis_definition)+
412     } // end Fail
413 }
414 HardBinDefs (HARD_BIN_DEF_NAME) { // hard bin definitions
415     (StartBinNumber <Integer>;) // Integer value must be > 0; Default if not specified is 1
416     (BinNumberIncrement <Integer>;) // Integer value must be > 0; Default if not specified is 1
417     Pass { // Increment Pass-bin if ReturnState is Pass
418         (Color color_string;) // Contains default color (name or hex RGB) for all Pass bins, “green” if unspecified
419         (hard_bin_definition)+ | (axis_definition) // hardbin defs can have only one axis – unnamed or named
420     } // end Pass
421     Fail {           // Increment Fail-bin if ReturnState is Fail
422         (Color color_string;) // Contains default color (name or hex RGB) for all Fail bins, “red” if unspecified
423         (hard_bin_definition)+ | (axis_definition) // hardbin defs can have only one axis – unnamed or named
424     } // end Fail
425 }
426 // One or more softbins (one from each axis to be mapped) will map to a single hardbin. The final hard_bin_expr
427 // in the map statement must be from a hardbin Bindef. All soft_bin_exprs must be from a softbin Bindef.
428
429 // If more than one softbin expression is present, white space is used to separate the list of softbin expressions
430 // It is illegal to map a Fail softbin to a Pass hardbin, or a Pass softbin to a Fail hardbin.
431 bin_map_stmt =
432     Map soft_bin_expr+ -> hard_bin_expr;
433
434 // When indexing Bin or Axis arrays, either the softbin or bin axis name can be used, or an integer >= 0 can be used.
435 // If bin names or bin numbers are unique across
436
437 // Binmap entries showing softbin to hardbin mapping when bin axes are used follow below:
438 // In the four examples below, the number “1” and the number “5” is the hardbin number, not the bin index.
439 Map Pass.Axes[CacheSize].Bins["8Mb"] Pass.Axes[ClockSpeed].Bins["3.00GHz"] -> 1;
440 Map Pass.Bins["8Mb"] -> 1;
441
442 // Binmap entries showing softbin to hardbin mapping when no bin axes are used (i.e., an anonymous axis) are
443 // shown below. When no axes are used, there is a single unnamed axis which can be accessed as “Axes[0].Bins[ ]”,
444 // as shown in the second example. However, in that case, the preferred syntax is to skip the axis specifier, and use
445 // only the group and bins specifier, as shown in the first example below.
446 Map Fail.Bins[ContactOpens] -> 5; // Without axis specifier
447 Map Fail.Axes[0].Bins[ContactOpens] -> 5; // With axis specifier – must use index 0, as there is only 1 axis
448
449
450 =====
451     BinMap BIN_MAP_NAME { // soft to hard bin mapping
452         // Specify the soft bin and hard bin defs to be used in this bin mapping.
453         SoftBins SOFT_BIN_DEF_NAME;
454         // If binmap statements are used, must specify hard bin definitions.
455         (HardBins HARD_BIN_DEF_NAME;
456         bin_map_stmt*)
457     }
458 =====
459 // Group Property Access Syntax
460 group = < Pass|Fail>
461
462 group_property =
463 <
464     Color | // color_string: String (hexadecimal RGB or name, e.g., red)

```

```

465     CountSetBins | // Integer, unsigned
466     Axes.Size    // Integer – number of axes. MUST BE >= 1 – no named axis implies anonymous unnamed axis,
467                 // so Size will always return value >=1.
468 >
469
470 group.group_property    // Any elements specified by this syntax must be found in the binmap (with its associated
471                          // softbin and hardbin definitions) or softbin definitions specified by the test program.
472
473 // Axis Property Access Syntax
474 axis_property =
475 <
476     Bins.Size | // Unsigned
477     Name // String
478 >
479 group.Axes[AXIS_INDEX | AXIS_NAME].axis_property // Any elements specified by this syntax must be found in the
480                                                       // binmap (with its associated softbin and hardbin definitions)
481                                                       // softbin definitions specified by the test program.
482 // Bin Property access syntax
483 reset_event =
484 <
485     Load |
486     LotStart |
487     Reset |
488     Start |
489     WaferStart |
490     <UserKeyword>
491 >
492
493 bin_property =
494 <
495     Color |                               // color_string: String (hexadecimal RGB or name, e.g., red)
496     counter.reset_event |                 // Unsigned
497     Enabled |                             // Boolean
498     Index |                               // Unsigned
499     isFailBin |                           // Boolean
500     isSet |                               // Boolean
501     Name |                               // String
502     Number |                             // Integer
503     retest.(current|Limit) |            // Integer – must be >= 0
504     Terse |                              // String
505     Verbose |                            // String
506     WafermapChar                        // Character
507 >
508
509 // Bindefs property access syntax. Any elements specified by this syntax must be found in the binmap (with its
510 // associated softbin and hardbin definitions) or softbin definitions specified by the test program.
511 group.Bins[BIN_INDEX | BIN_NAME ].bin_property |
512 group.Axes [AXIS_INDEX | AXIS_NAME].Bins[BIN_INDEX | BIN_NAME ].bin_property
513
514 Pass.Axes[CacheSize].Bins["8Mb"] Pass.Axes[ClockSpeed].Bins["3.00GHz"] 1; // UPDATE THIS EXAMPLE!!
515
516
517

```

518 // *Spec and Category access syntax. If spec blocks are specified using Specs/Variables/Categories notation rather*
519 *than Spec/Categories/Variables notation, same notation still applies (since in STIL dot0, spec variables are global*
520 *in scope with the combination of category name + variable name being unique).*
521 SPEC_NAME.**Size** returns the number of categories in a spec block (as an integer)
522
523 (SPEC_NAME.)**Categories[I]** returns the Ith category in a spec block (first category is index 0), which can then be
524 further indexed, as shown below.
525
526 (SPEC_NAME.)**Categories[I].Name** returns a string containing the category name corresponding to the Ith category
527 in a spec block (first category is index 0)
528
529 (SPEC_NAME.)**Categories[I].Size** returns the number of variables in the Ith category (first category is index 0) in a
530 spec block
531
532 (SPEC_NAME.)**Categories[I].Variables[J]** returns the Jth variable in the Ith category within a spec block, suitable for
533 further indexing, as shown below. The value of the variable is determined by the currently-active selector, unless
534 that selection is overridden explicitly, as shown below by the **[Min | Typ | Max | Meas]** syntax.
535
536 (SPEC_NAME.)**Categories[I].Variables[J].Name** returns a string containing the variable name of the Jth variable in
537 the Ith category within a spec block.
538
539 (SPEC_NAME.)**Categories[I].Variables[J].**[Min | Typ | Max | Meas]**** returns the Jth variable in the Ith category
540 within a spec block. The value of the variable is determined by the specification of **[Min | Typ | Max | Meas]**.
541
542 In the above syntax, the explicit category names or variable names can be substituted for Categories[I] and
543 Variable[J]. (example to follow).
544
545 =====
546 **TestBase** {
547 (**Parameters** {
548 ((<In | Out | **InOut**>) var_elements_stmt) *
549 }) // end Parameters
550 (**Variables** { var_elements_stmt* } | **Variables** VAR_DOMAIN) *
551 (**PreActions** { (<pre_action_stmt) * })
552 // Allows implementers to include a dummy TestExec in TestBase. The dummy TestExec can,
553 // for instance, display a warning or error that an actual Test did not include its own TestExec or
554 // FuncExec, or that an actual Flow did not include at least one FlowNode (which can be empty).
555 (**TestExec**;)
556 (**PostActions** { (post_action_stmt) * })
557 (**PassActions** { (post_action_stmt) * }) // end PassActions
558 (**FailActions** { (post_action_stmt) * }) // end FailActions
559 } // end TestBase
560
561 =====
562 **TestType** TEST_TYPE_NAME {
563 // If not inheriting from a previously-defined TEST_TYPE_NAME (either vendor-defined or user-defined),
564 // a TestType will inherit from TestBase. The “Inherit TestBase” statement is not required, but is
565 // recommended. If no Inherit statement is present, the behavior is equivalent to “Inherit TestBase”
566 (**Inherit TestBase**; | **Inherit** TEST_TYPE_NAME ; | **Inherit** FLOW_TYPE_NAME ;)
567 (**Parameters** {
568 ((<In | Out | **InOut**>) var_elements_stmt) *
569 }) // end Parameters
570 (**Variables** { var_elements_stmt* } | **Variables** VAR_DOMAIN) *
571 (**PreActions** { (<pre_action_stmt) * })
572 (**TestExec**; | **FuncExec** func_stmt | **FuncExec** { (func_stmt) * }
573 | flownode_stmt+) // Allows inline instantiation of a flow in a Test. That flow can't be reused.

```

574     ( PostActions { ( post_action_stmt)* } )
575     ( PassActions { ( post_action_stmt)* } ) // end PassActions
576     ( FailActions { ( post_action_stmt)* } ) // end FailActions
577 } // end TestType
578
579 =====
580     ( Test test_instance_stmt ) * // Instantiate named Test from TestType
581
582 =====
583 // A FlowType can be named or unnamed. Only one unnamed FlowType is allowed. If an unnamed
584 // FlowType is not provided by the user, then a default unnamed FlowType must be provided by the
585 // environment or toolset. The contents of this default unnamed FlowType are specified below.
586 FlowType (FLOW_TYPE_NAME) {
587     // If not inheriting from a previously-defined FLOW_TYPE_NAME (either vendor-defined or user-defined),
588     // a FlowType will inherit from TestBase. The "Inherit TestBase" statement is not required, but is
589     // recommended. If no Inherit statement is present, the behavior is equivalent to "Inherit TestBase"
590     ( Inherit TestBase; | Inherit FLOW_TYPE_NAME ; )
591     ( Parameters {
592         ( (<In | Out | InOut>) var_elements_stmt ) *
593     } ) // end Parameters
594     ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN ) *
595     ( PreActions { ( <pre_action_stmt)* } )
596     ( flownode_stmt ) *
597     ( PostActions { ( post_action_stmt)* } )
598     ( PassActions { ( post_action_stmt)* } ) // end PassActions
599     ( FailActions { ( post_action_stmt)* } ) // end FailActions
600 } // end FlowType
601
602 =====
603     ( Flow flow_instance_stmt ) * // Instantiate named Flow from FlowType
604
605 =====
606 TestProgram TEST_PROGRAM_NAME {
607     SocketDef "SOCKET NAME STRING";
608     // Variables are global to test program and below; local to site (a copy for each site)
609     ( Variables { var_elements_stmt* } | Variables VAR_DOMAIN ) *
610     ( BinMap BIN_MAP_NAME; )
611
612     EntryPoints {
613         ( On Load execute_stmt )
614         ( On LotEnd execute_stmt )
615         ( On LotStart execute_stmt )
616         ( On Reset execute_stmt )
617         ( On Start execute_stmt )
618         ( On WaferEnd execute_stmt )
619         ( On WaferStart execute_stmt )
620         ( On <USER_KEYWORD> execute_stmt )
621     } // End Entry Points
622 } // end TestProgram
623
624 =====
625

```

626 For Multi-Site

```

627
628 // PROPOSED – How do we want to specify different test programs for different sites?
629 // Is this necessary?
630 ( SiteDefs (SITE_DEF_NAME) {
631
632     // Variables are global across all sites (one copy across all sites)
633     (Variables { var_elements_stmt* } | Variables VAR_DOMAIN)*
634
635     ( NumSites integer_expr ; )
636
637     // One statement per site. Only one can be default. If there are more sites than statements, then all
638     // unspecified sites will use the default program.
639     // It has been stated that we'd like to accommodate a scenario in which the same test program is
640     // running on all sites, but a different flow is running on each site. Using the syntax proposed here,
641     // it is possible do this by specifying different test programs for each site. The test programs are
642     // identical EXCEPT that the Flow specified by OnStart is different.
643     ( (Default) SITE_NAME TestProgram TEST_PROGRAM_NAME;)+
644 })
645
```

646 Variable scoping rules.

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

649 Variables, constants, and parameters (including spec variables) declared in the local scope will hide any variables,
650 constants, or parameters of the same name which are declared in the global scope. Global scope includes specs
651 declared in Spec blocks, variables declared in the TestProgram scope, variables declared in the global scope in a
652 named variables block, and variables declared in an unnamed variables block.

654 To access the variables in the scopes other than the local scope, the following rules apply:

- 655 • To access a global variable that is not hidden by a local parameter or variable, simply refer to it by name.
- 656 • To access a global variable that IS hidden by a local parameter or variable, use the notation
657 Global.<var_name>.
- 658 • To access a local variable, simply refer to it by name, or use the notation Local.<var_name>. This
659 capability allows the programmer to make it clear that the local scope should be used, regardless of whether
660 or not this variable hides another by the same name in the global scope. This behavior happens
661 automatically if a variable in the local scope hides another variable in the global scope), the notation
662 Local.<var_name> can be used. The use of “Local.<var_name>” is analogous to the “this” pointer (this-
663 ><var_name>) from C++.
- 664 • For parameters of other instantiated objects, the notation <object_name>.<parameter_name> can be
665 used. upper level scope, the complete variable or spec access syntax can be used.

```

666      STIL.4 mandatory requirements for TestBase
667      // The STIL.4-mandated contents for TestBase are shown below. If a user does not provide a definition, the
668      // definition below is used. If a user provides an alternate definition, that definition MUST include all the elements
669      // shown below. The purpose of TestBase is to provide a common set of elements for all TestType and FlowType
670      // definitions.
671      // The user can define TestBase only once. After the first definition, redefinition of TestBase is NOT allowed.
672
673      // enums following the syntax of C/C++ are allowed. Syntax is as given in “The Annotated C++ Reference Manual”
674      // (Ellis and Stroustrup, McGraw Hill, 1990), section 7.2, pp 113-115)
675      // Standard definition of TestStatus. User can extend this with their own definitions, but those definitions must
676      // include the elements below. The user can only ADD to the end of this list, and can’t change the beginning.
677      enum TestStatus {
678          PASS = 0,
679          FAIL,
680          FAILED_TO_RUN,
681          NOT_TESTED
682      }
683
684      TestBase {
685          Parameters {
686              Out Const String TestId = “”; // Empty string – set when test or flow is instantiated.
687              InOut Const Integer TestNum = 0; // Can set test num on object creation; can read only at runtime
688              Out TestStatus ExecStatus = NOT_TESTED; // Return value from execution
689              // PASS , FAIL, FAILED_TO_RUN, NOT_TESTED
690              Out Integer Failed = 0; // Failed == 0 -> Pass, Failed != 0 -> Fail
691              In Bin FailBin = NoBin; // For component values, see section 3.2.5 of Ernie’s bin doc.
692              Out Const String TestType = “”; // Empty string – set to test type when test or flow is instantiated.
693          }
694          PreActions { } // No PreActions
695          // TestExec;
696          PostActions { } // No PostActions
697          // Semantics: PassActions or FailActions are selected by the implied arbiter:
698          // if (ExecResult == Pass) { Do PassActions} Else { Do FailActions }
699          PassActions { } // No PassActions. Pass Actions are executed if execResult == Pass
700          FailActions { // Fail Actions are executed if execResult == Fail
701              SetBinStop FailBin; // Perform binning based on the current value of FailBin, and stop
702              // If value of FailBin is undefined, then no action (binning and stopping)
703              // takes place.
704              // This is the equivalent to:
705              // If (FailBin != NoBin) { SetBin FailBin; Stop}
706          } // end FailActions
707      } // end TestBase
708
709

```

710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764

STIL.4 definition of default FlowNode

```
// When an unnamed (global) FlowNode definition occurs OUTSIDE of any other scope (i.e., outside
// the scope of Tests or Flows, which is the only other place where it's allowed to define a FlowNode),
// that FlowNode definition becomes the default FlowNode definition. If the user does NOT define a
// default FlowNode, the following definition is used.

// The user can define the default FlowNode definition any number of times prior to first use.
// After first use, redefinition of the default FlowNode is NOT allowed. First use of the default FlowNode
// is defined as the first user definition of a FlowType, or first instantiation of a Flow using default
// FlowType.
// It is HIGHLY recommended that a user definition of the default FlowNode occur prior to
// definition of any TestTypes, Tests, FlowTypes, Flows, or TestPrograms.

FlowNode {
    PreActions { }
    TestExec NullExec; // In actual use, an EXEC_OBJECT_NAME is substituted for NullExec.
    PostActions {}
    ExitPorts {
        Port FAIL CurrentExec.ExecResult == Fail {
            // Set the ExecResult of the Test or Flow containing
            // this FlowNode. Either the hard-coded value Fail
            // or the ExecResult of the Test or Flow which this
            // FlowNode executed can be used. CurrentExec is an
            // alias for EXEC_OBJECT_NAME
            ExecResult = Fail;
            // or
            // ExecResult = CurrentExec.ExecResult;
            Stop; // Stop execution of FlowNodes – return control
                // to containing Test or Flow, and execute any
                // PostActions and PassActions/FailActions of
                // Test or Flow.
        }
        Port PASS CurrentExec.ExecResult == Pass { Next; }
    } // end ExitPorts
} // end FlowNode
```

STIL.4 definition of default FlowType

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

FlowType {
    // 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
    // setResult CurrentExec.execResult;
    // PostActions – no PostActions – use TestBase PostActions
    // PassActions – no PassActions – use TestBase PassActions
    FailActions {
        Stop; // Stop, but do no binning here. Binning is done either at the FlowNode or the Test level.
    }
}
```

STIL.4 definition of TestType NoOpType and Test NoOp

```

765
766
767 // In order to help demonstrate concepts in STIL.4, it's necessary to define a TestType, and show how a
768 // Test can be instantiated using this type. Therefore, we define an example TestType called NoOpType,
769 // and instantiate a Test called NoOp from it. This test can be used to illustrate any of the flow concepts
770 // without getting bogged down in the definition of the many actual TestTypes that might be needed on any
771 // specific tester.
772
773 TestType NoOpType {
774     // No Inherit keyword – so this FlowType includes all elements defined by TestBase above
775     // Parameters – use TestBase Parameters
776     // Variables – no local variables
777     // PreActions – no PreActions
778     // No TestExec. No operation performed. Or – use TestExec; (with no tokens).
779     //     Test library must contain a Test named NoOp, which sets the value of
780     //     NoOp.ExecResult to the appropriate value (PASS (0) or FAIL (non-zero))
781     // PostActions – no PostActions
782     // PassActions – no PassActions
783     // FailActions – use FailActions as defined in TestBase
784 } // end TestType NoOp
785
786 Test NoOpType NoOp

```

Retest Behavior

787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811

Each hard bin will have a retest attribute (see *hard_bin_attribute*, line 386). This attribute will be an integer which specifies how the maximum number of times a DUT which bins to this hard bin will be retested. In addition to a retest attribute for each hard bin, there is also a retest counter associated with each hard bin. This retest counter is a read-only attribute (i.e., it cannot be set by the user, but it can be queried by the user, using the hard bin property query `<hard_bin>.retest.current`).

Binning only takes place at the end of the On Start sequence (that is, after all the bubble-up actions described in the comments preceding the `SetBinStop` and `Stop` action statements have completed) . At the end of the On Start sequence, the following actions take place:

- Mapping of soft bin(s) to a hard bin occurs. The bin map is traversed, and when a match is found between the list of currently set soft bins and the soft bin list in each bin map entry, the hard bin specified in that bin map entry is also set.
 - If the retest attribute (`<hard_bin>.retest.current`) of the mapped hard bin is non-zero, a retest is initiated. The retest counter (`<hard_bin>.retest.current`) for that hard bin is decremented.
 - If the retest attribute (`<hard_bin>.retest.current`) of the mapped hard bin is 0, no retest is done.
- If , at the end of the On Start sequence, the retest attribute (`<hard_bin>.retest.current`) for the mapped hard bin is 0 (indicating no retest for that hard bin), then the retest counters for ALL hard bins are reset to the retest limit (i.e. `retest.current` is set from the value of `retest.Limit`).
- There are two ways in which an On Start sequence can result in a hard bin with a retest counter = 0
 - The hard bin was not a retest bin, in which case the retest property was set to 0 (and therefore, the property `retest.current` is also set to 0).
 - The hard bin was a retest bin, with the retest property being non-zero, but the `retest.current` property was set to 0 because it had been decremented the number of times