

IEEE P1450.4 - Flow Extension

LEGEND:

RED = new stuff, currently under review by syntax group
~~REDX = stuff targeted for deletion at next syntax meeting~~
 BLUE = new semantic information, to be reviewed by syntax group
 BLACK = stuff agreed upon by syntax group

Change history for D16:

- 11/14/05 - New D16 will focus on the binning syntax and semantic definitions. D15 was frozen with the Oct31 update.
- 12/5/05 - Changed BinDefs such that Pass|Fail are sub blocks. Allow definition of multiple Axes within Bin-Defs. Defined a new function to return the count, from a given softbin name.
- 12/19/05 - Allow simplified TestFlow by allowing TestExec at the top level and using default conditions for pre/post-actions and exit-ports.
- 12/19/05 - Changed action_stmt such that Bypass is only allowed in pre_actions (i.e., not in post_actions or exit_ports).
- 2/28/06 - The following changes are made at the request of Dave Dowding and Jim O'Reiley:
 - 1) Changed keywords "TestMethod", "TestMethodDefs" to "TestFunction", "TestFunctionDefs",
 - 2) Changed keyword "TestModule" to "TestMethod"
 - 3) Added port label to the "Bypass" statement - to initiate a direct goto an exit port
 - 4) Added port label to the "ExitPorts" statement - to specify target of a bypass with goto

Questions/issues carried over from D14,15:

- 6/1/05 - need to reconcile with stuff in D11 and D13

1. Syntax summary for 1450.4

```

stil_block_type =
1:  Category
2:  | DCLevels
3:  | DCSequence
4:  | DCSets
5:  | Environment
6:  | MacroDefs
7:  | Pattern
8:  | PatternBurst
9:  | PatternExec
10: | Procedures
11: | ScanStructures
12: | Selector
13: | SignalGroups
14: | Signals
15: | Timing
16:
real_var_type =
17: Capacitance // F (Farads)
18: | Compound // combinations of types
19: | Current // A (Amperes)
20: | Double
  
```

```

21: | Frequency // Hz (Herz)
22: | Gain // db (Decibels)
23: | Inductance // H (Henries)
24: | Length // m (Meters)
25: | Power // W (Watts)
26: | Real
27: | Resistance // Ohm (Ohms)
28: | Temperature // Cel (Degrees Celsius)
29: | Time // s (Seconds)
30: | Voltage // V (Volts)
31:
var_type = (3)
32: | Boolean // True/False or Pass/Fail
33: | Integer
34: | SignalRef
35: | SignalVariable
36: | String
37: | real_var_type
38: | stil_block_type
39:
initial_value_stmt = (4)
40: | InitialValue
41: | ( integer_expr )+ // allow if var of typ integer_expr
42: | ( sig_ref_expr )+ // allow if var of type sig_ref_expr
43: | ( < True | False > )+ // allow if var of type Boolean
44: | ( string )+ // allow if var of type String
45: | ( wfc_string )+ // allow if var of type SignalVariable
46: | ( real_expr )+ // allow if var of type real_var_type
47: | ( BLOCK_NAME )+ // allow if var of type stil_block_type
48: ;
49:
execute_stmt = (5)
50: FUNC_NAME ; // name of - TestFunction, TestMethod, TestFlow, or TestInstance
51: | FUNC_NAME { // name of - TestFunction, TestMethod, or TestFlow (TestInstance = no params allowed)
52: | (VAR_NAME = expr;)*
53: | (VAR_NAME = [ expr (expr)+];)*
54: } // end execute_stmt
55:
instance_stmt = (6)
56: FUNC_NAME INSTANCE_NAME ; // name of - TestFunction, TestMethod, or TestFlow
57: | FUNC_NAME INSTANCE_NAME {
58: | (VAR_NAME = expr;)*
59: | (VAR_NAME = [ expr (expr)+];)*
60: } // end instance_stmt
61:
action_stmt =
62: ( < <If | Else If> boolean_expr | Else > { (post_action_stmt)* } ) *
63: ( VAR_NAME | VAR_NAME[INDEX] = expr; ) *
64: ( Bin (BIN_AXIS.)SOFTBIN_NAME; ) *
65:
66:
bypass_stmt =
67: Bypass; // terminate immediately and go to ExitPorts block
68: | Bypass ReturnState integer_expr; ) // terminate immediately and set ReturnState
69: | Bypass GoTo PORTLABEL; ) // terminate immediately and go to labeled exit port
70:
71:
exit_port_stmt = (7)
72: ( action_stmt ) *
73: ( ReturnState integer_expr; ) // default to last TestExec return value
74: ( < Next (FLOW_NODE_NAME); // only allowed in FlowNode
75: | Exit (integer_expr); // terminate the TestFlow or TestMethod; return to caller
76: | Exit (integer_expr) { ReTest integer_expr; } // repeat test specified number of times upon fail
77: | Stop; // terminate the initiating On* condition
78: | Reset; // terminate the initiating On* condition; generate an OnReset event
79: > )

```

```

80:
81: =====
   Variables ( VAR_DOMAIN ) { // extensions to 1450.1 (8)
82:   var_type VAR_NAME (Const) (Operator) ;
83:   var_type VAR_NAME (Const) (Operator) {
84:     (Length integer;) // array of var_type
85:     ( initial_value_stmt )
86:   }
87: }
88:
89: =====
   TestFunctionDefs { (9)
90:   ( TEST_FUNCTION_NAME {
   (Inherit TEST_FUNCTION_NAME;) (10)
   (Parameters { (11)
91:     ( var_type VAR_NAME (< In | Out | InOut > ; )*)
92:     ( var_type VAR_NAME (< In | Out | InOut >) { initial_value_stmt } )*)
93:   } ) // end Parameters
94: } )* // end test_function_name
95: } // end TestFunctionDefs
96:
97: =====
   BinDefs ( BIN_DEF_NAME ) { // soft bin definitions (12)
   (Pass { // Increment Pass-bin if ReturnState is Pass (13)
98:     (Bin SOFTBIN_NAME (integer);)*
99:     (Axis BIN_AXIS {
100:      (Bin SOFTBIN_NAME (integer);)*
101:    } )*)
102:   } ) // end Pass
103:   (Fail { // Increment Fail-bin if ReturnState is Fail
104:     (Bin SOFTBIN_NAME (integer);)*
105:     (Axis BIN_AXIS {
106:      (Bin SOFTBIN_NAME (integer);)*
107:    } )*)
108:   } ) // end Fail
109: }
110:
111: SoftBin \ ( SOFTBIN_NAME \ ) // function call to return contents of a softbin in an expression
112:
113: =====
   BinMap ( BIN_MAP_NAME ) { // soft to hard bin mapping (14)
114:   ( SOFTBIN_NAME -> integer; )*
115:   ( [ (BIN_AXIS.SOFTBIN_NAME)* ] -> integer; )*
116: }
117:
118: =====
   TestMethod TEST_METHOD_NAME { (15)
119:   (Inherit TEST_METHOD_NAME;)
120:   (Parameters {
121:     ( var_type VAR_NAME (< In | Out | InOut > ; )*)
122:     ( var_type VAR_NAME (< In | Out | InOut >) { initial_value_stmt } )*)
123:   } ) // end Parameters
   (Variables VAR_DOMAIN;)* (16)
124:   (PatternExec PAT_EXEC_NAME; )
125:   (PreActions { ( < action_stmt | bypass_stmt )* } )
   (TestExec execute_stmt | TestExec { ( execute_stmt )* } ) (17)
126:   (PostActions { ( action_stmt )* } )
127:   (ExitPorts {
128:     ( (PORTLABEL:) boolean_expr { (exit_port_stmt)* } )*)
129:   } ) // end ExitPorts
130: } // end TestMethod
131:
132: =====
   TestFlow TEST_FLOW_NAME { (18)

```

```

133:  ( Inherit TEST_FLOW_NAME; )
134:  ( Parameters {
135:    ( var_type VAR_NAME (< In | Out | InOut > ; ))*
136:    ( var_type VAR_NAME (< In | Out | InOut > ) { initial_value_stmt } ))*
137:  } ) // end Parameters
      ( Variables VAR_DOMAIN; )*                                     (19)
138:  ( PatternExec PAT_EXEC_NAME; )
139:  ( TestInstances { ( INSTANCE_STMT )* } )
140:  ( Title STRING ; )
141:  ( PreActions { ( < action_stmt / bypass_stmt )* } )
142:  <
143:    ( FlowNode (NODE_NAME) {
144:      ( PreActions { ( < action_stmt / bypass_stmt )* } )
145:      ( TestExec execute_stmt | TestExec { ( execute_stmt )* } )
146:      ( PostActions { ( post_action_stmt)* } )
147:      ( ExitPorts {
148:        ( (PORTLABEL:) boolean_expr { ( exit_port_stmt )* } ))*
149:      } ) // end ExitPorts
150:    })* // end FlowNode
151:  >
152:  ( PostActions { ( post_action_stmt)* } )
153:  ( ExitPorts {
154:    ( (PORTLABEL:) boolean_expr { ( exit_port_stmt)* } ))*
155:  } ) // end ExitPorts
156: } // end TestFlow
157:
158: =====
TestProgram TEST_PROGRAM_NAME {                                     (20)
159:   DUTType "DUT NAME STRING";
160:   SocketDef "SOCKET NAME STRING";
      ( Variables VAR_DOMAIN; )*                                     (21)
161:   ( TestInstances { ( instance_stmt)* } )
162:   ( BinDefs BIN_DEF_NAME; )
163:   ( BinMap BIN_MAP_NAME; )
164:
      EntryPoints {                                               (22)
165:       ( OnException execute_stmt )
166:       ( OnFinish execute_stmt )
167:       ( OnLoad execute_stmt )
168:       ( OnLotEnd execute_stmt )
169:       ( OnLotStart execute_stmt )
170:       ( OnMultiSiteDisable execute_stmt )
171:       ( OnMultiSiteEnable execute_stmt )
172:       ( OnPatternLoad execute_stmt )
173:       ( OnPowerDown execute_stmt )
174:       ( OnReset execute_stmt )
175:       ( OnSiteEnd execute_stmt )
176:       ( OnSiteStart execute_stmt )
177:       ( OnStart execute_stmt )
178:       ( OnUnload execute_stmt )
179:       ( OnWaferEnd execute_stmt )
180:       ( OnWaferStart execute_stmt )
181:   } // End Entry Points
182: } // end TestProgram
=====

```

2. Semantic Rules

(1) *stil_block_type*

(2) *real_var_type*

(3) *var_type*

(4) *initial_value_stmt*

(5) *execute_stmt*

FUNC_NAME: This is the name of either a TestFunction, TestMethod, or TestFlow and is used in the *execute_stmt* that invokes the function. These names all exist in the same name space and all names must be unique across all function types. Name conflicts are error conditions.

(6) *instance_stmt*

(7) *pre/post_action_stmt*

(8) **VAR_NAME | VAR_NAME[INDEX] = *expr*** : An expression is an operation to be performed on variables that are in scope in the current *action_stmt*. The left-hand-side of the expression may be any of the allowed variables in a Variables block, except for Constant variables (IntegerConstant, WFCCConstant). The left-hand-side may also be a spec-variable of type Meas (i.e. *varname.Meas*). The right-hand-side shall be compatible with the variable named and shall follow the expression rules as defined in STIL.1, clause 5.

(9) **Bin:** The Bin statement is used to specify that the specified bin-counter shall be incremented upon exit from the ExitPorts block if it meets certain criteria. If there are multiple Bin statements, then all specified soft-bins shall be incremented if they meet the following criteria. The specified soft-bins shall be incremented if the ReturnState and the BinDefs are the same (i.e., both Pass or both Fail). If the ReturnState is of a different type than the BinDefs, then no increment is done (i.e., if ReturnState is Fail, then a Pass bin is not changed).

(10) *exit_port_stmt*

(11) Variables

VAR_DOMAIN: This is the name assigned to a set of variables in a Variables block. If the Variables block is unnamed, then the variables within it are globally available. A named domain Variables block is allowed to re-define a variable in the global block. However, all named domains that are referenced in a given context shall not allow name conflicts.

(12) TestFunctionDefs

A test-function is a call that is implemented in some procedural language (C, C++, other). As such, the definition in the STIL file/stream is a STIL representation of the information that defines the function call in its implementation language (i.e., C, C++ header file). The STIL representation is required so that: a) the function names are known at the STIL level, and b) so that the parameters can be checked in the TestExec calls to the test functions.

(13) **Inherit:** The Inherit statement is used to import definitions from other like blocks - TestFunctionDefs, TestMethod, and TestFlow, which all have common semantic rules. For this definition, "donor" refers to the block that is being referenced, and "derived" refers to the block being defined. When Variables and Parameters subblocks are inherited, the information in the derived subblock is used to extend the definitions in the donor subblock and it is required that the names shall not conflict. For all other subblocks and statements the definition in the derived subblock completely replaces the definition in the donor subblock. Note that it is possible to prevent inheritance of a given subblock by defining an empty subblock in the derived structure - e.g., "PreActions { }" results in no pre-actions either from the donor or derived blocks.

(14) **Parameters:** This optional block is used to specify the parameters that are to be passed into the containing block (either a testfunction, a testobject, or a testflow). The named variables shall be defined in the environment of the calling block. If the block form of TestExec is used, then formal parameters may be passed to the named function. The parameters shall conform to the Parameter definition for the called function. It is also possible to pass the environment blocknames as parameters (e.g, Timing, Category, Selector, ...) in which case the usage of these parameters is determined by the function and may cause changes to be made to the pre-established environment.

(15) **BinDefs:** The BinDefs block defines the names of a set of soft bins. The block may be either named or unnamed. Only one BinDefs block shall be used by any TestProgram and it shall either be the global (unnamed) block or a referenced (named) block.

- a) The Pass|Fail identifier is a required attribute and indicates whether the defined soft bins are to be incremented when the ReturnState is Pass or Fail.
- b) The soft-bin names are available for reference in the BinMap block that is associated with the same TestProgram. Soft bin names are not available as variable names in an expression, however, the value contained in a soft bin can be accessed by the function - SoftBin (soft_bin_name).
- c) The Bin statement has an optional integer value that specifies a bin number. In the case where multiple soft-bins contain bin numbers, the last one encountered in the flow is the only one that is incremented.
- d) If the Axis statement is used, then an array of soft-bin names is specified.

(16) **BinMap:** The BinMap block defines the mapping of soft-bin-name to hard-bin-numbers. The block may be either named or unnamed. Only one BinMap block shall be used by any TestProgram and it shall either be the global (unnamed) block or a referenced (named) block.

(17) **TestMethod:**

(18) **TestMethod -> Variables:** The Variables statement within a TestMethod block is used to reference a block of variables that are to be local to this TestMethod only. The variables may be passed as formal parameters on an execute statement. This facility is probably most useful when an instance of the node is created (see TestInstance statement). Variables referenced by this statement may override names in a global (unnamed) Variables block. Variables in the global (unnamed) block that are not overridden are also globally available to this block.

(19) **TestExec:** A TestExec may be either a semi-colon terminated statement (without parameters) or a block statement (with parameters). TestExec can be used to execute a function which is defined in a STIL block which is part of the STIL file/stream (i.e., a TestMethod, TestFlow, or TestInstance). Or, TestExec can be used to execute an externally created function (i.e., a test function) in which case the function name and formal parameter list shall be defined in a TestFunctionDefs block which is part of the STIL file/stream.

- a) Sequential TestExec - Multiple TestExec statements shall be executed sequentially in the defined order. Each subsequent TestExec will execute as long as the TestResult returned from the prior execution is true (i.e., an integer greater than 0;)
- b) TestExec environment - Variables and PatternExec statements are used to establish environment conditions prior to the invocation of a TestExec statement. These environment conditions can be established at various levels in the hierarchy - from the TestProgram, TestFlow, or TestMethod block - with the lowest level of the hierarchy taking precedence over all higher levels. The PatternExec contains statements that further define the environment (i.e., Category, Selector, Timing, DCLevels, DCSets, PatternBurst, Variables) and which are to be used to establish the initial context of the PatternExec as well as any other statements within the block. The PatternExec does not actually "execute" until a test-function is invoked that calls for the pattern to execute. If a TestExec causes changes to be made in the environment (i.e., changes as a consequence of TestFunction execution), then those changes remain in effect for the life of that object - TestFlow, TestMethod, TestInstance.
- c) ReturnState - The ReturnState is an integer that is passed as an exit parameter back to the calling function upon completion. The value returned is determined by the function. The value returned from a TestMethod, TestNode, or TestFlow can be either a pass through of the value from the last TestExec or else it can be defined within the PostActions or ExitPorts block.
- d) Pass/Fail - A typical convention is to define Pass/Fail as IntegerConstant with the values of 1/0 respectively and use them as return values. These IntegerConstant definitions, if used, shall be defined in a scoped Variables block. Please refer to STIL.1 for details on boolean_expr.

(20) **TestFlow:**

(21) **TestFlow -> Variables:** The Variables statement within a TestFlow block is used to reference a block of variables that are to be local to this TestFlow only. The variables may be passed as formal parameters on an execute statement. Variables referenced by this statement may override names in a global (unnamed) Variables block. Variables in the global (unnamed) block that are not overridden are also globally available to this block.

(22) **TestProgram**

(23) **TestProgram -> Variables:** The Variables statement within a TestProgram block is used to reference a set of variables that is to be globally available to all TestFlow and TestMethod blocks that are referenced by the On* statements. Variables referenced by this statement may override names in a global (unnamed) Variables block. Variables in the global (unnamed) block that are not overridden are also globally available to this TestProgram block and all of the On* referenced blocks.

(24) **EntryPoints**