

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 D15:

- 8/29/05 - new D15 begun; D14 completes the definition of major structural blocks
- 9/13/05 - changes from syntax meeting (and discussion following the meeting)
- 9/19/05 - changes from syntax meeting (see minutes for detail); TestObjectDef changed to TestNodes; TestFlow-Defs changed to TestFlow.
- 9/26/05 - see syntax minutes for detail
- 10/3/05 - see syntax minutes for detail
- 10/10/05 - see syntax minutes for detail
- 10/17/05 - see syntax minutes for detail; TestNode changed to TestModule; added optional expr to Bypass and Exit;
- 10/31/05 - see syntax minutes for detail; mostly changes to the semantic definition
- 10/31/05 - D15 is frozen with this last update and D16 will be used to resolve the binning syntax.

Questions/issues carried over from D14:

- 6/1/05 - binning syntax needs to be added
- 6/1/05 - need to reconcile with stuff in D11 and D13
- 8/8/05 - Should we allow multiple BinDefs? Inside or outside of a TestProgram block? With domain names?

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 - TestMethod, TestModule, TestFlow, or TestInstance
51: | FUNC_NAME { // name of - TestMethod, TestModule, 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 - TestMethod, TestModule, or TestFlow
57: | FUNC_NAME INSTANCE_NAME {
58: ( VAR_NAME = expr; )*
59: ( VAR_NAME = [ expr (expr)+ ]; )*
60: } // end execute_stmt
61:
action_stmt = (7)
62: ( < < If | Else If > boolean_expr | Else > { ( action_stmt ) * } ) *
63: ( VAR_NAME | VAR_NAME [ INDEX ] = expr; ) *
64: ( Bin ( CATEGORY ) . SOFTBIN_NAME ; )
65: ( Bypass ( integer_expr ; ) // terminate FlowNode or TestModule immediately; go to ExitPorts block
66:
exit_port_stmt = (8)
67: ( action_stmt ) *
68: ( ReturnState integer_expr ; ) // default to last TestExec return value
69: ( < Next ( FLOW_NODE_NAME ) ; // only allowed in FlowNode
70: | Exit ( integer_expr ; ) // terminate the TestFlow or TestModule; return to caller
71: | Stop ; // terminate the initiating On* condition
72: | Reset ; // terminate the initiating On* condition; generate an OnReset event
73: > )
74:
75: =====
Variables ( VAR_DOMAIN ) { // extensions to 1450.1 (9)
76: var_type VAR_NAME ( Const ) ( Operator ) ;
77: var_type VAR_NAME ( Const ) ( Operator ) {
78: ( Length integer ; ) // array of var_type
79: ( initial_value_stmt )
80: }
81: }
82:
83: =====

```

```

TestMethodDefs { (10)
84:   ( TEST_METHOD_NAME {
      ( Inherit TEST_METHOD_NAME; ) (11)
      ( Parameters { (12)
85:         ( var_type VAR_NAME (< In | Out | InOut > ; )*)
86:         ( var_type VAR_NAME (< In | Out | InOut >) { initial_value_stmt } )*)
87:       } ) // end Parameters
88:     })* // end test_method_name
89:   } // end TestMethodDefs
90:
91: =====
TestModule TEST_MODULE_NAME { (13)
92:   ( Inherit TEST_MODULE_NAME; )
93:   ( Parameters {
94:     ( var_type VAR_NAME (< In | Out | InOut > ; )*)
95:     ( var_type VAR_NAME (< In | Out | InOut >) { initial_value_stmt } )*)
96:   } ) // end Parameters
97:   ( Variables VAR_DOMAIN; )* (14)
98:   ( PatternExec PAT_EXEC_NAME; )
99:   ( PreActions { ( action_stmt)* } )
100:  ( TestExec execute_stmt | TestExec { ( execute_stmt)* } ) (15)
101:  ( PostActions { ( action_stmt)* } )
102:  ( ExitPorts {
103:    ( boolean_expr { ( exit_port_stmt)* } ) *
104:  } ) // end ExitPorts
105: } // end TestModule
106: =====
TestFlow TEST_FLOW_NAME { (16)
107:   ( Inherit TEST_FLOW_NAME; )
108:   ( Parameters {
109:     ( var_type VAR_NAME (< In | Out | InOut > ; )*)
110:     ( var_type VAR_NAME (< In | Out | InOut >) { initial_value_stmt } )*)
111:   } ) // end Parameters
112:   ( Variables VAR_DOMAIN; )* (17)
113:   ( PatternExec PAT_EXEC_NAME; )
114:   ( TestInstances { ( instance_stmt)* } )
115:   ( Title STRING ; )
116:   ( PreActions { ( action_stmt)* } )
117:   ( FlowNode (NODE_NAME) {
118:     ( PreActions { ( action_stmt)* } )
119:     ( TestExec execute_stmt | TestExec { ( execute_stmt)* } )
120:     ( PostActions { ( action_stmt)* } )
121:     ( ExitPorts {
122:       ( boolean_expr { ( exit_port_stmt)* } ) *
123:     } ) // end ExitPorts
124:   })* // end FlowNode
125:   ( PostActions { ( action_stmt)* } )
126:   ( ExitPorts {
127:     ( boolean_expr { ( exit_port_stmt)* } ) *
128:   } ) // end ExitPorts
129: } // end TestFlow
130: =====
TestProgram TEST_PROGRAM_NAME { (18)
131:   DUTType "DUT NAME STRING";
132:   SocketDef "SOCKET NAME STRING";
133:   ( Variables VAR_DOMAIN; )* (19)
134:   ( TestInstances { ( instance_stmt)* } )
135:
136:   BinDefs < Pass | Fail > { (20)
137:     SoftBin BIN_NAME { ??? }
138:     HandlerMap MAP_NAME { ??? }
139:     Axis AXIS_NAME {
140:       SoftBin BIN_NAME

```

```

138:     HardBin integer ...
139:   } // end Axis
140: } // end BinDefs
141:
142:   EntryPoints {
143:     ( OnException execute_stmt )
144:     ( OnFinish execute_stmt )
145:     ( OnLoad execute_stmt )
146:     ( OnLotEnd execute_stmt )
147:     ( OnLotStart execute_stmt )
148:     ( OnMultiSiteDisable execute_stmt )
149:     ( OnMultiSiteEnable execute_stmt )
150:     ( OnPatternLoad execute_stmt )
151:     ( OnPowerDown execute_stmt )
152:     ( OnReset execute_stmt )
153:     ( OnSiteEnd execute_stmt )
154:     ( OnSiteStart execute_stmt )
155:     ( OnStart execute_stmt )
156:     ( OnUnload execute_stmt )
157:     ( OnWaferEnd execute_stmt )
158:     ( OnWaferStart execute_stmt )
159:   } // End Entry Points
160: } // end TestProgram

```

(21)

=====

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 TestMethod, TestModule, 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) *action_stmt*

(8) *exit_port_stmt*

(9) 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.

(10) TestMethodDefs

A test-method is a function 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 methods.

(11) **Inherit:** The Inherit statement is used to import definitions from other like blocks - TestMethodDefs, TestModule, 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.

(12) **Parameters:** This optional block is used to specify the parameters that are to be passed into the containing block (either a testmethod, 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.

(13) TestModule:

(14) **TestModule -> Variables:** The Variables statement within a TestModule block is used to reference a block of variables that are to be local to this TestModule 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.

(15) **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 TestModule, TestFlow, or TestInstance). Or, TestExec can be used to execute an externally created function (i.e., a test method) in which case the function name and formal parameter list shall be defined in a TestMethodDefs 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 TestModule 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-method 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 Test-Method execution), then those changes remain in effect for the life of that object - TestFlow, TestModule, 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 TestModule, 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.

(16) TestFlow:

(17) **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.

(18) TestProgram

(19) **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 TestModule 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.

(20) BinDefs

(21) EntryPoints