Test Gross_Functional {
    Result = Expr { String = "#"; Mode = Output; }
    LoopExpr = Expr { String = "FALSE"; }
    LoopNotify = True;
    LoopDepth = Outside;
    Mask[0] = DC_Easy;
    Mask[1] = ACRelaxed;
    Entry[0] = Functional_Levels;
    Entry[1] = Functional_PatSeq;
    PortExpr[0] = Expr { String = ".Result = tm_rslt:PASS"; }
    PortExpr[1] = Expr { String = ".Result = tm_rslt:FAIL"; }
    Title[0] = FuncTest;
    TestMethod = Ftest;
    Test_enable[0] = Expr { String = "Seq_en:DEFAULT_EXECUTION"; }
    Test_pins[0] = Expr { String = "all_pins"; }
    Test_result[0] = Expr { String = ">#"; Mode = Output; }
    Pattern_index[0] = Expr { String = "Functional_PatSeq.Thread.Single"; }
    Simulate_results[0] = Expr { String = "tm_rslt:FLOW_SIM_OFF"; }
}

SubFlow MainFlow_OnStart {
    Node[17] = FlowNode_ {
        Port[0] {
            To = 9;
        }
        Port[1] {
            To = 18;
        }
        SpecPairs {
            ACSpec = Expr { String = "ACSpec.ACTiming"; Type = INTEGER; }
            DC_Spec = Expr { String = "DC_Spec.T_25dC"; Type = INTEGER; }
        }
        TestId = "42";
        PortSelect = "Gross_Functional.ExecResult";
        Exec = Gross_Functional;
    }
    ... 
    ...
}

Fig. 1: LTX Envision syntax showing Test block and embedded TestMethod
Fig. 2: LTX Envision TestTool display showing TestMethods embedded in Test Blocks
The ETest object may contain any of the items listed below, with the test object execution algorithm as follows:

- Execute any flownode Spec Categories.
- Execute any global Spec Masks defining the specification parameters to be used.
- Execute/prepare any Levels, Pattern Sequence, and Pin State global Entry Objects.
- Execute any Microflow and Cadence Routine global Entry Objects.
- Execute all Enabled Test Blocks, from top to bottom, until a block failure occurs and ContinueOnFail for a failing block evaluates to FALSE. Within each executed test block, the following happens:
  - Execute/prepare any Levels block Entry Objects.
  - Execute any Microflow and Cadence Routine block Entry Objects.
  - Evaluate any block Entry Expressions.
  - If a Test Method is specified, execute it with its associated arguments, updating the block and any output method arguments upon returning from the Method.
  - Evaluate any block Exit Expressions.
  - Execute/prepare any Levels block Exit Objects.
  - Execute any Microflow and Cadence Routine block Exit Objects.
- Update the overall Test Result from all executed test block Results.
- Execute/prepare any Levels, Pattern Sequence, and Pin State global Exit Objects.
- Execute any Microflow and Cadence Routine global Exit Objects.
- Determine the exit port from the exit Port Expressions.

**Test Result**

This cell, in the upper right corner of the window, displays the overall tm_rslt result of the executed test. This result is accessible using the `.Result` expression. The result displayed is derived according to the following rules:

- Only Enabled test blocks are considered in the evaluation. If no test blocks are enabled, the test result will be `tm_rslt:NO_TEST`.
- The result of the test will be the highest valued Result of all evaluated test blocks. The results, in ranked in ascending order, are as follows:
  - `tm_rslt:NO_TEST`
  - `tm_rslt:PASS`
  - `tm_rslt:FAIL`
  - `tm_rslt:DUT_SETUP_FAIL`
  - `tm_rslt:SEARCH_FAIL`
  - `tm_rslt:PARAM_FAIL`
  - `tm_rslt:TIME_OUT_FAIL`
  - `tm_rslt:REPAIRABLE`

This output field is editable, but normally contains “#” as the expression string.
Test_Spec:
user_procedure = "simulate_test_time 3";
end

testfunctions
tf_1:
testfunction_description = "continuity1.cont";
testfunction_parameters = "continuity;@;0;uA; -80;mV; 200;mV; 800;mV; 900;ms; 4:0;Continuity (SP):mV;;1;"
end
testmethods
tm_3:
testmethod_class = "Test.test1";
testmethod_name = "Method1(STRING, INT, INT)"
testmethod_parameters = "@:io_out:<~/dec_27.log;20:1:;:";
testmethod_limits = "";
end
test_suites
Continuity:
override = 1; override_tim_equ_set = 1; override_lev_equ_set = 2; override_tim_spec_set = 2; override_lev_spec_set = 2;
override_timset = 1; override_levset = 1; override_seqlbl = "gross_func";
override_testf = tf_1;

Functional1:
override = 1; override_tim_equ_set = 1; override_lev_equ_set = 2; override_tim_spec_set = 1; override_lev_spec_set = 2;
override_timset = 1; override_levset = 1; override_seqlbl = "gross_func";
override_testf = tm_3;
end
test_flow
run_and_branch(continuity) then
{
}
else
{
    stop_bin "10", "", bad,noreprobe.red, , over_on;
}
run_and_branch(functional1) then
{
}
else
{
    stop_bin "20", "", bad,noreprobe.red, , over_on;
}
run_and_branch(simulate_test_time__2_) then
{
}
else
{
    stop_bin "20", "", bad,noreprobe.red, , over_on;
}
stop_bin "1", "", good,noreprobe.green, , over_on;
end

Tests with embedded TestFunctions or TestMethods

TestFunction or TestMethod embedded in Test. Can change TestMethod or TestFunction without changing outer level test constructs

Return values from TestFunctions, TestMethods, and UserProcedures:
CI_CALL_PASS = 0
CI_TEST_PASS = 0
CI_TEST_FAIL = 1
CI_CALL_ERROR = 2
CI_CALL_BREAKD = 3

UserProcedure called from testflow.

Fig. 3: Verigy SmarTest syntax showing TestSuite and embedded TestFunction or TestMethod
Fig. 4: Inovys/Verigy Stylus syntax showing TestParams block with embedded TestMethods.

Result = Pass or Fail
Within a TestParams block, execution continues until a TestMethod returns a failure.
segment BEGIN {
  ICON = "goseg_c",
  TOOL = "codetool",
  TEST_TYPE = 230,
  RETURN = {
    [ POS = E42, OTHERWISE, Function_4 ]
  }
}; /* end of SEGMENT BEGIN */

segment Function_4 {
  <03:31:1992 12:08:38>,
  X_POS = 134,
  Y_POS = 68,
  ENTRY = W58,
  FUNCTION = initialize(shl_test_cond),
  ICON = "functionseg_c",
  TOOL = "codetool",
  RETURN = {
    [ POS = E56, FAIL, DC_1 ],
    [ POS = E26, PASS, DC_1 ]
  }
}; /* end of SEGMENT Function_4 */

segment DC_1 {
  ICON = "dcseg_c",
  TOOL = "dctool",
  TEST = vdd_opens_cont,
  RETURN = {
    [ POS = S38, FAIL, End_1, BINS = fo_signal_opens ],
    [ POS = E41, PASS, Test_2 ]
  }
}; /* end of SEGMENT DC_1 */

segment Test_2 {
  ICON = "ftestseg_c",
  TOOL = "ftesttool",
  TEST = nom_functional_test,
  RETURN = {
    [ POS = S41, FAIL, End_1, BINS = fn_nom_func ],
    [ POS = E38, PASS, End_1, BINS = p_good_part ]
  }
}; /* end of SEGMENT Test_2 */

segment End_1 {
  FUNCTION = END_OF_TEST(),
  ICON = "stopseg_c",
  TOOL = "leveltool",
  TYPE = ENDTYPE,
  POWER_DOWN = powerdown
}; /* end of SEGMENT End_1 */

Tests with method as part of test. Cannot change TestMethod without changing test type.

dctest vdd_opens_cont {
  <03:31:1993 08:29:32>,
  PINDEF_TABLE = mh1ps,
  LEVELS = cont_levels,
  PRIOR_EVENT_INIT = FALSE,
  DEVICE = VOH,
  METHOD = STATIC,
  MODE = V_MEASURE,
  FORCE = 100uA,
  CLAMP_LO = -120mV,
  CLAMP_HI = 1.5V,
  MIN = OMIT,
  MAX = 1V,
  TIMEOUT = 100.000ms,
  PINS {
    NAME = ALL_SIGS
  }
}; /* end of TEST nom_functional_test */

Fig. 5: ITSS9000 ASAP syntax showing Test block
(test “method” is determined by the type of test – i.e., dctest or ftest)

0 = Fail (unless all port are pass is set), 1 = pass, other than 0 or 1 can be used, but meaning is user-defined.
0 = Pass, 1 = Fail, other than 0 or 1 can be used, but meaning is user-defined.

PASS: executed and the result was pass. (1)
FAIL: executed and the result was fail. (2)
FAILED_TO_RUN: got error(s) during execution. (3)
NOT_TESTED: not executed yet. (4)
OTHER: non-standard test-specific status. (5)

Fig. 6: OTPL syntax showing Test block
(test method is determined by the type of Test – i.e., DCParametricPMUTest or FunctionalTest)
Proposal for P1450.4:

The PostActions block of the Test is used to determine the Test result (ExecResult field). In order to include a function call in the test pass/fail decision, that function must have at least one parameter – an output parameter whose value is set by the function call, and which the user can use to help determine the test pass/fail result. For instance, this output parameter can be an integer whose value is essentially the function return value (and which could be used to directly set ExecResult of the calling test), or it could be

FunctionDefs {
    MyFunction {
        Parameters {
            Out Integer Result;
            Out float meas_value;
        }
    }
}

TestType MyFuncCallTestType {
    Variables {
        Float meas_result_for_caller;
    }
    // How do we specify actual output variables to be used in Function calls?
    FuncExec MyFunction {
        ExecResult;
        meas_value;
    }
}

TestType MyFuncCallTestType_cmp_against_meas_limits {
    Variables {
        Float meas_result_for_caller;
    }
    FuncExec MyFunction {
        ExecResult;
        meas_value;
    }
    PostActions {
        if (meas_value > lower_limit && meas_value > upper_limit) {
            ExecResult = Pass;
        }
        Else {
            ExecResult = Fail;
        }
    }
}