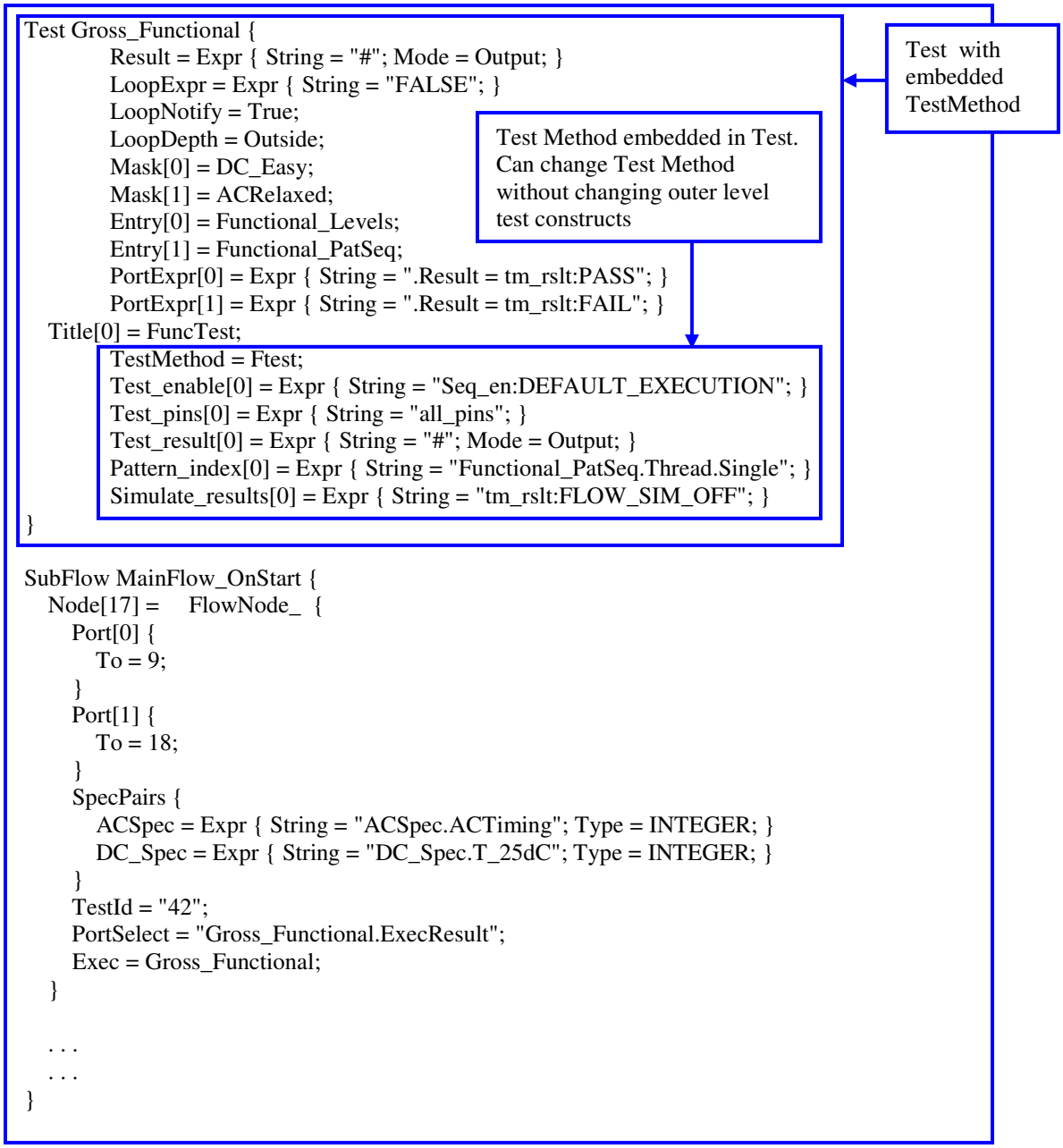

Fig. 1: LTX Envision syntax showing Test block and embedded TestMethod

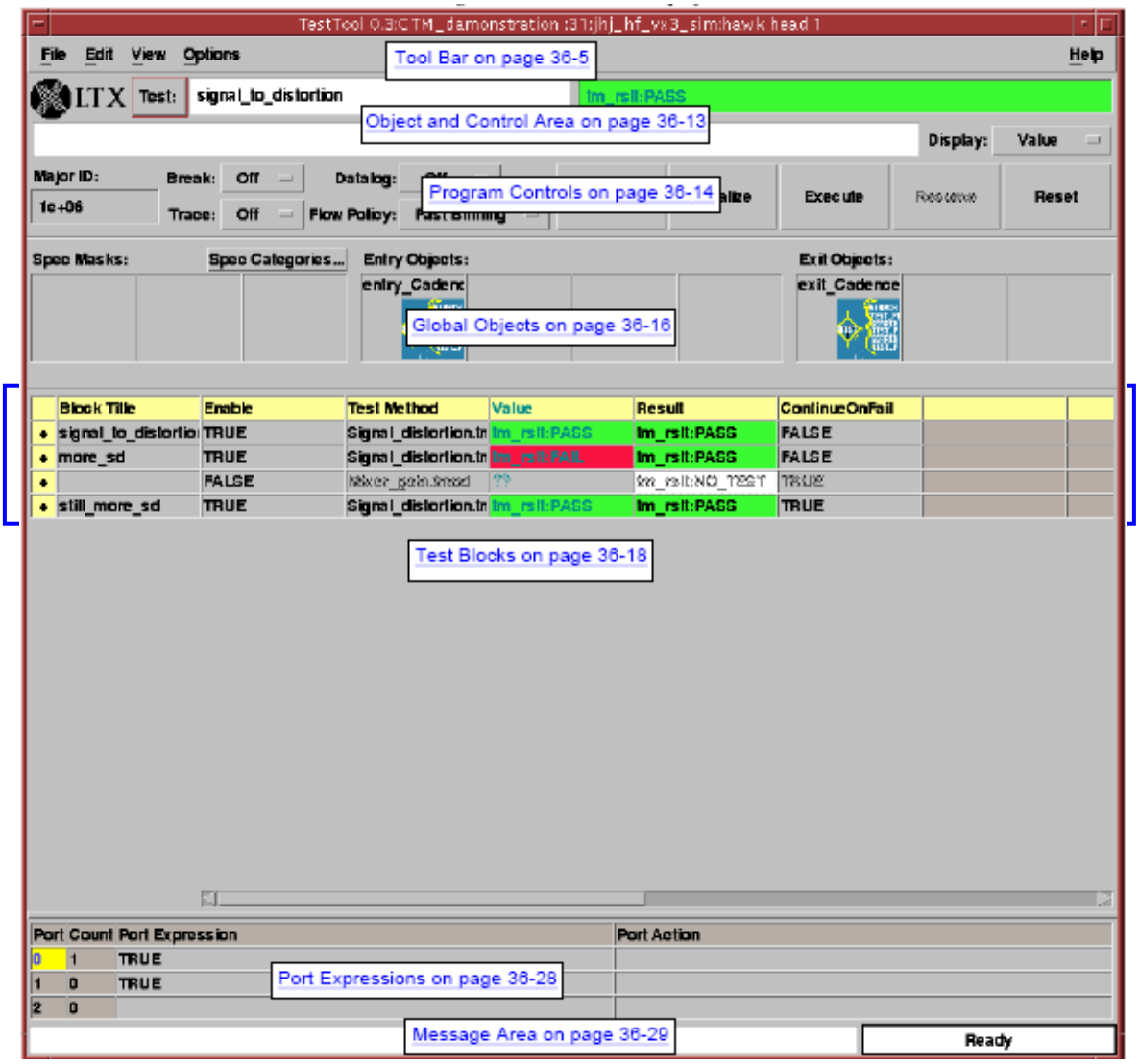

Fig. 2: LTX Envision TestTool display showing TestMethods embedded in Test Blocks

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 EntryObjects.
- 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 EntryObjects.
    - Evaluate any block Entry Expressions.
    - If a Test Method is specified, execute it with its associatedarguments, 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 ExitObjects.
- 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 **Enable**d 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.

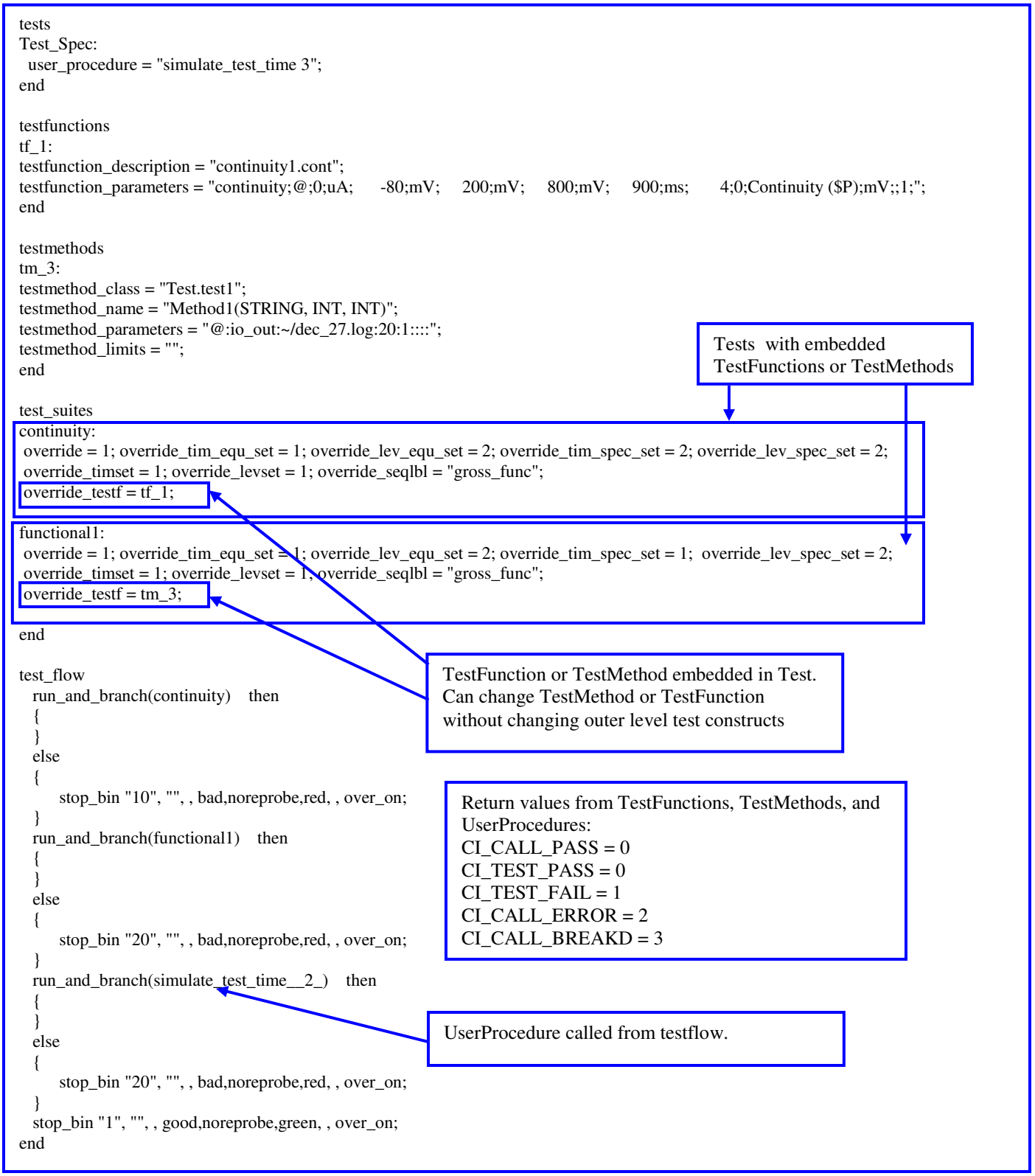

Fig. 3: Verigy SmarTest syntax showing TestSuite and embedded TestFunction or TestMethod

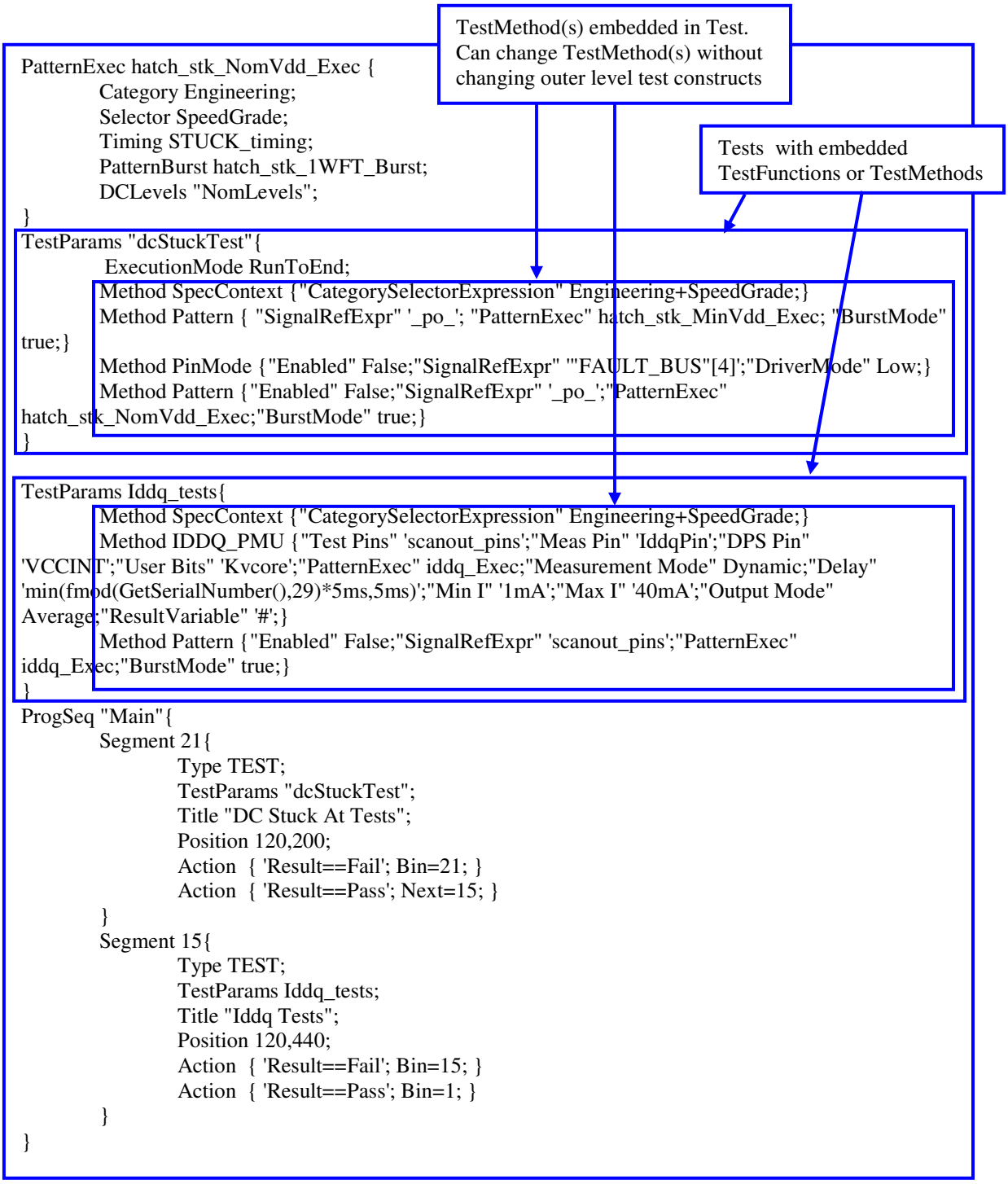

Fig. 4: Inovys/Verigy Stylus syntax showing TestParams block with embedded TestMethods.

Result = Pass or Fail
Within a TestParams block, execution continues until a TestMethod returns a failure.

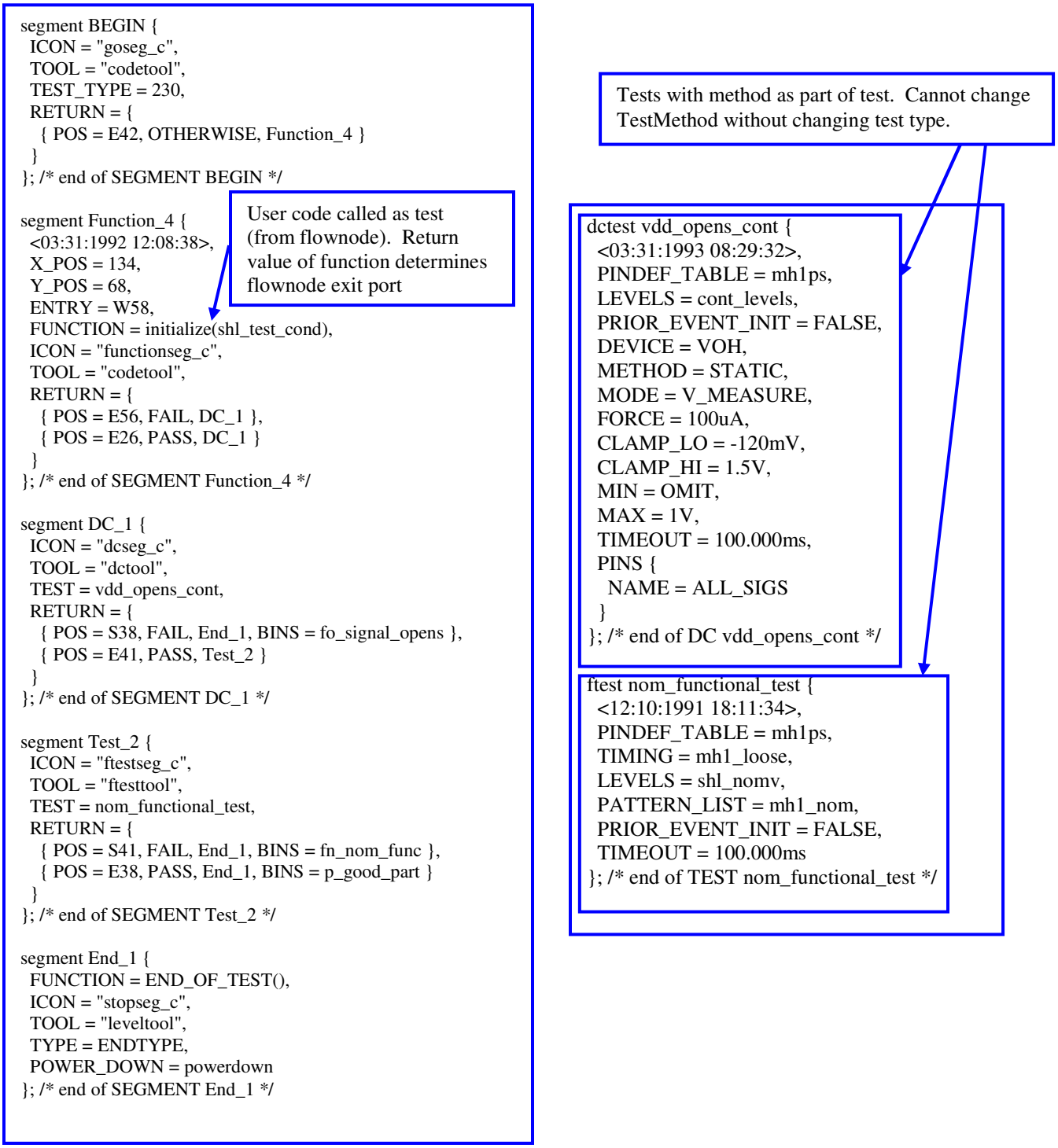

Fig. 5: ITSS9000 ASAP syntax showing Test block
(test "method" is determined by the type of test – i.e., dctest or ftest)

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

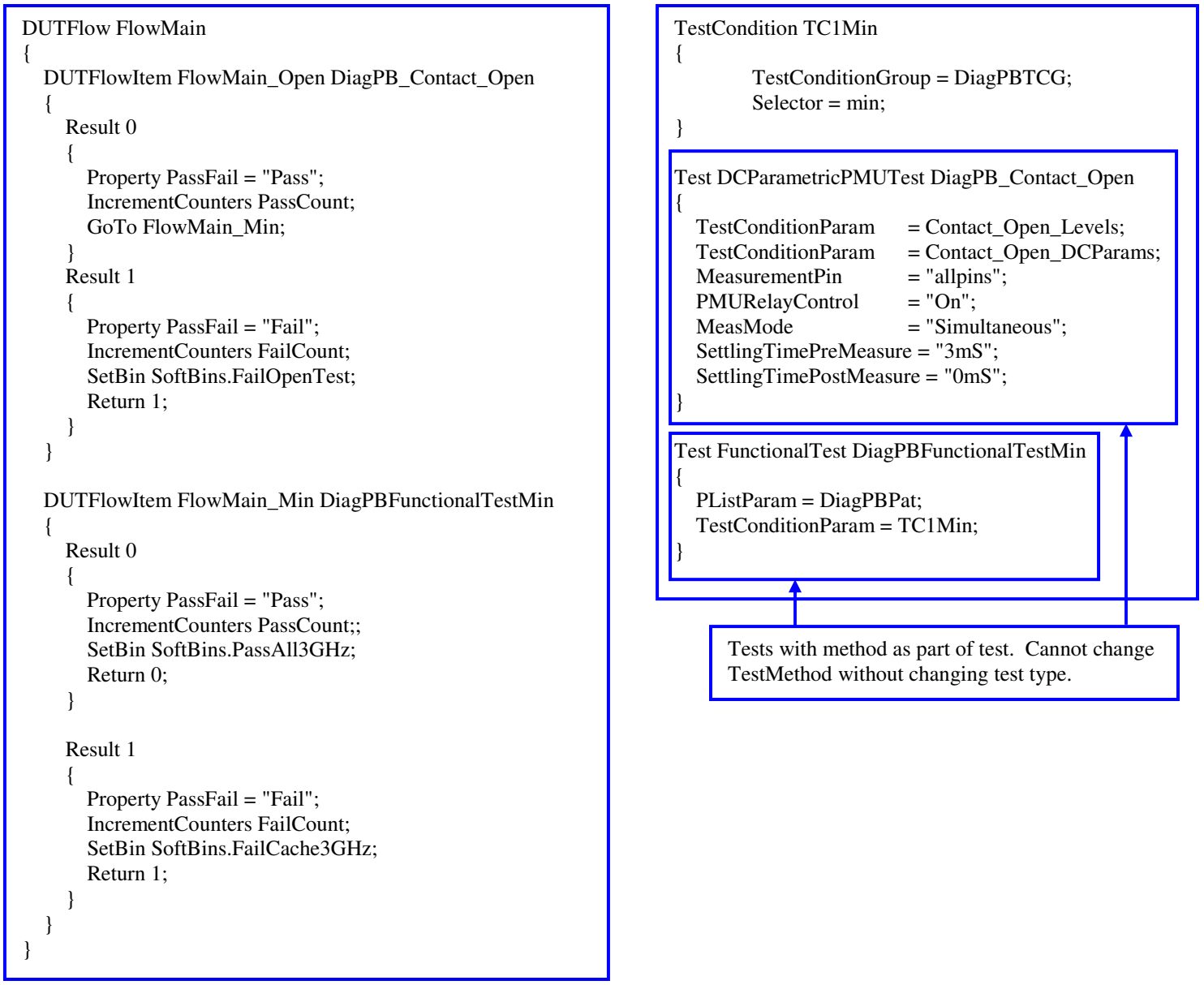

Fig. 6: OTPL syntax showing Test block
(test method is determined by the type of Test – i.e., DCParametricPMUTest or FunctionalTest)

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 reuslt 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)

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;
                            }
                        }
}
```