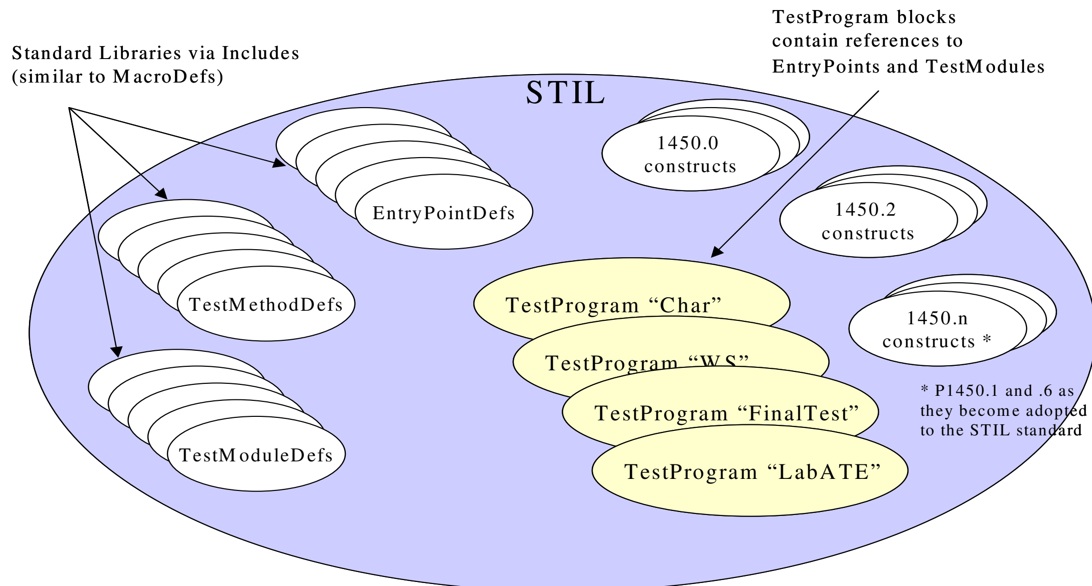
P1450.4 Test Program Flow Conceptual Model Discussion Document

1.0 Top-Down Conceptual View

The P1450.4 constructs will reside within a IEEE 1450-1999 STIL Std file. Its top level constructs will add to the STIL top level constructs/keywords. Figure1 shows the TestProgram block, TestModuleDefs, TestMethodDef and EntryPointDefs blocks. The "Defs" blocks contain all definitions of their respective definition blocks and parallel the MacroDef block in the 1450.0 STIL standard construct.

FIGURE 1. Top Level Test Program Flow Constructs Diagram



2.0 Test Program Flow Extension Terms

2.1 TestProgram Block:

The top level test program construct. There can be one or more. One may be global (unnamed). There may be one or more named TestProgram blocks in a STIL file.

2.2 Flow Block (or TestFlow):

This is the top level program flow construct. There can be one unnamed Flow block. There can be one or more named Flow blocks. This block contains definition of flow and bin entities that make up a given test program flow.

2.3 EntryPoint:

This is a reference to a special program level task activated by the tester (tester operating system, system interrupts, etc.) This entry point references a TestModule. There is a general set of EntryPoint entities defined by this extension (i.e. OnStart, OnReset, etc.). These can be named and one instance of each can be unnamed and treated as global to any Flow

block that does not declare a named one of each type. When a flow is active and a tester event requires an EntryPoint response, the associated EntryPoint TestModule/FlowModule that was declared, or the default if not declared is run.

2.4 TestMethod:

This represents a test type which when instantiated, becomes a TestModule. There are two kinds of types: integral and user defined. A user-defined TestMethod may be composed from a combination of integral and other user defined types. The means by which integral and user defined types are combined to form a new type is TestMethod Flow, the only concrete primitive TestMethod defined by P1450.4 (the other TestMethod, Harness, is abstract).

Integral types are sub-divided into primitives and purely derived types. All types are derived from "Harness", a base class which represents the common denominator (data and functions) between all TestMethods. An example of a primitive might be ForceMeas or VOH.

User-defined types are divided into combinatorial and purely derived types. An example of a combinatorial might be a vol/voh test performed both functionally and parametrically. An example of a purely derived type might be TopLevelFlow.

2.4.1 "TestMethod Harness" (abstract base class)

This represents the common denominator of all TestMethods, i.e., each TestMethod definition includes a single "Harness" component, explicitly or implicitly, but a TestMethod of purely type "Harness" can not be instantiated. Here are some proposed elements:

- Test id (a named data member and not a value)
- 0+ parameters/arguments: input, output, ioput, private (local)
- Ports: entry, exit (pass/fail), and associated actions:
  - variable assignment (conditional/unconditional)
  - bin (conditional/unconditional)
  - stop (conditional/unconditional)
  - skipTestAndActions (conditional/unconditional, entry action only)
  - actionlist (conditional/unconditional)
- Fail flag
- Result: scalar, array (Don Organ: should be typed, e.g., Volts, Seconds)
- Default fail bin (data container or an actual data item)

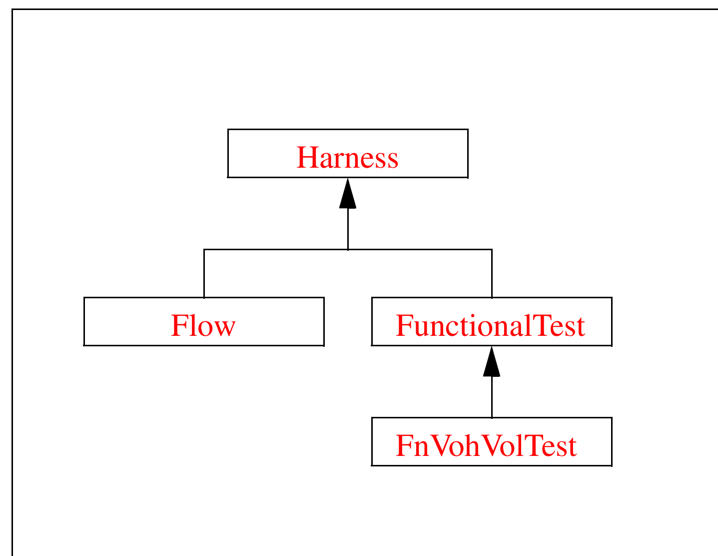
This is an informative term that is not intended to have a keyword in the extension language. It is a common denominator descriptor for all TestModule instantiations. Its use refers to a data type (or object type) from which the various types of TestModules (shown in figures 3 and 5) are derived.

2.4.2 Defaults

Per type defaults provide the ability to have clear and brief STIL flow descriptions where intent might otherwise be obscured by specification of redundant detail.

The types for which defaults are provided are FlowNode, Harness, and individual integral and user-defined TestMethods (recall that Flow is an integral TestMethod). User-defined defaults for these types, if present, override STIL P1450.4 defaults. Defaults are applied to a FlowNode or TestMethod when it is instantiated (a TestMethod becomes a TestModule at this point).

There are many types of TestMethods of which only the integral types are known to STIL and of those, only Flow is known to P1450.4, so we require a mechanism for providing defaults for as yet undefined TestMethods. The inheritance hierarchy provides that mechanism. P1450.4 needs to provide defaults for Harness only, since it is the base class of all TestMethods. Its defaults trickle down to the derived TestMethod unless the derived TestMethod overrides. Given that Harness is at the root, the defaults of the more distant TestMethod override. Overrides take place on a per white box basis (see TestModule in figure 7) namely, TestPreActions, TestPostActions, TestArbiter, PassActions, and FailActions.



For example, if the defaults for a particular TestMethod, e.g., FunctionalTest, describes FailActions only, it overrides base class FailActions but still inherits the others. If another TestMethod is derived from FunctionalTest, e.g., FnVohVolTest, it inherits defaults from its nearest ancestor.

Since there is only one type of FlowNode, FlowNode defaults apply to all implicitly instantiated FlowNodes and all explicitly instantiated FlowNodes which lack pre or post section descriptions. These descriptions refer to the contents of the white boxes in figure 6

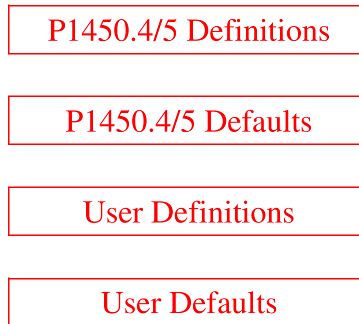
namely, PreActions, PostActions, Arbiter, and ExitActions 1 through n, and are applied on a per box basis, similar to the TestMethod defaults.

For example, the FlowNode default description may specify two exit actions, one called Pass and one called Fail, all boxes including Pass and Fail having a no-op content.

P1450.4 defaults are designed to provide a best common denominator from which production testflows for current ATE can be generated:

- FlowNode defaults:
  - Exit ports: pass and fail.
  - Actions: none on entry, post, pass, and fail.
  - Arbiter: exit via fail port if test object fails, exit via pass port otherwise.
- Harness defaults:
  - Arbiter: exit via fail actions if fail flag is set, exit via pass actions otherwise.
  - Test id: empty string.
  - Parameters/arguments: none.
  - Ports:
    - Entry actions: none.
    - Exit actions:
      - Post: none.
      - Pass: none.
      - Fail: bin with stop if any test failed.
  - Fail flag: false.
  - Result: scalar = NaN (zero dimension array).
  - Default fail bin: undefined (equivalent to *no bin*).
- Flow defaults:
  - Actions: inherit Harness defaults.
  - Arbiter: exit via fail actions if any directly referred or contained test objects' fail flag is set, exit via pass actions otherwise.
- TestMethod defaults:
  - Integral: all except TestMethod Flow and Harness are defined by P1450.5.
  - User-defined: defined by user.

FIGURE 2. TestMethod Parameter Default Assignment:



With regard to TestMethod parameter assignments, any parameter not assigned a default value during TestMethod definition must be assigned a value during Defaults specification. Defaults apply only to parameters left unset during TestMethod definition. User defaults override P1450.4/5 defaults.

FIGURE 3. Harness Abstract Base Class Example

```
Abstract Harness
{
  (In      (<modifier>)? <datatype> <id> (<array_size>)? = &<symbol_id>|<expr>;)*
  (InOut   (<modifier>)? <datatype> <id> (<array_size>)? = &<symbol_id>;)*
  (Out     (<modifier>)? <datatype> <id> (<array_size>)? = &<symbol_id>|<expr>;)*
  (Private (<modifier>)? <datatype> <id> (<array_size>)? = &<symbol_id>|<expr>;)*
  Ports
  {
    Entry
      {
      (Actions { NoPostActionExitIf(<bool_expr>)|<preactions>; })*
      }
    Exit
      {
      Condition Fail <bool_expr>
        {
        (Actions { <postactions>; })*
        }
      Condition Pass // No <bool_expr> (necessary|permitted) on last Condition
        {
        (Actions { <postactions>; })*
        }
      }
  }
}
```

FIGURE 4. Example View of an Instantiation of the Abstract Harness- Test Method Flow

```
TestMethod Flow
{
  <harness_instantiation>
  (FlowNode <flownode_id>*
  {
    (Execute
      ((<test_object_id>
          {
          (In    <id> = Default|&<symbol_id>|<expr>;)*
          (InOut <id> = Default|&<symbol_id>;)*
          (Out   <id> = Default|&<symbol_id>|<expr>;)*
          })|
      ((<method_id>  // Anonymous instantiation of Method
          {
          (In    <id> = &<symbol_id>|<expr>;)*
          (InOut <id> = &<symbol_id>;)*
          (Out   <id> = &<symbol_id>|<expr>;)*
          })
    )
    (Ports
      {
      (Entry
        {
        (Actions { ((NoPostActionExitIf(<bool_expr>) <condition_name>)|<preactions>)? })?
        }
      )*
      (Exit <condition_name>
        {
        (Condition <bool_expr>;)?
        (Actions { <postactions> })?
        (NextNode  <port_connection>;)
        })+
      })
  })*
}

       Entry {}
       Exit Fail
           {
           // Method Harness: same as ThisTest.failed, Method Flow: (test1.failed || test2.failed .... || testN.failed)
           Condition AnyTest.failed;
           Actions { ThisTest.BinFail }
           }
       Exit Pass {}
    }

    Method Flow             // Integral type
    {
       IsA Harness;         // Explicit statement not necessary: implicit as per syntax
       Fail.Actions {}      // Override inherited Fail Actions **** syntax description missing ****
    }

    Method TopLevelFlow     // Statement not necessary: currently same as integral type
    {
       IsA Flow;            // Inherited overridden Fail Actions
    }
}
```

FIGURE 5. Defaults

2.5 TestModule:

This is an instantiation of a TestMethod.

(See Figures 3, 4 and 5)

2.6 FlowNode:

(See Figure2) A node in the program flow that contains a ModuleRef (Body) that references a TestModule or FlowModule. This node has PreActions that defines the entry point into the node and may contain actions, declarations such as Spec/Category selection, etc. Absence of actions may in the Pre section may cause default actions (tbd). The Post section contains PostActions, Arbitrator, and ExitActions. The ExitActions give directives as to the follow-on flow path taken out of the FlowNode.

2.7 BinNode and BinMap:

Not yet defined/discussed

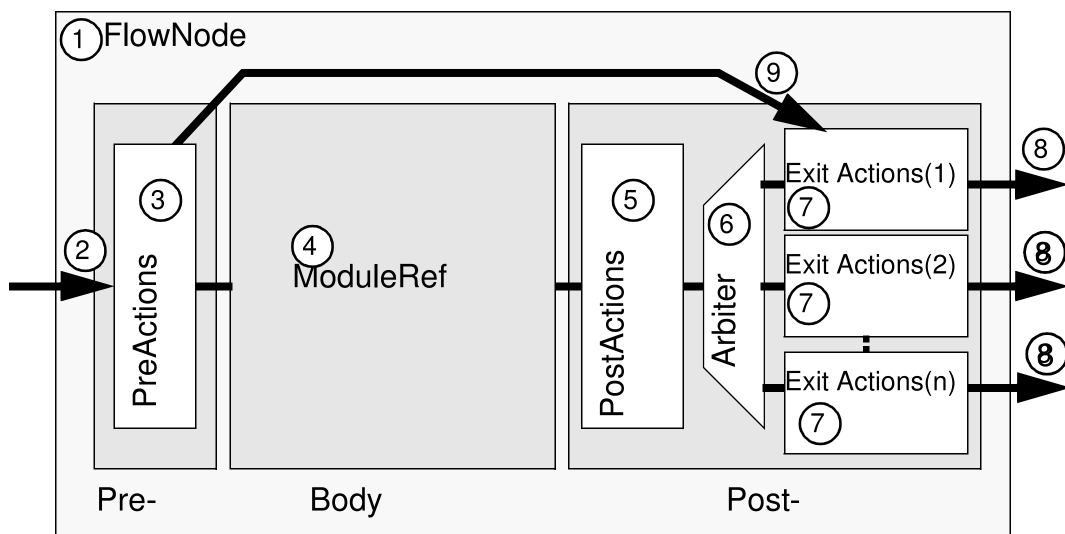
(Need two natures: terminal and flow-through)

2.8 TaskNode and DecisionNode:

These are non-test type nodes used for non-test activities and flow decision content.

3.0 FlowNode Conceptual Model

FIGURE 6. The FlowNode Conceptual Model Diagram (with its named components)



3.1 FlowNode Components Descriptions

1. FlowNode

2. EntryPath

3. PreActions Block

4. ModuleRef (Module Reference)

5. PostActions Block

6. Arbiter Block

7. ExitActions Block

8. ExitPath

9. SkipPath (can goto to any ExitAction Block)

3.2 FlowNode Informative Term Descriptions

1. “Pre-” portion

2. “Body” portion

3. “Post-” portion

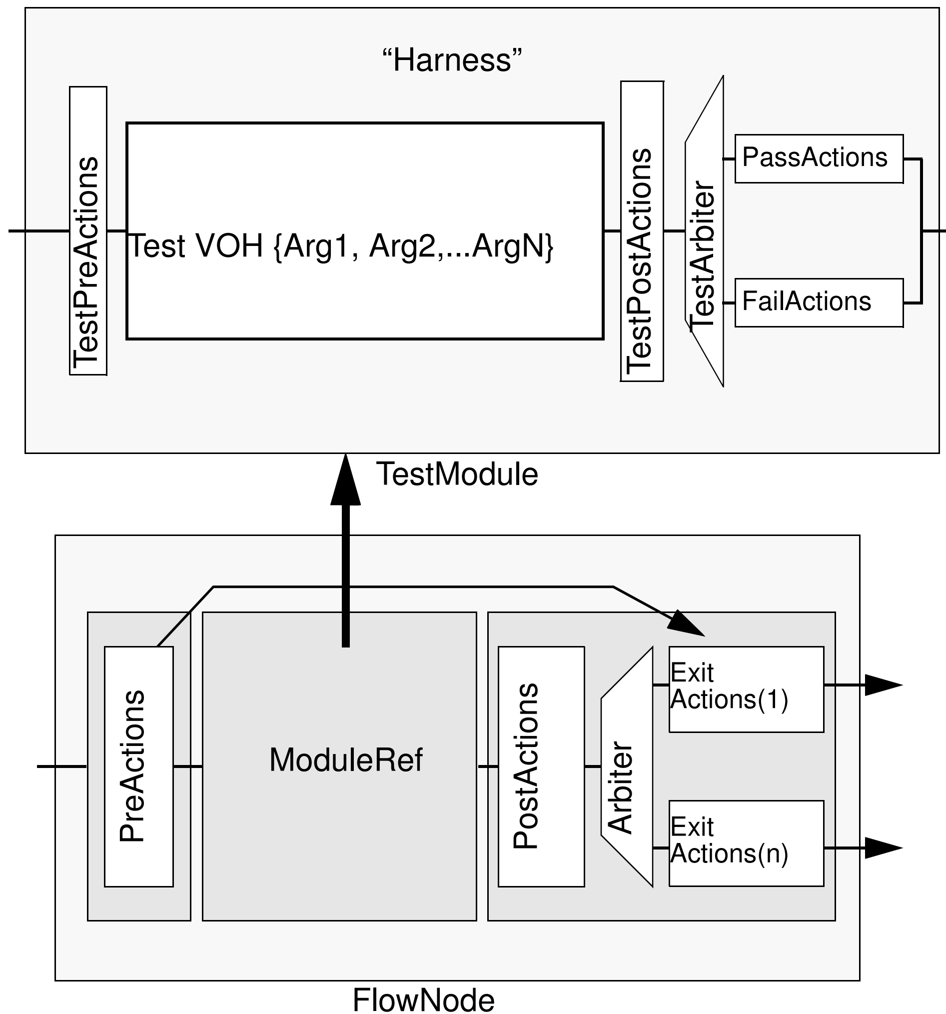
4.0 Relationship of FlowNodes to TestModules

The instantiation of the TestMethod (i.e. VOH) can be “in-line” or “defined-before-use”. “Define-before-use” is the mechanism by which two or more FlowNodes can refer to the same “Test Object” (i.e. Module). Test Module instantiation involves placing the instantiated TestMethod inside the harness as shown to the left (the harness is the grey portion of the box labeled TestModule.)

TestMethod “VOH” {Arg1, Arg2,...ArgN} is the type definition, and

Test VOH {Arg1, Arg2,...ArgN}is the instantiation of the VOH TestMethod in the test module.

FIGURE 7. Block Diagram View of FlowNode Where ModuleRef References a TestModule



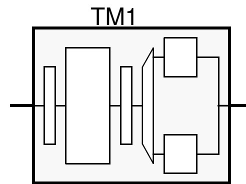
5.0 TestModule Characteristics

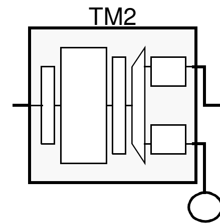
5.1 The "Harness" of the TestModule (a better title will emerge)

Commonalities. What the harness provideds. The FlowNode has dependencies on the the harness. Describing the data interaction model. Perhaps an upper level view of the interface between the FlowNode and the TestModule. Examples. Cummunication with input and output arguments flowing into and out of the TestModules.

5.2 Two Types of "Outflow" Configurations for TestModules

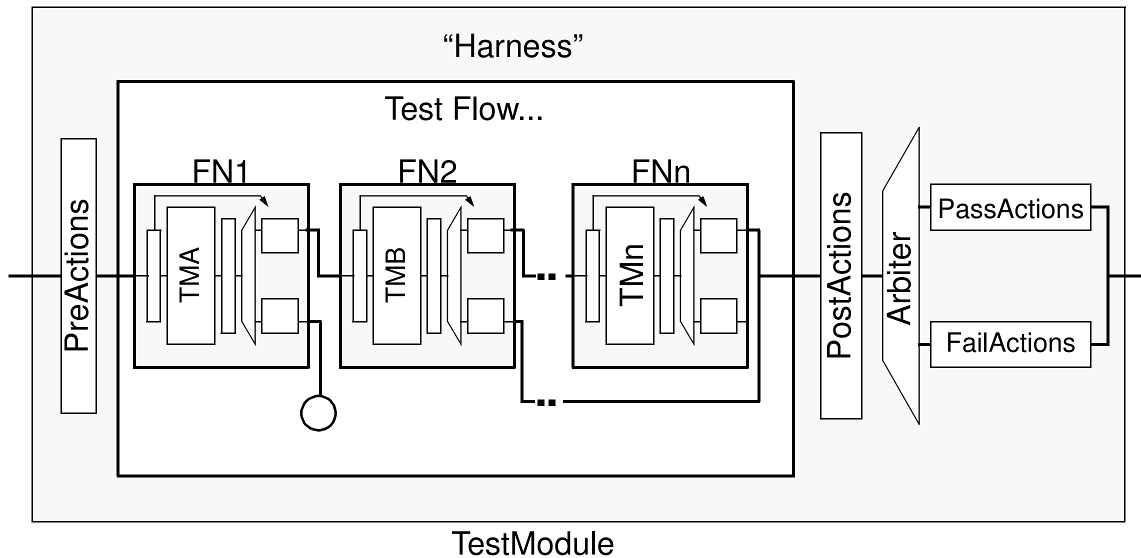
FIGURE 8. Conceptual Block Diagrams of the Two Outflow Types





5.3 TestFlow as the TestModule Body

FIGURE 9. TestModule Referencing a TestFlow



6.0 Binning thoughts

6.1 Guiding Values for Binning

1. Simplicity: Have same mechanism serve "Pass" and "Fail" binning
2.

6.2 Bin Mapping

Single axis for Fail bins

Single-Multiple axis for Pass binning with a BinMapping (expected usage is the multiple axis)

Single axis: Cell number 1 is associated with soft bin 1 cell number 2 is associated with soft bin 2.

There is a notion of “hard bin” and “soft bin”.

Concept of a default mapping with an option to have domain named mapping (per handler with different bins available.)

Expected usage of pass bins, when you fail, registration on an axis... so for example a VOH test fails

One axis is speed binning and the other axis is power supply tolerance. So there is an association of speed testing as a row of an array and a column for Power Supply tolerance. Now that we have two axis, you have an x and a y axis then you can have a cell. Each cell has a counter or (counters?). At the end you would be able to tally the counters to determine the tested performance and then you can determine a

6.3