Summary of Syntax Definitions for STIL-Flow Extensions (P1450.4)

Table 1: History

Date	By	Comments
12/17/98	tony taylor	Original document - created from notes and discussions at STIL extension meetings
1/14/99	tony taylor	Updated as a result of review at P1450.4 meeting on 1/11, 1/12. Many of the syntax constructs were generated by Don Organ at that meeting.
1/27/99	tony taylor	Added a new test case suggested by Ernie Wahl - using multiple categories withing a test node.
2/23/99	tony taylor	Adjustments to syntax to improve clarity. Added ebnf definitions back into the document.
3/8/99	tony taylor	More tweaks to the syntax based on review with Greg Maston, and further discussion via phone conferences: make Axis available to a Test object; removed keyword 'Eval'; added Algorithm to the Plot object; added DCLevel to the Test object; changed Category to a list in the Test object; removed the type definition from the ProgramOption statement; added type definition to Spec variable definition. Added a section defining each of the blocks of the data model.
10/31/02	Don Organ	Starting a major rework from the 2002 reincarnation of the STIL.4 working group.

The P1450.4 standard contains the definition of the data blocks necessary to specify the sequence of activities that are to be performed on each device in order to "test" that device. The definitions that are contained in this document depend on other definitions in the base IEEE-P1450 document as well as Test Method definitions that are being developed in parallel with this work.

Definitions

(from Ernie 4/3- there has been no group consensus on these yet, also a few other definitions are captured at http://65.119.15.228/stil.4/FlowControlIssueResolutions.html)

Testmethod:	parameters and a defined or implied sequence of events yielding a pass/fail judgment and potentially parametric data.
Test:	an object with a pre and two post actions (pass and fail) and a body (the test method instantiation).
Testblock:	A Test whose associated test method consists of executing the flow of Tests/Testblocks inside the block.

Testflow:	The top-level Testblock. The only Testblock not pointed to by a Testflow-node.
Testflow-node:	an object with a pre and multiple post actions and associated ports (port of entry, pass, fail, etc ports) and a pointer to a Test or Testblock.
BinMap:	minimally stores software to hardware bin mappings, i.e., might additionally store alphanumeric alias to bin number mappings.
Variables:	
Mathematical Expressions:	
Logical Expressions:	These form the basis for the following
Assign: conditional/unconditional, pre/post test execution:	assign a value, potentially a mathematical expression, to a variable potentially based on the evaluation of a logical expression.
Bin: conditional/unconditional, pre/post test execution:	set a soft bin potentially based on the evaluation of a logical expression.
Execute conditional, pre test execution:	execute the test if the associated logical expression evaluates as true (unconditional test execution is the default).

New Syntax proposal (far from complete)

```
FlowGroup[1] FlowGroupName {
        (In <input_fg_argument_name> = 'default_expr';)*
        (Out <output_fg_argument_name> = 'default_expr';)*
        (If 'boolean_expr' Return;)*[2]
        (PreActions actions_to_be_defined*;)[3]
        (FlowNode FlowNode_name {
                (If 'boolean_expr' GotoPort <port_label>)*[4]
                (PreActions actions_to_be_defined*;)
                (Execute[5]
                        TEST_OBJECT_NAME
                        | (TEST_METHOD_NAME {
                                (In <input_argument_name> = 'expr';)*
                                (Out[6] <output_argument_name>;)*
                                }) // end of test method
                        | (FlowGroup_Name {
                                (In <input_argument_name> = 'expr';)*
                                (Out <output_argument_name>;)*
                                }) // end of flow group
                ) // end of Execute
                (PostActions actions_to_be_defined*;)[7]
                ( Ports[8]  {
                        (<port_label>:?
                                'boolean_expr':
```

1. Ernie's model also had a Flow contruct - which, so far, we don't have here, since a Flow seems to be the same as a FlowGroup. In Ernie's model, the Flow and the FlowGroup differ in default behaviors (implemented as virtual functions in the underlying C++) - these differences may show up as different defaults - once we define the defaults.
2. Two questions here: a) Potentially, this If...Return could be one of the PreActions - and therefore a separate syntax entry is not necessary. b) There is a question about the multiplicity; should there be a '*' here?
3. Three questions: a) The actions probably include: i) assignment to a variable (may be conditional) and ii) jump to the port_label. b) Again a question about the multiplicity; should there be a '*' here? c) Should this be a place where "user code" could be inserted or referred to?
4. Same questions from If and PreActions here in the FlowNode - as for in the FlowGroup.
5. Several issues here: a) Execute might not be the best choice for the name, but let's stay away from overloading the word "test" here. b) There are two somewhat contradictory desires. First, would like to be able to refer to an external TestObject (which could also be referred to from other places). Second, in some cases, it is desireable to avoid the extra level of indirection via the TestObject - and just reference the test (TestMethod of FlowGroup) direction. c) The intent is that the TEST_METHOD_NAME and the FlowGroup_Name have the same syntax - Ernie's model shows that a Flow is derived from Test (i.e. object-oriented inheritance). There is a question as to what "inheritance" means from a syntax perspective. d) Multiplicity - should we allow multiple Execute blocks? or multiple references to the TEST_OBJECT_NAME, TEST_METHOD_NAME, or FlowGroup_Name from within an Execute?
6. Is it legal to have an Out argument with the same name as an In argument? If so, is it an In/Out argument? This may have some semantic issues associated with default values, assignments, value lifetime (when it is acceptable to access a value), etc.

```
                    (PortActions action_to_be_defined*;)
                    : (Next=FlowNode_Name)
                    | (Bin=Bin_Name)
                    | (ExitFlowGroup);
                )*
            }) // end of Ports
        })* // end of FlowNode
        (PostActions actions_to_be_defined*;)[9]
} // end of FlowGroup
```

FlowGroup: Start of a FlowGroup block. A FlowGroup contains the individual FlowNodes - which are the basic unit of sequencing. A FlowGroup can be "executed" and can accept input arguments and produce output arguments. These arguments may be referred to from any expression within the FlowGroup - including those in the contained FlowNodes.

In, Out (in FlowGroup): Identifies the input and output arguments to the FlowGroup. The input arguments may be assigned to (by name) from the calling FlowNode's Execute block, but are read-only within the FlowGroup (i.e. they can't be modified from within the FlowGroup - or any of the contained FlowNodes). The output arguments may be assigned to from within the FlowGroup (and any of the contained FlowNodes), and are accessible by name from the calling FlowNode's execute block. The optional defaults identify values to be assigned to these arguments if they are not explicitly assigned another value.[10]

If ... Return (in FlowGroup): If the optional boolean expression evaluates to TRUE, then execution skips all FlowNodes and proceeds to the exit from the FlowGroup. Note that the assignment of defaults to the In, and Out arguments occurs before the return is executed.

PreActions (in FlowGroup): Actions (to be defined) that are "executed" before the first FlowNode.

FlowNode: Defines a FlowNode - which is the basic unit of the execution sequencing. Conceptually, the FlowNode serves as the vertex in a directed graph repre-

7. PostActions: similar to PreActions (and similar issues). Also, should binning be a PostAction? Or should it be referred to as something like a FlowNode? Ernie would like the encapsulation of attaching the binning to the cause; separating them may increase maintenance complexities.
8. Numerous issues related to this ports syntax: a) this is a "horizontal" approach - there is also a "vertical" approach (listing all of the arbiters first, then all of the actions and then all of the Next=). b) the syntax - as far as identifying the fields and what to use for separators - is weak. c) the question (associated with the PostActions question above) as to where binning should be. d) also, the ExitFlowGroup statement could be considered as a PostAction. e) defaults - can we balance "keeping simple things simple" with allowing headroom to do more complicated things? We probably want to define defaults so that a passing test goes out a port labeled "pass" and a failing test goes out a port labeled "fail".
9. PostActions here has the similar issues to FlowNode's PostActions.
10. A couple of further questions: a) If the same name is used for an In argument and also for an Out argument - does that make that an In/Out argument? Is this legal? Are special semantics needed (such as when referencing for other expressions)? b) Should STIL.4 define some standard output arguments (by name, type and meaning - such as a pass/fail indicator)?

senting the test program execution.
There may be any number of FlowNodes (including zero) in a FlowGroup. Semantically, the first (if there are one or more) FlowNode has special meaning; it will be the first FlowNode executed when the FlowGroup is executed (unless the FlowGroup's If/Return is true).
The FlowNode_name is optional, but is required if this FlowNode is to be referenced from elsewhere. (An unnamed FlowNode can be executed in two situations: 1) if it is the first FlowNode in a FlowGroup or 2) if the preceeding FlowNode's Exit:Next does not identify a FlowNode_Name).

If ... GotoPort (in FlowNode): If the optional boolean expression evaluates to TRUE, then execution skips the PreActions (if any), the Execute block and the PostActions (if any) and proceeds to the Ports block. The identified Port is taken (that port's boolean expression is not evaluated). That port's PortAction is executed, followed by its action (one of Next=, Bin=, or ExitFlowGroup)

PreActions (in FlowNode): Actions (to be defined) that are "executed" before the ExecuteBlock. (Not executed if Skip is true.)

Execute: Identifies the TestObject, TestMethod or FlowGroup that is to be executed. The FlowNode's syntax and semantics are identical for executing TestMethods and FlowGroups. In either case, the entity to be executed (TestMethod of FlowGroup) is identified by name. Data may be passed into the TestMethod or FlowGroup by identifying In arguments. Data may be retrieved from the TestMethod or FlowGroup by the Out arguments.

In, Out (in FlowNode:Execute): Identifies the input and outputargument namess for passing data into and out of the referenced TestMethod of FlowGroup. It is an error if the argument names identified in the FlowNode's Execute block do not match those of the referenced TestMethod or FlowGroup by name, by direction (input or output) or by type of expression (to be defined - the intention is to be type safe across Volts and time, etc.).

PostActions (in FlowNode): Actions (to be defined) that are "executed" after the Execute block, but before the Exit block. (Not executed if Skip is true).

Ports: Determines what is to be executed next - either another FlowNode in the same FlowGroup, or a terminal Bin, or a return from the FlowGroup to the calling FlowGroup.
Consists of a list of quadruplets. The optional first element of each quadruplet is a port-label. This label may be referred to from the If/GotoPort statement earlier in the FlowNode. The second element of each quadruplet is a boolean expression. Its execution semantics are described below. The third element of the quadruplet is an option list of actions to be executed - these are to be defined. The fourth and final element identifies the next execution point. It can be one of Next, Bin or ExitFlowGroup.
When the Ports block is executed, the boolean expression of the first quadruplet in the list is evaluated. If it is false, then the second quadruplet's boolean expression is evaluated and so on, until either a boolean expression evaluates to true, or there are no more elements in the list.
If the boolean expression evaluates to true, then that quadruplet's Actions

(if any) are executed in the order they are listed. Then, the last element in the quadruplet is examined:
If Next, then the next FlowNode in this FlowNode's FlowGroup is identified to be executed. Next can identify the FlowNode explicitly by name (it is an error if no such FlowNode exists in the containing FlowGroup). If Next does not identify a FlowNode by name (i.e. the FlowNode_Name field is blank), then the subsequent FlowNode in theFlowGroup (i.e. in the order defined in the STIL file) is executed, or it is an error if there is no subsequent FlowNode in this FlowGroup.

If Bin, then the identified Bin is executed and execution terminates.[11]
If ExitFlowGroup then execution jumps to the containing FlowGroup's first PostAction block - if one exists, else to the FlowGroup's Return statement.
The order in which the ports are listed have additional significance in a multisite situation; a port listed before another has a higher priority than the other. In the case of branching, where different sites exit through different ports, the site which follows an earlier port continues testing, while a site utilizing a later port will be suspended.

PostActions (in FlowGroup): Actions (to be defined) that are "executed" after an ExitFlowGroup statement is encountered in a FlowNode in this FlowGroup. (Not executed if If/Return is true).

```
TestMethod TEST_METHOD_NAME {[12]
    (InheritFrom TEST_METHOD_NAME_BASE;)[13]
    (In <input_argument_name> = 'expr';)*
    (Out <output_argument_name>;)*
}) // end of test method
```

TestMethod: Start of a TestMethod block. This block introduces to the STIL reader a TestMethod of a given name. The STIL reader can then check to subsequent references of this TestMethod are done in a consistent manner.

InheritFrom: Indicates that this TEST_METHOD_NAME is to inherit all of the **In** and **Out** arguments from the TEST_METHOD_NAME_BASE.

11.Note, that is assuming all bins are terminal bins, if we want "flow-through" bins, then we could easily allow both a Bin and a Next statement to be identified in the same quadruplet. Ernie has cases where the decision as to whether a bin is terminal or "flow-through" is made at run-time - such as for "continue on fail".

12.This serves as a declaration, similar in intent to a function or a struct declaration in a C/C++ header file.

13.Should there be multiple inheritance?

In, Out:	Identifies the input and output arguments to the TestMethod. The input arguments may be assigned to (by name) from the where the TestMethod is referenced (i.e. either a FlowNode's Execute block or from within a TestObject). The output arguments may be assigned to from within the TestMethod, and are accessible by name externally. The optional defaults identify values to be assigned to these arguments if they are not explicitly assigned a value.

TestObject *TestObjectName* **{**[14]
    (TEST_METHOD_NAME {
        (**In** <input_argument_name> = 'expr';)*
        (**Out** <output_argument_name>;)*
        }) // end of test method
    | (FlowGroup_Name {
        (**In** <input_argument_name> = 'expr';)*
        (**Out** <output_argument_name>;)*
        }) // end of flow group
} // end of TestObject

TestObject:	Defines a test-object such that it can be referenced from multiple FlowNodes. Has the same capabilities as the Execute block in the FlowNode.
In, Out:	Identifies the input and outputargument namess for passing data into and out of the referenced TestMethod of FlowGroup. It is an error if the argument names identified in the FlowNode's Execute block do not match those of the referenced TestMethod or FlowGroup by name, by direction (input or output) or by type of expression (to be defined - the intention is to be type safe across Volts and time, etc.).

TestProgram NAME **{**
    (**Device "***string***";**)
    (**OnLoad** FLOW_GROUP_NAME**;**)
    (**OnInitFlow** FLOW_GROUP_NAME**;**)
    (**OnStart** FLOW_GROUP_NAME**;**)
    (**OnReset** FLOW_GROUP_NAME**;**)
    (**OnPowerDown** FLOW_GROUP_NAME;)
    (**OnMultisiteDisable** FLOW_GROUP_NAME**;**)
    (**OnMultisiteReenable** FLOW_GROUP_NAME**;**)
    (**ProgramOption** 'OPT_NAME = *expr*') ;)*
}* *// end of TestProgram*

TestProgram:	Defines a TestProgram object. The purposes of this TestProgram object are beyond the scope of this standard, but includes the typical case of providing a NAME that a production operator can load and execute.

14.Should a TestObject be able to reference another TestObject? This seems like a general solution (allowing the same capability as is in a FlowNode's Execute block). But does it solve any problems?

Device:	The usage of this string is beyond the scope of this standard - but may include messages to the production operator, and usage in datalogging.
OnLoad:	Identifies a FlowGroup that is to be executed when this test-program is loaded onto the tester. This occurs before exiting the “SETTING UP” State in the TSEM Processing State Model.
OnInitFlow:	Identifies a FlowGroup that is to be executed when the tester receives a “start” signal. This FlowGroup might be typically used to power-up the device-under-test. This occurs after entering the “WORKING” State in the TSEM Processing State Model.
OnStart:	Identifies a FlowGroup to be executed when the tester receives a “start” signal - this FlowGroup is executed after the OnInitFlow (if any). This FlowGroup might be typically used for the main test sequencing for the device-under-test. This occurs within the “WORKING” State in the TSEM Processing State Model.
OnReset:	Identifies a FlowGroup to be executed when the tester receives a “reset” signal. The precise semantics of the reset operation, including when this OnReset FlowGroup is to be executed, is to be documented by the tester vendor. Note that Reset is not contained in the TSEM processing State Model
OnPowerDown:	Identifies a FlowGroup to be executed when returning the device-under-test to the READY state (in the TSEM Processing State Model). This includes after the completion of the OnStart FlowGroup, and after the completion of the OnReset FlowGroup.
OnMultisiteDisable:	Applicable to multisite testing only. Identifies a FlowGroup to be executed on a site or sites as that site (or sites) is temporarily suspended to allow test execution to continue on other sites that have branched differently via the FlowNode’s Ports constructs.
OnMultsiteReenable:	Applicable to multisite testing only. Identifies a FlowGroup to be executed on a site or sites as that site (or sites) is being re-enabled after a temporary suspension as described above under OnMulsiteDisable.
ProgramOption:	To be defined.

Variable Semantics

This specification incorporations three forms of variables - 1) Spec variables from 1450-1999, 2) Program Options, and 3) In and Out arguments in various constructs (FlowGroup, FlowNode, TestObject). Variables of any form may be referenced in the expression associated with any other variable. However, the variables have differences in the following ways: naming, initialization/ default, and access rules.

Variable Naming

Variables are accessed by references to their name from within an expression. Spec-variables' names are described in 1450-1999. ProgramOptions are referenced by their OPT_NAME as declared in the TestProgram block.
The In and Out arguments of FlowGroup, FlowNode and TestObject have hierarchically scoped names.
FlowGroup_Name.input_fg_argument_name
FlowGroup_Name.output_fg_argument_name
FlowGroup_Name.FlowNode_Name.input_argument_name
FlowGroup_Name.FlowNode_Name.output_argument_name
TestObjectName.input_argument_name
TestObjectName.output_argument_name

STIL-1999 supports referencing a Variables (in a Spec block) from an expression. The above syntax adds additional "variables" that can be referenced from expression, but these additional variables do not reside in the Spec block. Instead, the reside in, and are scoped to, the containing FlowNode or FlowGroup block. The variables identified in the FlowGroup (input_fg_argument_name and output_fg_argument_name) may be accessed from any expression within the FlowGroup - including nested FlowNodes. The variables identified in the FlowNode (result_name, input_argument_name, output_argument_name) may be accessed from any expression with the FlowNode (but not from other FlowNodes nor from expressions in the FlowGroup)

Issues/Questions

1. **Where does the stuff from a PatExec go?** We'll need to consider this. Perhaps some of its components (such as Category) go into the FlowGroup, others may go into FlowNodes (Selector) and maybe others are considered as arguments to a TestMethod (such as Timing, PatternBurst and the 1450.2 levels stuff).
2. **There's a lot of stuff missing, compared to the '99 proposal.** I know. I thought it would be good to settle on the basic execution structure and semantics before adding things like looping and TestIDs, or getting bogged down in bin syntax.

Here's what I had in April:

Flow Control

Implicit consensus: Flow control will be specified in a declarative manner (probably resembling a flow-chart or a Harel finite state machine diagram). For this document, I'll call the representation a "Flow", consisting of "FlowNodes" (the boxes in the flow-chart or state diagram) and "Arcs" (the edges connecting the FlowNodes). Hierarchy will be supported, such that a Flow can be encapsulated as a single block in a larger Flow.
It is an open issue as to the number and types of specific FlowNode types. Also an open issue is whether the number and types of the Arcs from a FlowNode should be predefined (such as exactly two, called Pass and Fail - a counter example would be a search which could have three results: SearchSuccessfulDevicePasses, SearchSuccessfulDeviceFails, SearchUnsuccessful).
Some number of entry points will be defined in the specification (such as Load, StartOfTest, Unload) - each will resolve to a Flow.
Binning must be coupled with the Flow - the Flow will probably indicate the points where binning operations occur. However, the Binmap (see below) will be a separate construct. Note that there are approaches to binning where there is no references to a bin in the Flow.
The working group is still open as to the specific structure of these FlowNodes, as well as the execution semantics. However, there should probably be at least the following: a) a FlowNode that encapsulates a call to a TestMethod, b) a FlowNode that encapsulates another flow (sometimes called a sub-Flow), c) a FlowNode that encapsulates termination (either a bin, or the exit point of a sub-Flow), and d) a FlowNode that can be used as a decision point (such as for operator test-options, or to control looping or other forms of conditional execution).
Note that Ernie's proposal includes pre and post actions, an arbiter, and definitions for the blocks at various levels in the hierarchy.
Other issues to consider: Inheritance (C++ style, STIL style), variables (will need to extend the types defined in 1450-1999

BinMaps

There is general agreement that there should be binmaps that are defined independently from the Flow Control. A possible example - the Flow Control might terminate in a Bin named "FailAC". Each Binmap would contain an entry named "FailAC", but each could identify a unique handler bin (often called "hardware bin"). The production operator could select the specific Binmap as the program is being loaded.
Note also that an existing test program may need a new Binmap after it is released. STIL.4 should allow defining an additional Binmap without requiring the original program to be "modified" (i.e. once a program is released to production it can be expensive to modify it).
Most people are familiar with the concepts of "Soft" and "Hard" bins. Perhaps the "soft" bins are defined in the Flow (bin-name, and pass/fail attribute), while the "hard" bin is defined via the Binmap.

Multisite

There is a general feeling that STIL.4 should identify the testing of a single device. Applying STIL.4 simultaneously to multiple devices (i.e. multisite) is considered a tester vendor's issue. However, there are specific issues associated with multisite testing, and the working group has agreed that Tom Micek ill head a sub-group to explore these issues and generate a proposal.

Test Methods Interface

TestMethods will be addressed by the yet-unformed 1450.5 group. However, the STIL.4 working group believes that it should propose an interface from STIL.4 to the TestMethods. No specifics have been explored.

Operator Control

Most test systems have some form of production operator interface that allows the operator to enter some limited amount of information. The acceptable values might be predefined by the test program (such as "Probe" or "Final", or the package type). The Flow Control must be able to query these values and by able to branch based on the values.
Note: the Semi standard TSEM (Testing Equipment Specific Equipment Model (E30.3) defines some number of variables. The working group may wish to consult that list when evaluating what information an operator may be responsible for.

Other forms of Concurrency

Aside from Multisite testing, other forms of concurrency could be helpful in some testing applications. Cores testing leads to the desire to have the test operations for independent cores on the same device to proceed concurrently. Similarly objectives arise in mixed-signal testing. The working group agreed (4/19/2002) to table these discussions at least until after much of the rest of this specification takes shape.

Test Identification

For datalogging (and debugging) it may be desirable to have the standard allow for a precise identification for each test operation (many tester systems use "test numbers" for this purpose).

Other Issues:

The following are issues that I added - thinking they are topics we may wish to discuss (DVO 4/30/2002)

Integration with 1450-1999 and other 1450.X specifications

Presumably STIL.4 sits at the highest level and references the other STIL specifications. For example, 1450-1999 identifies the PatternExec as the highest level block. Presumably, STIL.4 will somehow refer to the PatternExec from somewhere in its Flow Control (perhaps from a FlowNode, or from a TestMethod). Similarly, STIL.4 will need to reference the levels information from STIL.2.

Operator Output

Most test systems have some way in which the test program can output a message to the production operator. Should STIL.4 have some form of a printf statement?

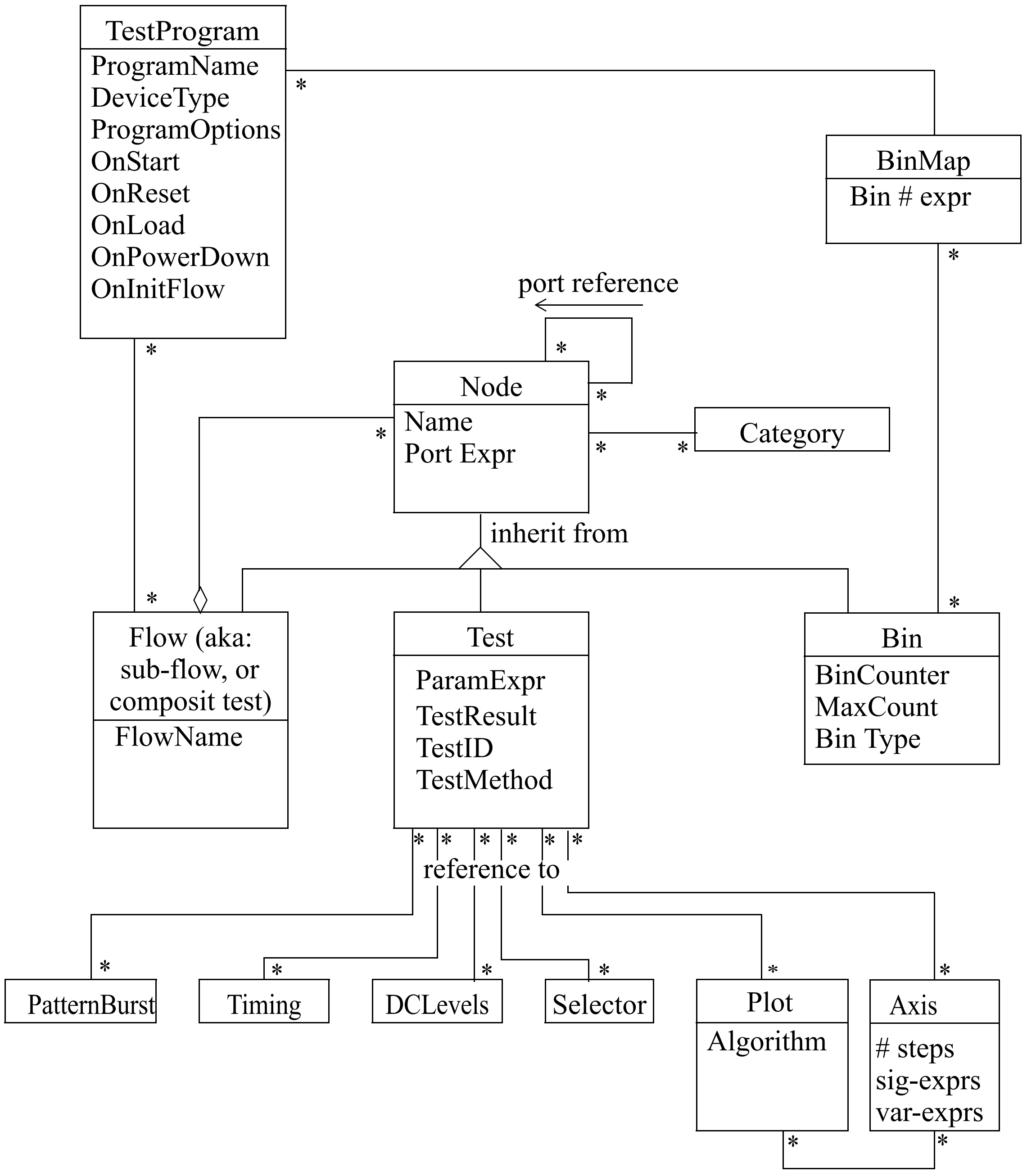
Data Flow

Information may be generated in early tests that are needed in later tests. 1450-1999 allows for this via the Meas variable name qualifier. Does this go far enough? Must we use a spec variable to share information? Can the data be a more complicated type (such as an array)? Are there scoping or life-span issues? For example, if a program refers to a Meas before the measurement occurs, will the program use the measurement from the previously socketed device?

Also, we'll need to extend some of the expression information of 1450.1999. The 3/8/99 document specified the types as Boolean, Integer, Real, Voltage, Current and Time. It added the following operators &, |, &&, ||.

Here's from the original ('99) draft.

Data Model for STIL-Flow Extensions



Flow Syntax Block Overview STIL (P1450.4)

There are several new blocks of information used in the definition of the Test Program Flow in STIL. These blocks are depicted graphically in the above data model. This section describes the main purpose of each of the blocks. The individual statement syntax for each of the blocks is defined in the section that follows.

Axis name { } - The Axis block is used to define parameters and usage for the purpose of performing a search operation. The typical use of axis objects is in the generation of a Shmoo plot.

Bin name { } - The Bin block is a node in a flow that is used as a terminal or counter, and is used to collect information about the flow. The Bin may be a “hard” bin - i.e., the termination of a path through the flow. The bin may be a “soft” bin which collects information and then continues through the flow.

BinMap name { } - The BinMap block is used to specify bin number mapping for each of the terminal bins in a given test program.

Category - A Category object is a derived object that is to be created at run time in the STIL environment. It is formed by collecting the set of variables in all Spec objects that are defined for the specified category.

DCLevels name { } - The DCLevels block specifies all of the logic levels and supply levels used by the device. This details of this block are being defined by the P1450.2 working group.

Flow { } - A flow object contains a set of nodes (Flow, Test, and Bin object) that are to be traversed in some order. The flow object is ofter referred to as a “sub-flow” if it is not the top-most flow in the test program, or a “composit-test” if it is to be considered as a test itself.

Node - A Node object is an abstract object that is used to define common attributes of any of the specific instances of nodes in a flow. The allowed node instances are: Flow, Test, and Bin.

PatternBurst name { } - A PatternBurst object specifies the list of patterns that are to be executed by a test program. This is defined in the base 1450 document

Plot name { } - A Plot object is the definition of the required parameters for the performing a Shmoo plot on a device.

Selector { } - A Selector object specifies the Min/Typ/Max values to be used for each of the spec-variables that are used by a test program. This is defined in the base 1450 document.

Test name { } - A Test object is a node that performs an actual test on the device. The activity to be performed and the set of required or optional data that is required for each test is dependent on the TestMethod that is referenced by the Test. The PatternExec as defined in the base 1450 is the simplest form of an execute functional test. The full set of TestMethods is being defined by the 1450.5 working group.

TestProgram name { } - The TestProgram block is the top level definition of the entry points into a flow for a given test program execution. There are typical several entry conditions that can activate the various sub-flows, such as “start running a test” or “reset a test”. This block also references variables that are used for controlling test program operation (Note: The set of program option variables would typically be displayed to a user when running the test program in non-debug mode.)

Timing name { } - A Timing object specifies the wave shape and event times to be used for each of the signals and waveform on those signals that are used by a test program. This is defined in the base 1450 document.

Flow Syntax Extensions to STIL (P1450.4)

```
Axis AXIS_NAME {
    Title "string";
    NumberSteps 'expr';
    Variable VAR_NAME {
        Start 'var_expr';
        Stop 'var_expr';
        SignalGroup 'sigref_expr';
    }
}  // end of Axis

Bin BIN_NAME {
    BinType ( Pass | Fail | Status ) ;
    BinNumber 'expr';
    ( CheckOverFlow ( True | False ); )
    ( MaxBinCount 'expr'; )
    ( Exit  {
        ( 'exit_port_expression'; )*
        ( 'exit_port_expression {  'VAR_NAME = expr'; }  )*
    })*   // end of Exit
}  // end of Bin

BinMap BIN_MAP_NAME {
    ( Bin BIN_NAME = 'expr'; )*
}  // end of BinMap

Flow FLOW_NAME {
    LoopExpr 'expr';
    LoopNotify ( TRUE | FALSE ) ;
    ( Node NODE_NAME FLOW_TEMPLATE {
        ( Category ( CATEGORY_NAME)+ ; )
        GoTo {
            ( NODE_NAME; )+
            ( NODE_NAME { 'port_expr'; } )+
    } )*  // end of Node
}  // end of Flow
Plot NAME {
    Title "string";
    ( Axis AXIS_NAME; )*
    Algorithm < Fixed | Stretch | Margin > ;      // Fixed - all unspecified vars unchaged
                                                  // Stretch - all unspecified vars scaled
                                                  // Margin - one parameter at a time
}  // end of Plot

Test TEST_NAME {
    LoopExpr 'expr';
    LoopNotify ( TRUE | FALSE ) ;
    ( Timing TIMING_NAME; )*
    ( DCLevels LEVELS_NAME ; )*
    ( PatternBurst PAT_BURST_NAME; )*
    ( Eval 'expr'; )
    ( Plot PLOT_NAME; )*
    ( Category (CATEGORY_NAME)+ ; )
    ( Selector SELECTOR_NAME; )*
    ( Exec TEST_METHOD_NAME {
        ( test_parameter_stmts; )*
    })+   // end of Exec
    ( Exit  {
        ( 'exit_port_expression'; )*
        ( 'exit_port_expression {  'VAR_NAME = expr'; }  )*
    })*   // end of Exit
}  // end of Test

TestProgram NAME {
    ( Device "string"; )
    ( BinMap BIN_MAP_NAME; )
    ( OnStart SUB_FLOW_NAME; )
    ( OnReset SUB_FLOW_NAME; )
    ( OnLoad SUB_FLOW_NAME; )
    ( OnPowerDown SUB_FLOW_NAME; )
    ( OnInitFlow SUB_FLOW_NAME; )
    ( ProgramOption  'OPT_NAME  = expr') ; )*
}  // end of TestProgram
```

Changes to the STIL 1450 Spec Block and Expressions

The following additions are required in the definition of the Spec variables in the basic STIL-1450 definition. Note: The initial STIL definition only supported variables of type 'Time', so the type was assumed to be always the same.

Definition of variable types

In the Spec block, the type of the variables may be specified immediately following the name of the variable. For compatibility with 1450 STIL, the default value is Time.

VAR_NAME **Boolean** | **Integer** | **Real** | **Voltage | Current** | **Time = ‘** *expr* **’ ;**
VAR_NAME **Boolean** | **Integer** | **Real** | **Voltage | Current** | **Time { }**

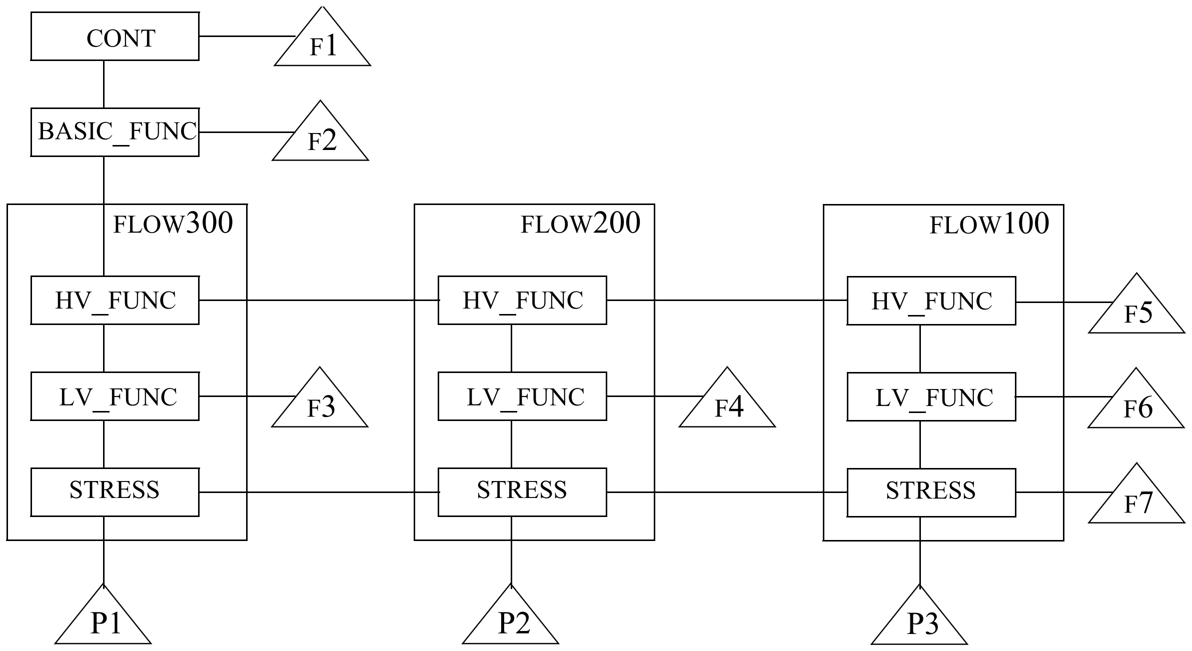
New Expression Operators

The following operators need to be added to the table of allowed operators (table 5 in the STIL document).

&	bitwise AND operator
|	bitwise OR operator
&&	Logical AND operator
||	Logical OR operator

Flow Example

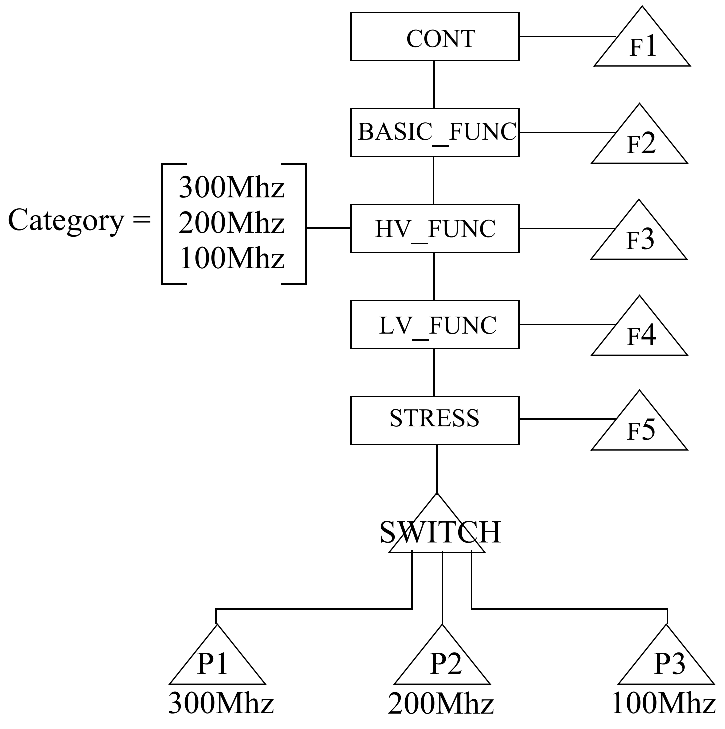
This section is an example of a test program flow to illustrate the STIL syntax described herein. The diagram below shows graphically the example flow. The flow nodes named CONT and BASIC _FUNCT are test nodes that exist within the main flow. The flow nodes named HV_FUNC, LV_FUNC, and STRESS are nodes that exist within a sub flow and hence may be re-used. The nodes named FLOW300, FLOW200, and FLOW100 are sub-flows that are defined by the Flow block named FLOWx00. The nodes named P1..P3 and F1..F7 and ‘pass’ bins and ‘fail’ bins respectiviely.



```
TestProgram EXAMPLE {
     DeviceType “IC-xyz”;
     OnStart MAIN_FLOW;
}
Flow MAIN_FLOW {
     Node CONT {
          GoTo { BASIC_FUNC; F1; }
     }
     Node BASIC_FUNC {
          GoTo { FLOW300; F2; }
     }
     Node FLOW300 FLOWx00 {
          Category SPEED_300MHZ;
          GoTo { P1; FLOW200.1; F3; FLOW200.2; }
     }
     Node FLOW200 FLOWx00 {
          Category SPEED_200MHZ;
          GoTo { P2; FLOW100.1; F4; FLOW100.2; }
     }
     Node FLOW100 FLOWx00 {
          Category SPEED_100MHZ;
          GoTo { P3; F5; F6; F7; }
     }
}
Flow FLOWx00 {
     Entry { HV_FUNC, Default; STRESS; }
     Node HV_FUNC { GoTo { LV_FUNC; Port 2; }}
     Node LV_FUNC { GoTo { STRESS; Port 3; }}
     Node STRESS { GoTo { Port 1; Port 4; }}
}
Test CONT {
     Selector AC_LOOSE;
     PatternBurst ABC;
     Timing DEF;
     DCLevels GHI;
     Exec Continuity {                      // execute the named test method
          PinsToTest ALL_PINS;
          IForce '-200mA';
          ShortThreshHold '-200mV';
          OpenThreshHold '-1.5V';
     }
     Exit {
          'TestResult == Pass';             // exit port 1, if true
          'TestResult == Fail';             // exit port 2, if true
}
Test BASIC_FUNC {
     Exec FunctionalTest { }
     Exit {
          'TestResult == Pass';             // exit port 1, if true
          'TestResult == Fail';             // exit port 2, if true
     }
}
Test HV_FUNC {
     Exec FunctionalTest { }
     Exit
          'TestResult == Pass';             // exit port 1, if true
          'TestResult == Fail';             // exit port 2, if true
     }
}
Test LV_FUNC {
     Exec FunctionalTest { }
     Exit {
          'TestResult == Pass';              // exit port 1, if true
          'TestResult == Fail';              // exit port 2, if true
     }
}
Test STRESS {
     Exec FunctionalTest { }
     Exit {
          'TestResult == Pass';              // exit port 1, if true
          'TestResult == Fail';              // exit port 2, if true
     }
}
```

Flow Example #2

This example shows a flow wherein a test node (HV_FUNC) is passed in 3 different category names and has the responsibility of testing each device at each category until a Pass condition occurs, or until all categories Fail. The result as to which category passed is used as the binning criteria at the end of the flow.



```
TestProgram EXAMPLE2 {
     DeviceType "IC-xyz";
     OnStart MAIN_FLOW;
}
Flow MAIN_FLOW {
     Node CONT {GoTo { BASIC_FUNC; F1; }}
     Node BASIC_FUNC {GoTo { HV_FUNC; F2; }}
     Node HV_FUNC {
          Category SPEED300MHZ, SPEED200MHZ, SPEED100MHZ;
          GoTo { LV_FUNC; F3; }
     }
     Node LV_FUNC { GoTo { STRESS; F4; }}
     Node STRESS { GoTo { P1; F5; }}
     Node SWITCH { GoTo { P1; P2; P3; }}
}
Test CONT {
     Exec Continuity { }
     Exit {'TestResult == Pass';              // exit port 1, if true
           'TestResult == Fail'; }            // exit port 2, if true
}
Test BASIC_FUNC {
     Exec FunctionalTest { }
     Exit {'TestResult == Pass';              // exit port 1, if true
           'TestResult == Fail';}             // exit port 2, if true
     }
}
Test HV_FUNC {
     Selector AC_LOOSE;
     PatternBurst ABC;
     Timing DEF;
     DCLevels GHI;
     Exec FunctionalTest {                      // execute the named test method
          PinsToTest ALL_PINS;
          IForce '-200mA';
          ShortThreshHold '-200mV';
          OpenThreshHold '-1.5V';
          Category = 1;
     }
     Exit {'TestResult == Pass' { Eval 'HVRes=1'; }} // exit if Pass 300mhz
     Exec FunctionalTest {
          PinsToTest ALL_PINS;
          IForce '-200mA';
          ShortThreshHold '-200mV';
          OpenThreshHold '-1.5V';
          Category = 2;
     }
     Exit {'TestResult == Pass' { Eval 'HVRes=2'; }} // exit if Pass 200mhz
     Exec FunctionalTest {
          PinsToTest ALL_PINS;
          IForce '-200mA';
          ShortThreshHold '-200mV';
          OpenThreshHold '-1.5V';
          Category = 3;
     }
     Exit {'TestResult == Pass' { Eval 'HVRes=1'; } // exit if Pass 100mhz
           'TestResult == Fail';                   // exit port 2
     }
}
Test LV_FUNC {
     Exec FunctionalTest { }
     Exit {
          'TestResult == Pass';              // exit port 1, if true
          'TestResult == Fail';              // exit port 2, if true
```

```
    }
}
Test STRESS {
    Exec FunctionalTest { }
    Exit {
        'TestResult == Pass';             // exit port 1, if true
        'TestResult == Fail';             // exit port 2, if true
    }
}
Bin SWITCH {
    BinType Status;
    Exit {
        'HVRes==1';   // exit port 1
        'HVRes==2';   // exit port 2
        'HVRes==3';   // exit port 3
    }
}
```