OPENSTAR™

STC-S0010R1.0

Test Programming Language Specifications

DISCLAIMER OF WARRANTIES

THIS DOCUMENT, INCLUDING ANY SPECIFICATIONS AND OTHER INFORMATION OR MATERIALS INCLUDED IN OR ACCOMPANYING THIS DOCUMENT (THE "MATERIALS"), ARE PROVIDED "AS IS." STC AND EACH OF ITS MEMBERS (INCLUDING ANY MEMBERS THAT MAY HAVE AUTHORED OR CONTRIBUTED TO THE MATERIALS) MAKE NO REPRESENTATIONS OR WARRANTIES (EXPRESS, IMPLIED OR OTHERWISE), INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT, AND ALL

SUCH REPRESENTATIONS AND WARRANTIES ARE EXPRESSLY DISCLAIMED.

Without limitation of the generality of the foregoing, STC and each of its members caution that:

(1) The Materials are merely drafts of a provisional or experimental nature, and may contain inaccuracies, defects and deficiencies that may not be corrected by STC or its members;

And

(2) Use of the Materials and compliance with the specifications included in the Materials ("Specifications") may infringe third party patents and other intellectual property rights, and users of the Materials (not STC or its members) are solely responsible for (a) identifying what, if any, patents may be relevant to use of the Materials or compliance with the Specifications and (b) seeking and obtaining licenses of any such patents, if users determine that such licenses are necessary or appropriate.

STC may (but is not obligated to) update or otherwise modify the Materials from time to time. If STC provides updates or other modifications, it may, at its option, make them available in accordance with its then-current license for such updates or other modifications. In addition, neither STC nor any of its members is committing to make available a product (on a commercial basis or otherwise) complying with or otherwise using the Specifications.

LIMITATIONS OF LIABILITY

IN NO EVENT WILL STC OR ITS MEMBERS (INCLUDING ANY MEMBERS THAT MAY HAVE AUTHORED OR CONTRIBUTED TO THE MATERIALS) BE LIABLE FOR ANY SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, OR ANY LOST REVENUE, LOST PROFITS OR LOST BUSINESS (OR ANY LOSS OF DATA), HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY

(WHETHER BASED ON BREACH OF WARRANTY, BREACH OF CONTRACT, TORT, INDEMNIFICATION, OR OTHERWISE), ARISING OUT OF OR OTHERWISE RELATED TO THE MATERIALS OR THE SUBJECT MATTER THEREOF, INCLUDING THE FURNISHING, PRACTICING, IMPLEMENTATION, MODIFICATION OR OTHER USE OF THE MATERIALS, EVEN IF STC AND/OR ITS MEMBERS WERE ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

IN ADDITION, IN NO EVENT WILL THE AGGREGATE LIABILITY OF STC AND ITS MEMBERS (INCLUDING ANY MEMBERS THAT MAY HAVE AUTHORED OR CONTRIBUTED TO THE MATERIALS) IN CONNECTION WITH THE MATERIALS OR THE SUBJECT MATTER THEREOF (REGARDLESS OF THE THEORY OF LIABILITY, WHETHER BASED ON BREACH OF WARRANTY, BREACH OF CONTRACT, TORT, INDEMNIFICATION, OR OTHERWISE) EXCEED THE AMOUNT OF THE MOST RECENT ANNUAL DUES PAID BY THE PERSONS CLAIMING LIABILITY.

Table of Contents

Revision	Date	Revised By	Description
1.0		SWG	*Initial STC Release*

1

Introduction

Preparatory to embarking upon first reading of this specification document the reader must be familiar with the architecture and terminology of the OPENSTAR™ Tester Operating System (***TOS***). Such familiarity is assumed in order to facilitate brevity and obviate the need to introduce discussion already present in other texts.

TOS supports testing of a Device Under Test (*DUT*) through *patterns*. A *pattern* is a named set of *vectors*. A *vector* is a signal waveform that spans the pins of a DUT and, potentially, multiple cycles of the DUT tester machine. A *vector* is specified as a series of *waveform characters*. A *waveform character* is an alphanumeric mnemonic for a signal waveform segment. A *waveform character* is specified as a labeled set of *predefined tester events* (governing waveform level changes) that are scheduled to occur at points in time. *Waveform characters* are defined in *waveform table*s each of which provides its own decryption for a *waveform character* in a pattern. *Waveform tables* are associated with a *pattern* via *Waveform(Table) Selectors*. Waveforms are generic, they may apply both to pins that drive and pins that compare (expect) values. *Patterns* are given context through *test plans* that determine the semantic interpretation of *pattern* content, the states of the tester machine and the execution flow of the subordinate *tests* that execute the *patterns* against the DUT.

This document shall present the framework that *TOS* provides for the developer of *tests* and *test plans* and the framework specifications to which *TOS* developer, *test* and *test plan* developers alike shall conform. Third party vendors providing customized extensions shall be treated as developers.

This document shall refer to methods that are scoped to their containers using the familiar notation:

container::method

and, in some cases, shall further specify the signature of the method for illustrative clarity. In general, signatures shall not be presented and the reader should refer to class documentation for the method container to obtain detailed information on method properties.

1.1 The Test Programming Language

TOS provides support for the development of *tests* and *test plans* through a dedicated Test Programming Language (***TPL***). *TPL* provides structure and hierarchy to *test* and *test plan* through the following elements:

- *User Vars:* comprising collections of u*ser variables and constants*
- *Specification sets*: comprising collections of enumerated values and their indices (*selectors*)
- *Levels*: specifying static states of the tester machine during *parametrics* and *calibrations*
- *Timings*: specifying *waveform characters,* digital waveform transition placement (*edges)*, time periods of tester machine cycles
- *Timing Maps*: specifying correlation between *pattern* and *timing*
- *Test Condition Groups:* specifying an association of *specification set, timing* and *timing map*
- *Test Conditions*: specifying an association of a *test condition group* and a *selector* index for its *specification set*
- *Bin definitions*: specifying the hierarchical storage of *test* results
- *Flows*: specifying execution order of *tests* within a *test plan*
- *Flowables:* the base classes for *flow* and *test* classes providing for branch entry points
- *Pre-headers*: specifying signatures and customization of *test* methods
- *Test plans:* incorporating the previously specified features to administer execution of the *tests*.

1.2 Specification Identifiers

For simplicity specifications are straight enumerations within sections of this document. It is recommended that a specification be referred to by its number and scoped to its document section.

1.3 Syntax Conventions

The *TPL source* file is a plain ASCII file. The general syntax for this file is presented for which:

- Syntax words other than keywords are taken to be generic.
- Syntax keywords are indicated by boldface.
- The token |optional(optional|)shall denote that inclusion of the preceding(following) syntax element is not mandatory and optional<default> shall indicate any *default* value.
- The token |repeat (repeat|)shall denote that the preceding(following) syntax element may be indeterminately repeated to generate a list.
- The separator token | shall denote mutually exclusive *or*.

For clarity, syntax presentation may be delimited by bracketed number, line and `[]`. Unless in boldface, these are not part of the *pattern* syntax. An element in a combinatorial set may be highlighted in grey to provide a concrete example of how this element shall appear in code.

Alert: The syntax representation is illustrative of the usage of the native Backus-Naur formulation (*BNF*) specified in Appendix A and includes semantic constraints not evident in the *BNF*. The *BNF* is to be regarded as the ***formal and definitive*** specification of the syntax.

2

The User Vars Syntax

User Vars are the definitions of variables and constants used throughout a *test plan*. They comprise *lhs* declaration of the variable or constant as a strong type and *rhs* constant value or algebraic expression.

2.1 Requirements upon User Vars

1. The syntax:

   UserVars name|optional<name=**_UserVars**>

   shall declare a *User Var* collection whose content is delimited by **{** and **}**

2. The UserVars syntax:

   Const|optional

   [**Integer** | **UnsignedInteger** | **String** | **Voltage** | **VoltageSlew** | **Current** | **Power** |
   Time | **Length** | **Frequency** | **Resistance** | **Capacitance**]

   [variable =

   number[[V | mV]|Voltage [/[S | ms | uS | nS | pS]|VoltageSlew

   [A | mA | uA]|Current [W | mW | uW | nW]|Power [S | mS | uS | nS | pS]|Time [KM | M | dM | cM | mM | uM | pM | fM]|Length [Hz | KHz | MHz | GHz | THz]|Frequency [Ohms | KOhms | MOhms]|Resistance

   [F | mF | uF | nF | pF | fF]|Capacitance];|optional | expression;]

   |

   variable[size] = { [expression[, expression]|repeat,optional [, **Others**=expression]|optional] |
   Others=expression }

   shall declare the constant or variable with designation variable, or rank size array with designation variable[size] and type:

   Integer or UnsignedInteger or String or Voltage or VoltageSlew or Current or Power or Time or Length or Frequency or Resistance or Capacitance

   to carry the value number that

- is specified in integer, floating point or scientific notation
- has optional dimensionality, consistent with the variable type, expressed in SI notation with `u` denoting μ

or result of the expression `expression` where `expression` comprises mathematical operators and operands of form:

`number`, dimensioned or dimensionless

`variable[array_index]` denoting the element `array_index` of array variable `variable`

and shall define the meaning of `type, variable` and `expression` ***in all subsequent references***

3. The `UserVars` syntax:

   `type (variable)`

   shall redefine the type of the variable `variable` to be `type`

4. The syntax:

   `[user_var.]`$|_{\text{optional}}$ `variable`

   shall represent the `variable` declared in the

   User Var `user_var`, current *User Var* or **`_UserVars`**

5. The syntax:

   `container_name::function([expression [, expression]`$|_{\text{repeat,optional}}$

   shall return the value of the *user function* `function,` declared in `container_name`, for the argument(s) `expression`.

6. The declarations for *user vars* shall reside in files having extension *usrv*

2.2 Usage Example

The following is a usage example of the *user vars* syntax:

```
# ------------------------------------------------------
# File limits.usrv
# ------------------------------------------------------

Version 1.0;

#
# This UserVars collection declaration declares a set of
# globally available variables and constants.
#
UserVars
{
   # Some constant Integer globals used in various places.
   Const Integer    MaxInteger = 2147483647;
   Const Integer    MinInteger = -2147483647-1;

   # Smallest value such that 1.0 + Epsilon  != 1.0
   Const Double  Epsilon = 2.2204460492503131e-016;

   # Some important constants related to Double
   Const Double MaxDouble = 1.7976931348623158e+308;
   Const Double MinDouble = - MaxDouble;
   Const Double ZeroPlus = 2.2250738585072014e-308;
   Const Double ZeroMinus = - ZeroPlus;


}
```

3

Custom Types Syntax

Custom types are compound types comprising variables whose types are the base types of *TPL*.

3.1 Requirements upon Custom Types

1. The syntax:

   `CustomType` `custom_type`

   shall declare a *custom type* with designation `custom_type` and content delimited by **`{`** and **`}`**

   1. The *custom type* syntax:

      `[ type | custom_type] variable;`

      shall declare the variable `variable` whose type is `type` or `custom_type` and where `custom_type` is the name of a previously defined *custom type*.

   2. The syntax:

      `custom_type->variable`

      shall represent the variable `variable` in *custom type* `custom_type`

   3. The syntax:

      `custom_type object{ [custom_type->]`$|_{\text{optional}}$ `variable=expression`

      `[, [custom_type->]`$|_{\text{optional}}$ `variable=expression]`$|_{\text{repeat}}$ `}`

      shall initialize the object, `object`, of *custom type* `custom_type` by assignment of `expression`(s) to `variable`(s)

   4. The declarations for *custom type* shall reside in files having extension *ctyp*

3.2 Usage Example

The following is a usage example of the *custom types* syntax*:*

```
# ------------------------------------------------------------
# File MyCustomTypes.ctyp
# ------------------------------------------------------------

Version 1.0;

CustomType Foo
{
   String    S1;
   Integer   I1;
   String    S2;
}

CustomType Bar
{
   Foo    Foo1;
   String S1;
   Foo    Foo2;

}
```

4

The Levels Syntax

Levels are sets of assignments of values to measurement parameters used by *parametric* and *calibration tests*.

4.1 Requirements upon Levels

1. The syntax:

   `[Levels | DCParametrics | Calibration] name`

   shall declare a *levels*, *DC parametrics* or *calibration* with designation `name` whose content is delimited by `{` and `}`

2. The *levels*, *DC parametrics* or *calibration* syntax:

   `[[Delay | MinDelay] expression ;]]`$|_{\text{optional}}$

   `pin { parameter = [expression | Slew( expression, expression )];`$|_{\text{repeat}}$ `}` $\Big|_{\text{repeat}}$

   shall assign the value of `expression` or `Slew` to the parameter `parameter` of pin(group) `pin` with a preceding or following *delay* of:

   $\texttt{expression}|_{\textbf{MinDelay}} \leq \textit{delay} \leq \texttt{expression}|_{\textbf{Delay}}$

3. The *levels* syntax shall reside in files having extension *lvl.*

4.2 Usage Example

The following is a usage example of the *levels* syntax:

```
# ------------------------------------------------------
# File myDUTLevels.lvl
# ------------------------------------------------------

Version 1.0;

Import CPUXResources.rsc;
Import CPUXpindesc.pin;

Levels CPU_XLevels
{
    #
    # Specifies pin-parameters for various pins and
    # pin groups using globals and values from
    # the specification set.
    #
    # The order of specification is significant.
    # Pin parameters will be set in order from
    # first to last in this Levels section, and
    # from first to last for each pin or pin-group
    # subsection.
    #
    # From the imported pin description file cpuXpins.pin,
    # the InPins group is in the "dpin" resource.  From the
    # imported resource definition file cpuXresources.rsc,
    # the "dps" resource has parameters named VIL and VIH.
    #
    InPins { VIL = v_il; VIH = v_ih + 1.0; }

    # The following statement requires a delay of 10 uS after
    # the call to set the InPins levels.  Actual delay will be
    # a small system defined range around 10.0E-6:
    #      10.0E-6 - delta <= actual <= 10.0E-6 + delta
    Delay 10.0E-6;

    #
    # For the OutPins, the levels for the parameters
    # VOL and VOH are specified.
    #
    OutPins { VOL = v_ol / 2.0; VOH = v_oh; }

    # The clock pin will have special values.
    Clock { VOL = 0.0; VOH = v_ih / 2.0; }

    # A Delay of 10 uS after the call to set Clock levels.
    # This is a minimum delay, that is guaranteed to be for
    # at least 10.0 uS, though it may be a little more:
    #      10.0E-6 <= actual <= 10.0E-6 + delta
    MinDelay 10.0 uS;

    #
    # The PowerPins group is in the "dps" resource.  Pins of this
    # pin group have special parameters:
    PowerPins
    {
        # VForce reaches its final value of 2.0 V from its
        # present value in a ramp with a Voltage Slew Rate
        # of ±.01 Volts per Second.
        VForce = Slew(0.01, 2.0 V);

        # VBumpMode can be Enable or Disable.  This is to
        # allow testing at various voltage ranges.
        VBumpMode = Disable;
    }
}
```

As seen above, each `Levels` block is made up of a number of *levels items*, each of which specifies parameters for a pin or pin group. Each levels item can specify a number of *resource parameters*.

The Levels declaration syntax is also used to declare values for DCParametrics and Calibration. These have exactly the same syntax as a Levels declaration, and the same runtime semantics.

5

The Specification Set Syntax

Specification Sets are the definitions of variables, finite sets of discrete values held by each variable and a set of named indices serving as *selectors* of elements of the sets.

5.1 Requirements upon Specification Sets

1. The syntax:

   `SpecificationSet` `name`$|_{\text{optional}}$ `( selector` $|_{\text{repeat}}$ `)`

   shall declare a *Specification Set* named `name` or unnamed, having indices named `selector` $|_{\text{repeat}}$ and content delimited by **`{`** and **`}`**

2. The specification set syntax:

   `type variable = [expression][, expression]`$|_{\text{optional,repeat}}$`;]`$|_{\text{repeat}}$

   shall declare the strongly typed variable `variable` to hold the list of values

   `expression[, expression]`$|_{\text{repeat}}$ where:

   - the size of the list is equal to the number of indices `selector` $|_{\text{repeat}}$
   - *or* the size of the list is less than the number of indices and the last value in the list shall span the remainder of the list

3. The *specification set* syntax shall reside in files having extension *spec.*

5.2 Usage Example

The following is a usage example of the *specification set* syntax:

```
# ------------------------------------------------------
# File Aaa.spec
# ------------------------------------------------------

Version 1.0;

Import limits.usrv;

SpecificationSet Aaa(s1, s2, s3, s4)
{
   Double xxx = 1.0, 2.0, 3.0, 4.0;
   Integer yyy = 10, 20, 30, 40;
   Integer zzz = MaxInteger - xxx,
                 MaxInteger - xxx - 1,
                 MaxInteger - xxx - 2,
                 MaxInteger - xxx;

   # The following declaration associates a single
   # value, which will be chosen regardless of the
   # selector.  It is equivalent to:
   #  Integer www = yyy + zzz, yyy + zzz, yyy + zzz, yyy + zzz
   Integer www = yyy + zzz;
}
```

The above *Specification Set* with the selector s3 will make the following associations:

```
xxx = 3.0;
yyy = 30;
zzz = MaxInteger – xxx – 2;
```

www = yyy + zzz;

6

Timing Syntax

Timing blocks define the *waveform characters* used to encrypt waveform segments within *vectors* of a pattern. Each *waveform character* is defined as a set of *events* occurring at specified points in time that determine waveform voltage transitions and for which hardware dedicates resources referred to as *edge resources*.

6.1 Requirements upon Timing Blocks

1. The syntax:

   `Version` `version;`

   shall specify the *TOS* version, `version`.

2. The syntax:

   `[`**`Import`** `[[file.]`$_{\text{optional}}$ `block][,[file.]`$_{\text{optional}}$ `block]`$_{\text{optional,repeat}}$ `;]` $_{\text{optional,repeat}}$

   shall insert the *timing block(s)*, `block`, contained in timing file(s), `file`

3. The syntax:

   `Timing` `block`

   shall declare the *timing block*, `block`, whose content is delimited by **`{`** and **`}`**

4. The *timing block* syntax:

   `CommonSection`

   shall declare the *common section* of the *timing block* whose content is delimited by **`{`** and **`}`**

5. The *timing block common section* syntax:

   `(1)` **`Domain`** `domain_name` **`[ DataRate`** `data_rate` **`]`**

   `{`

(2) **PeriodTable** { [period_name { expression; }]|repeat }

(3) **Pin** pin

{

(4) [**DefaultState**=[**ForceDown** | **ForceUp** | **ForceOff** | **D** | **U** | **Z**];]|optional

(5) **WaveformTable** wft_name

{

(6) { wf_char[/wf_char] |optional,repeat [mdr_cycle]|optional

(7) { [[**D**|**U**|**Z**|**P**|**L**|**X**|**x**|**D**|**V**|**l**|**h**|**t**|**v**][/[**D**|**U**|**Z**|**P**|**L**|**X**|**x**|**D**|**V**|**l**|**h**|**t**|**v**]]|optional,repeat @ time

(8) [, **E**[**dge**]|optional number [/**E**[**dge**]|optional number] |optional,repeat]|optional ;]|repeat }

repeat }

repeat }

PinOptions

{

(9) **CompareMode**=[**Single** | **Multi**]; |repeat

optional }

repeat }

}

shall declare:

(1) the timing domain to be `domain_name` with data rate `data_rate`

(2) the period `period_name` with value `expression`

(3) the pin(group) `pin` to have, at multi data rate cycle `mdr_cycle`:

(4) default state: `ForceDown or ForceUp or ForceOff or D or U or Z`

(5) waveform table `wft_name` defining:

(6) *waveform character* `wf_char` to be the event: `D|U|Z|P|L|X|x|D|V|l|h|t|v`

at time `time`, where `time` is an `expression`, and the edge resource `number` that shall generate the event

(9) high-impedance (Z) comparator enabling `CompareMode=Multi` in addition to high and low.

Multiple declarations of `CompareMode` are ignored except in the case of the last declaration.

6. The *timing* syntax shall reside in files having extension *tim.*

6.2 Usage Example

The following is a usage example of the *timing* syntax:

```
#
# Timing file

Version 1.0;

Import pindesc.pin;

Timing basic_functional
{
    CommonSection
    {
        Domain default          # The default time-domain
        {
            PeriodTable
            {
                # The variable tper_prog referenced here is
                # specified in a SpecificationSet
                Period per0 {60nS;}
                Period per1 {tper_prog;}
                Period per2 {120nS;}
                Period per3 {tper_prog * 3;}
            }
            Pin SIG
            {
                WaveformTable seq1
                {
                    { 0/1 { D/U@10nS, E1/E1; D@30nS, E2; Z@45nS, E3; }    }
                    { d/u { D/U@12nS, E1/E1; U@32nS, E2; Z@42nS, E3; }    }
                    { L/H { L/H@17nS, E5/E6; }                            }
                    { m/n { L/H@15nS, E5/E6; }                            }
                }
                WaveformTable seq2
                {
                    { 0 { D@22nS, E1; U@42nS ,E2; Z@52nS, E3; }    }
                    { 1 { U@22nS, E1; D@42nS ,E2; Z@52nS, E3; }    }
                    { L { L@37nS, E5; }                             }
                    { H { H@37nS, E6; }                             }
                }
                WaveformTable seq3
                {
                    { a { D@10nS, E1; D@30nS, E2; Z@45nS, E3; }    }
                    { b { U@10nS, E1; D@30nS, E2; Z@45nS, E3; }    }
                    { c { L@17nS, E5; }                             }
                    { d { H@17nS, E6; }                             }
                }
                WaveformTable seq4
                {
                    { L { L@15nS, E5; } }
                    { H { H@15nS, E6; } }
                }
            }
```

```
            Pin CLK
            {
                WaveformTable seq1
                {
                    { 1 { U@20nS, E1; D@40nS, E2; }}
                }
            }
            Pin FASTCLK
            {
                # This illustrates a faster clock.
                WaveformTable seq1
                {
                    {1 { U@10nS, E1; D@20nS, E2; U@40nS, E3; D@50nS, E4; }}
                }
            }
            Pin DATA
            {
                # This illustrates the Window Compare.
                WaveformTable ReadCycle
                {
                    { L { l@10nS, E5; x@48nS, E6; }}
                    { H { h@10nS, E5; x@48nS, E6; }}
                }
            }
        }
    }
}
```

7

Timing Map Syntax

Timing maps provide named mappings between a *pattern* and a *waveform table*, defining its *waveform characters*, and the period to be used for each cycle.

7.1 Requirements upon Timing Map

1. The syntax:

   `Version` `version;`

   shall specify the *TOS* version, `version`.

2. The syntax:

   [**`Import`** `[[file.]`$_{\text{optional}}$ `map][,[file.]`$_{\text{optional}}$ `map]`$_{\text{optional,repeat}}$ `;]` $_{\text{optional,repeat}}$

   shall insert the *timing map*, `map`, contained in timing file, `file`

3. The syntax:

   `TimingMap` `map`

   shall declare the *timing map*, `map`, whose content is delimited by **`{`** and **`}`**

4. The *timing map* syntax:

```
Domain domain_name
{
   WaveformMap { {pin [, pin]|optional,repeat } wf_selector period { wf_table [, wf_table]|optional,repeat}
}
```

shall generate in timing domain `domain_name` the mapping pin(group)(s) `pin` to period `period` and waveform table(s) `wf_table` with the designation `wf_selector`.

5. The *timing map* syntax shall reside in files having extension *tmap.*

7.2 Usage Example

The following is a usage example of the *timing map* syntax:

```
#
# Timing Map file
#
Version 1.0;

Import pindesc.pin;

TimingMap myGalaxyTimingMap
{
    Domain default
    {
        WaveformMap
        {
            PinFormat { SIG, CLK, FASTCLK, DATA }
            wfs1, per0, { seq1, seq1, seq1, ReadCycle }
            wfs2, per0, { seq2, seq1, seq1, ReadCycle }
            wfs3, per0, { seq3, seq1, seq1, ReadCycle }
            wfs4, per2, { seq1, seq1, seq1, ReadCycle }
        }
    }

}
```

8

The Test Condition Group Syntax

Test condition groups are associations of *specification sets*, *levels*, *timings* and *timing maps.*

8.1 Requirements upon Test Condition Groups

1. The syntax:

   `TestConditionGroup` `name`

   shall declare a *test condition group* with designation `name` and content delimited by **`{`** and **`}`**

2. The *test condition group* syntax:

   `specification_set_name; | specification_set`

   shall reference the previously defined and named specification set `specification_set_name` or declare an unnamed specification set `specification_set`

3. The *test condition group* syntax:

   `levels_name; | levels`

   shall reference the previously defined and named *levels, DC parametric* or *calibration* `levels_name` or declare an unnamed *levels, DC parametric* or *calibration* `levels`

4. The *test condition group* syntax:

   `Timing` `{ [` **`Timing`**`=timing_block [`**`TimingMap`**`=timing_map]`$|_{\text{optional}}$ `]`
   `| [` **`TimingMap`**`=timing_map` **`Timing`**`=timing_block ] }`

   shall reference the timing block `timing_block` and timing map `timing_map` where each may be expressed in form `[filename:]`$|_{\text{optional}}$ `name`. Example: *timing_file.tim***:***timing_block_1*

5. The *test condition group* syntax shall reside in files having extension *tcg*

8.2 Usage Example

The following is a usage example of the *test condition group* syntax:

```
# ------------------------------------------------------
# File myTestConditionGroups.tcg
# ------------------------------------------------------

Version 1.0;

Import myvars.usrv;
Import myDUTLevels.lvl;
Import edges.spec;
Import simple.tim;
Import simple.tmap;
```

```
TestConditionGroup TCG1
{
    # This Local SpecificationSet uses user-defined selectors
    # "min", "max" and "typ".  Any number of selectors with any
    # user defined names is allowed.
    #
    # The specification set specifies a table giving values for
    # variables that can be used in expressions to initialize
    # timings and levels.  The specification set below defines
    # values for variables as per the following table:
    #             min            max          typ
    #  v_cc       2.9            3.1          3.0
    #  v_ih  VInHigh + 0.0  VInHigh + 0.2 VInHigh + 0.1
    #  v_il  VInLow + 0.0   VInLow + 0.2  VInLow + 0.1
    #  ...
    # A reference such as "VInHigh" must be previously defined
    # in a block of UserVars.
    #
    # Thus, if the "max" selector was selected ina functional
    # test, then the "max" column of values would be bound to
    # the variables, setting v_cc to 3.1, v_ih to vInHigh+2.0
    # and so on.
    #
    # Note that this is a local specification set, and has no
    # name.
    SpecificationSet(min, max, typ)
    {
        # Minimum, Maximum and Typical specifications for
        # voltages.
        Voltage v_cc = 2.9, 3.1, 3.0;
        Voltage v_ih = MyVars.VInHigh + 0.0,
                       MyVars.VInHigh + 0.2,
                       MyVars.VInHigh + 0.1;
        Voltage v_il = MyVars.VInLow + 0.0,
                       MyVars.VInLow + 0.2,
                       MyVars.VInLow + 0.1;

        # Minimum, Maximum and Typical specifications for
        # leading and trailing timing edges.  The base
        # value of 1.0E-6 uS corresponds to 1 nanosecond,
        # and is given as an example of using scientific
        # notation for numbers along with units.
        Time t_le = 1.0E-6 uS,
                    1.0E-6 uS + 4.0 * MyVars.DeltaT,
                    1.0E-6 uS + 2.0 * MyVars.DeltaT;
        Time t_te = 30ns,
                    30ns + 4.0 * MyVars.DeltaT,
                    30ns + 2.0 * MyVars.DeltaT;
    }

    # Refers to the CPU_XLevels imported earlier.  It
    # is one of possibly many levels objects that have been
    # imported from the above file.
    Levels CPU_XLevels;
    # Refers to file simple.tim containing the single timing
```

```
    # Tim1, and simple.tmap contianing the single timing
    # map TMap1
    Timings
    {
        # Pick up Tim1 from simple.tim
        Timing = Tim1;

        # Pick up TMAP1 from simple.tmap
        TimingMap = TMap1;
    }
}

# Another test condition group
TestConditionGroup TCG2
{
    # ClockAndDataEdgesSpecs is a specification set which
    # is available in the edges.specs file.  Assume it has
    # the following declaration:
    #   SpecificationSet ClockAndDataEdgesSpecs(min, max, typ)
    #   {
    #       Time clock_le = 10.00 uS, 10.02 uS, 10.01 uS;
    #       Time clock_te = 20.00 uS, 20.02 uS, 20.01 uS;
    #       Time data_le  = 10.0 uS, 10.2 uS, 10.1 uS;
    #       Time data_te  = 30.0 uS, 30.2 uS, 30.1 uS;
    #   }
    # A SpecificationSet reference to this named set is below:
    SpecificationSet ClockAndDataEdgesSpecs;

    # An inlined levels declaration.  Since the associated
    # specification set (above) does not have variables such
    # as VInLow, VInHigh, VOutLow and VOutHigh, they must
    # resolve in the default UserVars collection.
    Levels
    {
        InPins { VIL = VInLow; VIH = VInHigh + 1.0; }
        OutPins { VOL = VOutLow / 2.0; VOH = VOutHigh; }
    }

    # Refers to file simple.tim containing the single timing
    # Tim1, and simple.tmap contianing the single timing
    # map TMap1
    Timings
    {
        # Pick up Tim1 from simple.tim
        Timing = Tim1;

        # Pick up TMAP1 from simple.tmap
        TimingMap = TMap1;
    }
}
```

In the above example, the test condition group TCG1 describes a specification set with three selectors named “min”, “typ” and “max”. There can be any number of distinct selectors. Within the body of the specification set, variables `v_il`, `v_ih`, `t_le` and `t_te` are initialized with triples of values, corresponding to the selectors. So in the above example, an instance of TCG1 with the selector “min” will bind the variable `v_il` with the first numeric value, `(VInLow+0.0)`.

9

Test Condition Syntax

Test conditions comprise a *test condition group* and a *selector* name that determines the *specification set* values applied to the *test condition group*.

9.1 Requirements upon Test Conditions

1. The syntax:

   `TestCondition` `name`

   shall declare a *test condition* with designation `name` and whose content is delimited by **`{`** and **`}`**

2. The *test condition* syntax:

   [**`TestConditionGroup`** `tcg_name;` [**`Selector`** `= selector_name;`]|optional]|repeat

   |

   [**`Selector`** `= selector_name;` **`TestConditionGroup`** `tcg_name;`] |repeat

   shall reference the previously defined *test condition group* `tcg_name` and specify its *specification set selector* `selector_name`

3. The *test condition* syntax shall reside in *test plans*

9.2 Usage Example

The following is a usage example of the *test condition* syntax:

```
TestCondition TCMin
{
    TestConditionGroup = TCG1;
    Selector = min;
}

TestCondition TCTyp
{
    TestConditionGroup = TCG1;
    Selector = typ;
}

TestCondition TCMax
{
    TestConditionGroup = TCG1;
    Selector = max;

}
}



#
# Declare a FunctionalTest "MyFunctionalTest" that refers to three
# Test Condition Group instances.
#
Test FunctionalTest MyFunctionalTest
{
    # Specify the Pattern List
    PList = pat1Alist;

    # Any number of TestConditions can be specified:
    TestCondition = TCMin;
    TestCondition = TCMax;
    TestCondition = TCTyp;

}
}
```

10

The Bin Definitions and Counters Syntax

Bin definitions generate a hierarchical structure serving as repository for *test* results. Bin definitions comprise groups of bins each containing bins that may parent other bins or serve as terminal leaf node daughter bins. Parent and daughter bins need not belong to the same bin group. *Counters* are integers that may be incremented by one.

10.1 Requirements upon Bin Definitions and Counters

1. The syntax:

   BinDefs

   {

   repeat| **BinGroup** group_name { [[**Bin** | **LeafBin**] bin_name bin_id : bin_desc [, parent]|optional ;] |repeat}

   }

   shall define bin group `group_name` to contain the `Bin` or `LeafBin` named `bin_name` with integer identifier `bin_id`, description string `bin_desc` and previously defined parent bin `parent`

2. The syntax:

   SortBinGroup = bin_name;

   shall specify that the contents of bin group `group_name` and only `group_name` shall be used.

3. The syntax:

   Counters { name[, name] |optional, repeat }

   shall declare a *counter* whose designation is `name`

4. The *bin definitions* syntax shall reside in files with extension *bdefs*

10.2 Usage Example

The following are usage examples of the *bindef* and *counter* syntax:

```
# ------------------------------------------------------------
# File cpuxbins.bdefs
# ------------------------------------------------------------

Version 1.0;

BinDefs
{
    # The PassFailBins are an outermost level of
    # bins.  They are not a refinement of any other
    # bins.
    BinGroup PassFailBins
    {
        Bin Pass 0: "Count of passing DUTS.";
        Bin Fail 1: "Count of failing DUTS.";
    }

    # The HardBins are a next level of refinement.
    # HardBins are a refinement of the PassFailBins
    # declared just before.
    BinGroup HardBins
    {
        Bin Pass3GHz10: "DUTs passing 3GHz",Pass;
        Bin Pass2_8GHz  11: "DUTs passing 2.8GHz",      Pass;
        Bin Fail3GHz12: "DUTs failing 3GHz",Fail;
        Bin Fail2_8GHz  13: "DUTs failing 2.8GHz",      Fail;
        Bin FailLeakage 14: "DUTs failing leakage",     Fail;
    }

    # The SoftBins are a next level of refinement.
    # SoftBins are a refinement of HardBins declared
    # just before.
    BinGroup SoftBins
    {
        LeafBin PassAll3GHz       20:
          "Good DUTs at 3GHz",Pass3GHz;
        LeafBin FailCache3GHz     21:
          "Cache Fails at 3GHz",  Fail3GHz;
        LeafBin FailSBFT3GHz      22:
          "SBFT Fails at 3GHz",   Fail3GHz;
        LeafBin FailLeakage3GHz   23:
          "Leakages at 3GHz",     FailLeakage;
        LeafBin PassAll2_8GHz     24:
          "Good DUTs at 2.8GHz",  Pass2_8GHz;
        LeafBin FailCache2_8GHz   25:
          "Cache Fails at 2.8GHz", Fail2_8GHz;
        LeafBin FailSBFT2_8GHz26:
          "SBFT Fails at 2.8GHz",Fail2_8GHz;
        LeafBin FailLeakage2_8GHz 27:
          "Leakages at 2.8GHz",   FailLeakage;
    }

   # The keyword SortBinGroup identifies one of the preceding
   # BinGroup's as the one that is used for the sorter.  The bin
   # numbers of bins of that group will be used by the hardware
   # for sorting.
   SortBinGroup = HardBins;
}
```

```
# ------------------------------------------------------------
# File testplan.tpl
# ------------------------------------------------------------

# Various declarations
...


# Counter declarations. Counters are variables that are
# incremented during the execution of a test.  They are
# UnsignedIntegers that are initialized to zero.
Counters {PassCount, FailCount}

...

# A Flow and a FlowItem in it that uses Counters
Flow FlowTest1
{
    FlowItem Xxx SomeTest
    {
        Result 0
        {
            IncrementCounters PassCount;
            ...
        }

        Result 1
        {
            IncrementCounters FailCount;
            ...
        }
    }

    ...
}
```

11

The Pre Headers Syntax

The pre-headers syntax provides custom extensibility to *TPL* and declaration of *TPL* classes, methods and arguments that is used for compiler validation of proper invocation of a *TPL* method and method introspection. *TPL* classes may be *flowables* whose object instantiations determine execution *flow* of *tests* within a *test plan* or *tests* whose object instantiations execute *patterns* against the DUT.

11.1 Requirements upon Pre Headers

1. The syntax:

   `[`**`TestClass`** `|` **`FlowableClass`**`]=[class_name |` **`Test`**`]` **`TestClassDLL`**`=dll_name`

   `PublicBases` `[base_name |` **`Test`**`][, base_name |` **`Test`**`]` $|_{\text{optional,repeat}}$

   shall declare the *test* (*flowable)* class `Test(class_name)` whose implementation resides in `dll_name`.dll and which derives from the class(es) `base_name` and/or **`Test`**

2. The syntax:

   `[`**`Enum`** `|` **`ExternalEnum`**`] enum = member[, member]`$|_{\text{optional,repeat}}$

   shall define the enumerated list `enum` of values `member` to be globally accessible in the test plan.

3. The *test/flowable class* syntax:

```
(1) Parameters
    {
(2)  [[user_type | Integer | UnsignedInteger | String | Voltage | VoltageSlew | Current
     | Power | Time | Length | Frequency | Resistance | Capacitance | TestCondition |
       PatternList]
(3)    param_name
      {
(4)      Cardinality=[1 | 0-1 | 1-n | 0-n];
(5)    | Attribute=attribute;
(6)    | SetFunction=|optional _name [Implement]|optional ;
(7)    | Default = expression;
(8)    | Description = “descriptor”; | GuiType = “type”; | Choices = list; |repeat
   }]
                  |
(9)    ParamGroup group_name
      {
(10)     Cardinality=[number | 0-1 | 1-n | 0-n]; |
(11) |   Attribute=attribute;                     |
(12) |  SetFunction=accessor [Implement]|optional ; |repeat
(13) [[user_type | Integer | UnsignedInteger | String | Voltage | VoltageSlew | Current |
     | Power | Time | Length | Frequency | Resistance | Capacitance | TestCondition |
       PatternList]
 (14)   param_name { Description = description; }                                      |repeat
      }
                  |
 (15) Enum  enum = member[, member]|optional,repeat
   }
 (16) CodeTemplate
```

shall declare a parameter(s) to have:

(2) type: `user_type|Integer|UnsignedInteger|String|Voltage|VoltageSlew|Current|Power|Time|Length|Frequency|Resistance|Capacitance|TestCondition|PatternList`

where `user_type` is a *custom type*

(3) designation `param_name`

(4) singularity of `1` or optionality, `0-1`, membership in a list with at least one element, `1-n`,or membership in a list that may be empty indicating optionality

(5) corresponding C++ class member variable `attribute`

(6) corresponding C|++ set accessor method `accessor` for which *TOS* shall provide default and minimal implementation *iff* `Implement` is specified

(7) default value `expression`

(8) string descriptor `descriptor`, GUI type `type`,type

(9) membership of the parameter group `group_name`

(15) status as a finite enumeration `enum` of permissible values `member`

(16) Inlined C++, **`CodeTemplate`**, delimited by **`CPlusPlusBegin`** and **`CPlusPlusEnd`**

4. The syntax:

   Functions = container_name;

   shall declare the container container_name spanning a set of *user functions*

5. The *user function* syntax:

   [**Void** | **Integer** | **UnsignedInteger** | **String** | **Voltage** | **VoltageSlew** | **Current** | **Power** | **Time** | **Length** | **Frequency** | **Resistance** | **Capacitance**]

   function_name(argument[[array_size]]$|_{\text{optional}}$[, argument[[array_size]]$|_{\text{optional}}$] $|_{\text{repeat}}$)

   shall declare the *user function* function_name with return type Void|Integer|UnsignedInteger|String|Voltage|VoltageSlew|Current|Power|Time|Length|Frequency|Resistance|Capacitance

   and argument(s) argument where argument is strongly typed to be Integer|UnsignedInteger|String|Voltage|VoltageSlew|Current|Power|Time|Length|Frequency|Resistance|Capacitance

   and may be an array of size array_size

6. The pre-headers syntax for *classes* shall reside in files with extension *ph*

7. The pre-headers syntax for *custom functions* shall reside in files with extension *fh*

11.2 Usage Example

The following is a usage example of the pre headers syntax:

```
Version 1.0;
#
# OTPL Parameterization specification pre-header for FunctionalTest
#
Import Test1.ph;              # For base class Test1
Import Test2.ph;              # For base class Test2
TestClass = MyFunctionalTest; # The name of this test class
PublicBases = Test1, Test2;   # List of public base classes
# The parameters list:
Parameters
{
    # The following declaration specifies that a MyFunctionalTest has
    #   - a parameter of OTPL type PList
    #   - [represented by C++ type OASIS::PatternTree]
    #   - stored in a member named m_pPatList
    #   - a function to set it named setPatternTree.
    #   - a parameter description for the GUI to use as a tool tip
```

```
    PList PListParam
    {
       Cardinality = 1;
       Attribute = m_pPatList;
       SetFunction = setPatternTree;
       Description = "The PList parameter for MyFunctionalTest";
    }
    #
    # The following declaration specifies that a MyFunctionalTest has
    #   - 1 or more parameters of OTPL type TestCondition
    #   - [represented by C++ type OASIS::TestCondition]
    #   - stored in a member named m_testCondnsArray
    #   - a function to set it named addTestCondition.
    #   - a parameter description for the GUI to use as a tool tip
    # The [implement] clause causes the translation phase of OTPL to
    # generate a default implementation of this function.
    #
    TestCondition TestConditionParam
    {
       Cardinality = 1-n;
       Attribute = m_testCondnsArray;
       SetFunction = addTestCondition [Implement];
       Description = "The TestCondition parameter for MyFunctionalTest";
    }

}


# ------------------------------------------------------------
# File MyFunctions.fh
#
# OTPL Parameterization specification pre-header for MyFunctions
# ------------------------------------------------------------

Version 1.0;

          Functions = MyFunctions;    # The name of the DLL which contains the      #
                                        following functions. The dLL should not
                                      # contain the extention

# Declare the following C++ function in the
# MyFunctions namespace to determine the minimum
# of two values.
#  // Return the minimum of x, y
#  double MyFunctions::Min
#            (ITestPlan* pITestPlan,int& x, int& y);
Integer Min(Integer x, Integer y);

# Declare the following C++ function in the
# UserRoutines namespace to return the average of
# an array.
#  // Return the average of the array
#  double MyFunctions::Avg
#            (ITestPlan* pITestPlan, double* a, const int a_size);
# The C++ function will be called with a and a'Length
Double Avg(Double a[], Integer a_size);

# Declare the following C++ function in the
# UserRoutines namespace to print the dut id
# and a message
#  // Return the average of the array
#  double MyFunctions::Print
#            (ITestPlan* pITestPlan, String* msg);
# The C++ function will be called with a and a'Length
Void Print(String msg);
```

12

The Flowables and Flow Syntax

Flowables are generic base types from which *test* objects and test execution *flow* objects may be subclassed and initialized by argument list. A further category of *flows* is dedicated to execution on the *system controller*.

Flows are named sets of *flow items*. *Flow items* are named mappings of *flowable* return code (result) to actions that increment values, execute functions, set properties and then branch to another part of the *test plan*. *Flow items* execute the *flowable* object whose return code is mapped.

12.1 Requirements upon Flowables and Flows

1. The syntax:

   `[Flowable | Test] type object`

   shall instantiate a *flowable* or *test* object named `object` of type `type` whose argument assignment list is delimited by **`{`** and **`}`**

2. The *flowables* syntax:

   `argument=value; | group{ argument=value [, argument=value]`$|_{\text{repeat}}$ `}`

   shall assign `value` to the argument `argument` of `object` where:

   - `value` is a string, signed or unsigned integer or floating point number
   - `argument` may belong to the parameter group `group`

3. The syntax:

```
(1) Flow flow_name
    {
(2)   FlowItem item_name flowable_name
      {
(3)     | Result result[:result1]|optional [, result[:result1]|optional]|optional,repeat
        | {
(4)     |   dll_name::function( argument[, argument] |optional,repeat );
(5)     | IncrementCounters counter[, counter] |optional,repeat;
(6)     | SetBin bin_name;
(7)     | Property property_name = expression;
(8)     |  [Return number; | Goto item_name;]
  repeat| }                                                          | repeat
    }
   }
```

shall

(1) create the *flow* `flow_name`

(2) containing *flow item* `item_name` that executes the *flowable* or *test* `flowable_name` and for the return value(s):

(3) `result (to result1)`:

(4) executes custom function `function` in `dll_name`.dll with argument(s) `argument`

(5) increments counter(s) `counter` ***or***

(6) increments the value of the Bin Definitions leaf bin `bin_name` ***or***

(7) assigns `expression` to the previously defined property `property_name`

(8) ***and*** returns to caller with code `number` ***or*** branches to the *flow item* `item_name`

4. The syntax:for *flowables* shall reside in pre-header files

5. The syntax for *flows* shall reside in *test plans*

12.2 Usage Example

The following is a usage example of the *flow* syntax:

```
# A Flow consists of a number of FlowItems and transitions
# between them.  FlowItems have names which are unique in
# the enclosing Flow, execute a "Flowable" object, and then
# transition to another FlowItem in the same enclosing Flow.
#
# Flowable objects include Tests and other Flows.  When
# a Flowable object executes, it returns a numeric Result
# which is used by the FlowItem to transition to another
# FlowItem.  As a result of this, both Tests and Flows
# terminate by returning a numeric Result value.
#
# FlowTest1 implements a finite state machine for the
# Min, Typ and Max flavors of MyFunctionalTest1.  On
# success it tests Test1Min, Test1Typ, Test1Max
# and then returns to its caller with 0 as a successful
# Result.  On failure, it returns 1 as a failing Result.
#
# Assume that the tests MyFunctionalTest1Min, ... all
# return a Result of 0 (Pass), 1 and 2 (for a couple
# of levels of failure).  The Transition Matrix of the
# finite state machine implemented by FlowTest1 is:
# ------------------------------------------------------------
#                  Result 0         Result 1  Result 2
# ------------------------------------------------------------
#  FlowTest1_Min    FlowTest1_Typ    return 1  return 1
#  FlowTest1_Typ    FlowTest1_Max    return 1  return 1
#  FlowTest1_Max    return 0         return 1  return 1
#
# where the IFlowables run by each FlowItem are:
#      FlowItem         IFlowable that is run
#      FlowTest1_Min    MyFunctionalTest1Min
#      FlowTest1_Typ    MyFunctionalTest1Typ
#      FlowTest1_Max    MyFunctionalTest1Max
#
Flow FlowTest1
{
    # This FlowItem represents a state named FlowTest1_Min.
    # It runs the test MyFunctionalTest1Min.  When that completes
    # it will:
    #   - Result 0: Increment the PassCount counter, and
    #       go to FlowItem FlowTest1_Typ.
    #   - Results 1, 2: Increment the FailCount counter,
    #       and return.
    FlowItem FlowTest1_Min MyFunctionalTest1Min
    {
        Result 0
        {
            Property PassFail = "Pass";
            IncrementCounters PassCount;
            GoTo FlowTest1_Typ;
        }

        Result 1,2
        {
            Property PassFail = "Fail";
            IncrementCounters FailCount;
            Return 1;
        }
    }
```

```
        FlowItem FlowTest1_Typ MyFunctionalTest1Typ
        {
            Result 0
            {
                Property PassFail = "Pass";
                IncrementCounters PassCount;
                GoTo FlowTest1_Max;
            }

            Result 1,2
            {
                Property PassFail = "Fail";
                IncrementCounters FailCount;
                Return 1;
            }
        }

        # Likewise for FlowTest1_Max
        FlowItem FlowTest1_Max MyFunctionalTest1Max
        {
            Result 0
            {
                Property PassFail = "Pass";
                IncrementCounters PassCount;
                Return 0;
            }

            Result 1,2
            {
                Property PassFail = "Fail";
                IncrementCounters FailCount;
                Return 1;
            }
        }
    }
```

13

Run Result Map Syntax

A run result map comprises correlation between a *flow* return value and a string that is used to describe the result.

13.1 Requirements upon Run Result Maps

1. The syntax:

```
RunResultMap
{
    result[:result1] = "description"; |optional,repeat
    Default = "default_description";
}
```

shall map *flow* return value(s) `result (to result1)` to the string descriptor `description` and any remaining return values to `default_description`

13.2 Usage Example

The following is a usage example of the *run result map* syntax:

```
RunResultMap
{
    -108:-101 = "Fail 1 GHz";
    -208:-201 = "Fail 2 GHz";
    -308:-301 = "Fail 3 GHz";
    -408:-401 = "Fail 4 GHz";
    -508:-501 = "Fail 5 GHz";

    -1 = "All Fail";

    101:108 = "Pass 1 GHz";
    201:208 = "Pass 2 GHz";
    301:308 = "Pass 3 GHz";
    401:408 = "Pass 4 GHz";
    501:508 = "Pass 5 GHz";

    0 = "All Pass";

    Default = "Uninterpreted Run Result";
}
```

The above RunResultMap specifies that if a result value r is in the interval $-108 \leq r \leq -101$ it will be interpreted by the string “Fail 1 GHz”. Likewise, a value 0 is interpreted as “All Pass”. Finally, any integer that is not explicitly mapped is interpreted by the default value “Uninterpreted Run Result”.

14

The Test Plan Syntax

Test plans initialize and control the flow of *tests* that execute *patterns* against the DUT.

14.1 Requirements upon Test Plans

1. The syntax:

   `Version` `version;`

   shall specify the *TOS* version `version`

2. The syntax:

   `Import` `"file";`

   shall insert the contents of `file`

3. The syntax:

   `[`**`TestPlan`** `test_plan |` **`SysCFlows`** `sysc_flow];`

   shall declare a *test plan* with designation `test_plan` or *test plan* `sysc_flow` that comprises *flowable declarations* and *flows* **and only** *flowable declarations* and *flows* that manipulate the *system controller* prior to loading of *test plans* to the *site controllers*

4. The ***site controller*** *test plan* syntax:

   `DUTType` `"dut_type";`

   shall specify that the *test plan* shall execute against a DUT whose type is `dut_type`

5. The *site controller test plan* syntax:

   `PListDefs``{ [file:pattern_list | variable]`$|_{\text{repeat}}$`}`

   shall declare that the *pattern list* `pattern_list` in file `file` shall be used by the *test plan* and the variable `variable` serve as repository for a *pattern list* name specified at runtime

.

6. The ***site controller*** *test plan* syntax:

   `SocketDef` `= "file";`

   shall specify that the *socket mapping* used by the *test plan* resides in socket file `file`

7. The ***site controller*** *test plan* syntax:

   `OfflineDef` `= "file";`

   shall specify that the *simulation configuration* file used by the *test plan* resides in `file`

8. The ***site controller*** *test plan* syntax:

   `[ user_var; | counter; | test_condition; | flowable; | bin_def; | custom_type; | test_condition_group; | spec_set | test_flow;]`$|_{\text{repeat}}$

   shall declare the local *user var(s)* `user_var`, *counter(s)* `counter`, *flowable* class(es) `flowable`, *bin definition(s)* `bin_def`, *test condition group(s)* `test_condition_group` , *specification set(s)* `spec_set`, *flow* class(es) `test_flow` in accordance with the syntax for such declarations and as alternative to inclusion by **`Import`**

9. The ***site controller*** or ***system controller*** *test plan* syntax:

   `flowable` $|_{\text{repeat}}$

   `flow` $|_{\text{repeat}}$

   shall declare the *flowable* class(es) `flowable` and create the *flow(s)* `flow`

10. The ***site controller*** or ***system controller*** *test plan* syntax:

    `FlowDefs``{ predefined_flow = flow}`

    shall specify assign *flow* `flow` to the predefined flow entry point `predefined_flow` currently supported to be: `CfgPLLoadFlow`, `InitFlow`, `LotStartFlow`/`LotEndFlow`, `DUTChangeFlow`, `TestPlanStartFlow`, `TestPlanEndFlow`, `TestStartFlow`, `TestEndFlow`, `MainFlow`, `TestFlow`

11. The ***site controller*** *test plan* syntax:

    `EndSequence` `{ test_condition [, test_condition ]`$|_{\text{repeat}}$

    shall specify that *test condition(s)* `test_condition` shall apply at end of *test plan*.

12. The *test plan* syntax shall reside in files with extension *tpl.*

14.2 Usage Example

The following is a usage example of the *test plan* syntax:

```
# ------------------------------------------------------------
# File mySimpleTestPlan.tpl
# ------------------------------------------------------------

Version 1.0;

# This is how a pin file would be imported.
#   Import xxx.pin;

# Constants and variables giving limiting values.
Import limits.usrv;

# Import test condition groups
Import myTestConditionGroups.tcg;

# Import some bin definitions.
Import bins.bdefs;

# Import functional tests
Import FunctionalTest.ph;

#------------------------------------------------------------
# Start of the test plan
#------------------------------------------------------------

# The name of this testplan
TestPlan OTPLSample;

# The type of DUT
DUTType "74LS245";

# This block defines Pattern Lists file-qualified names and
# Pattern List variables that are used in Test declarations.
# Pattern list variables are deferred till customization is
# examined.
PListDefs
{
   # File qualified pattern list names
   list1.plist:plist1,
   list2.plist:plist2
}
```

```
# The socket for the tests in this test plan (this is not imported,
# but resolved at activation time):
SocketDef = socket.soc;


# Declare some user variables inline
UserVars
{
    # String name for current test
    String CurrentTest = "MyTest";
}


TestCondition TC1Min
{
    TestConditionGroup = TCG1;
    Selector = min;
}

TestCondition TC1Typ
{
    TestConditionGroup = TCG1;
    Selector = typ;
}

TestCondition TC1Max
{
    TestConditionGroup = TCG1;
    Selector = max;
}

# Likewise for TC2Min, TC2Typ, TC2Max ...


#
# Declare a FunctionalTest.  'FunctionalTest' refers to a C++
# test class that runs the test, and returns a 0, 1 or 2 as
# a Result.  The Test Condition Group TCG1 is selected with
# the "min" selector by referring to the TC1Min TestCondition.
#
Test FunctionalTest MyFunctionalTest1Min
{
    PListParam = plist1;
    TestConditionParam = TC1Min;
}
```

```
# Another FunctionalTest selecting TCG1 with "typ"
Test FunctionalTest MyFunctionalTest1Typ
{
    PListParam = plist1;
    TestConditionParam = TC1Typ;
}

# Another FunctionalTest selecting TCG1 with "max"
Test FunctionalTest MyFunctionalTest1Max
{
    PListParam = plist1;
    TestConditionParam = TC1Max;
}

# Now select TCG2 with "min"
Test FunctionalTest MyFunctionalTest2Min
{
    PListParam = plist2;
    TestConditionParam = TC2Min;
}

# Likewise for TCG2 with "typ" and TCG2 with "max"
Test FunctionalTest MyFunctionalTest2Typ
{
    PListParam = plist2;
    TestConditionParam = TC2Typ;
}

Test FunctionalTest MyFunctionalTest2Max
{
    PListParam = plist1;
    TestConditionParam = TC2Max;
}


#
# At this time the following Test objects have been defined
#      MyFunctionalTest1Min
#      MyFunctionalTest1Typ
#      MyFunctionalTest1Max
#      MyFunctionalTest2Min
#      MyFunctionalTest2Typ
#      MyFunctionalTest2Max
#
#
# Counters are variables that are incremented during the
# execution of a test.  They are UnsignedIntegers that are
# initialized to zero.
#
Counters {PassCount, FailCount}
```

```
#
# Flows can now be presented.  A Flow is an object that
# essentially represents a finite state machine which
# can execute "Flowables", and transition to other flowables based
# on the Result returned from executing a Flowable.  A Flow can also
# call another flow.
#
# A Flow consists of a number of FlowItems and transitions
# between them.  FlowItems have names which are unique in
# the enclosing Flow, execute a "Flowable" object, and then
# transition to another FlowItem in the same enclosing Flow.
#
# Flowable objects include Tests and other Flows.  When
# a Flowable object executes, it returns a numeric Result
# which is used by the FlowItem to transition to another
# FlowItem.  As a result of this, both Tests and Flows
# terminate by returning a numeric Result value.
#
# FlowTest1 implements a finite state machine for the
# Min, Typ and Max flavors of MyFunctionalTest1.  On
# success it tests Test1Min, Test1Typ, Test1Max
# and then returns to its caller with 0 as a successful
# Result.  On failure, it returns 1 as a failing Result.
#
# Assume that the tests MyFunctionalTest1Min, ... all
# return a Result of 0 (Pass), 1 and 2 (for a couple
# of levels of failure).  The Transition Matrix of the
# finite state machine implemented by FlowTest1 is:
# ------------------------------------------------------------
#                  Result 0         Result 1  Result 2
# ------------------------------------------------------------
#  FlowTest1_Min    FlowTest1_Typ    return 1  return 1
#  FlowTest1_Typ    FlowTest1_Max    return 1  return 1
#  FlowTest1_Max    return 0         return 1  return 1
#
# where the IFlowables run by each FlowItem are:
#      FlowItem         IFlowable that is run
#      FlowTest1_Min    MyFunctionalTest1Min
#      FlowTest1_Typ    MyFunctionalTest1Typ
#      FlowTest1_Max    MyFunctionalTest1Max
#
Flow FlowTest1
{
    # This FlowItem represents a state named FlowTest1_Min.
    # It runs the test MyFunctionalTest1Min.  When that completes
    # it will:
    #   - Result 0: Increment the PassCount counter, and
    #       go to FlowItem FlowTest1_Typ.
    #   - Results 1, 2: Increment the FailCount counter,
    #       and return.
    FlowItem FlowTest1_Min MyFunctionalTest1Min
    {
        Result 0
        {
            Property PassFail = "Pass";
            IncrementCounters PassCount;
            GoTo FlowTest1_Typ;
        }

        Result 1,2
        {
            Property PassFail = "Fail";
            IncrementCounters FailCount;
            Return 1;
        }
    }
```

```
    FlowItem FlowTest1_Typ MyFunctionalTest1Typ
    {
        Result 0
        {
            Property PassFail = "Pass";
            IncrementCounters PassCount;
            GoTo FlowTest1_Max;
        }

        Result 1,2
        {
            Property PassFail = "Fail";
            IncrementCounters FailCount;
            Return 1;
        }
    }

    # Likewise for FlowTest1_Max
    FlowItem FlowTest1_Max MyFunctionalTest1Max
    {
        Result 0
        {
            Property PassFail = "Pass";
            IncrementCounters PassCount;
            Return 0;
        }

        Result 1,2
        {
            Property PassFail = "Fail";
            IncrementCounters FailCount;
            Return 1;
        }
    }
}

#
# FlowTest2 is similar to FlowTest1.  It implements a
# finite state machine for the Min, Typ and Max flavors
# of MyFunctionalTest2.  On success it tests Test2Min,
# Test2Typ, Test2Max and then returns to its caller with
# 0 as a successful Result.  On failure, it returns 1 as
# a failing Result.
#
# Assume that the tests MyFunctionalTest2Min, ... all
# return a Result of 0 (Pass), 1 and 2 (for a couple
# of levels of failure).  The Transition Matrix of the
# finite state machine implemented by FlowTest1 is:
# ------------------------------------------------------------
#                   Result 0         Result 1  Result 2
# ------------------------------------------------------------
#  FlowTest2_Min    FlowTest2_Typ    return 1  return 1
#  FlowTest2_Typ    FlowTest2_Max    return 1  return 1
#  FlowTest2_Max    return 0         return 1  return 1
#
```

```
# Where the IFlowables run by each FlowItem are:
#      FlowItem         IFlowable that is run
#      FlowTest2_Min    MyFunctionalTest2Min
#      FlowTest2_Typ    MyFunctionalTest2Typ
#      FlowTest2_Max    MyFunctionalTest2Max
#
Flow FlowTest2
{
   # ...
}


#
# Now the FlowMain, a main flow can be presented.  It
# implements a finite state machine that calls FlowTest1
# and FlowTest2 as below:
# -----------------------------------
#               Result 0     Result 1
# -----------------------------------
#  FlowMain_1FlowMain_2return 1
#  FlowMain_2return 0     return 1
#
# Where the IFlowables run by each FlowItem are:
#      FlowItem      IFlowable that is run
#      FlowMain_1    FlowTest1
#      FlowMain_2    FlowTest2
Flow FlowMain
{
   # The first declared flow is the initial flow to be
   # executed.  It goes to FlowMain_2 on success, and
   # returns 1 on failure.
   FlowItem FlowMain_1 FlowTest1
   {
      Result 0
      {
         Property PassFail = "Pass";
         IncrementCounters PassCount;
         GoTo FlowMain_2;
      }

      Result 1
      {
         # Sorry ... FlowTest1 failed
         Property PassFail = "Fail";
         IncrementCounters FailCount;

         # Add to the right soft bin
         SetBin SoftBins.FailSBFT3GHz;

         Return 1;
      }
   }
```

```
    FlowItem FlowMain_2 FlowTest2
    {
        Result 0
        {
            # All passed!
            Property PassFail = "Pass";
            IncrementCounters PassCount;

            # Add to the right soft bin
            SetBin SoftBins.PassAll3GHz;

            Return 0;
        }

        Result 1
        {
            # FlowTest1 passed, but FlowTest2 failed
            Property PassFail = "Fail";
            IncrementCounters FailCount;

            # Add to the right soft bin
            SetBin SoftBins.FailCache3GHz;

            Return 1;
        }
    }
}

#
# The FlowDefs section specifies the selection of previously
# declared Flows for various purposes.  This section has the
# form:
#   FlowDefs
#   {
#       <keyword> = <Flow Name>;
#       ...
#   }
# The allowed keywords are:
#   MainFlow
#   InitFlow
#   TestPlanStartFlow
#   TestPlanEndFlow
#   TestStartFlow
#   TestEndFlow
#   SiteLoadFlow
#   LotStartFlow
#   LotEndFlow
#   DUTChangeFlow
#   CfgPLLoadFlow

# TestConditionGroups and TestConditions in support of an EndSequence
# and an EndSequence could be declared here, as in the code sample
# provided earlier.
#
FlowDefs
{
    # Only the main test flow is specified.
    MainFlow = FlowMain;
}
```

The above test plan is structured as follows:

1. First, a version number is provided. This number is used to ensure compatibility with an OTPL compiler version. This is done for all OTPL sub-languages.

2. Then, as in other OTPL sub-languages, a number of imports are declared. These are various files with declarations needed in order to resolve names used in the test plan.
3. Next, the Test Plan name is declared, after which come the inline declarations of the test plan.
4. Next a set of *PListDefs* are declared. These include file-qualified names naming **`GlobalPList`**s from the named files. They also include Pattern List variables. Pattern List variables are variables that can be initialized in custom flowables at execution time. They provide a means of delaying binding tests to actual pattern lists until runtime.
5. Following this a *SocketDef* is provided. This merely informs the TestPlan of the socket file that will be used. Information from the socket file is not used in the TestPlan description.
6. Next, a set of *UserVars* is declared. These include a string.
7. Some *Counters* are then declared, to determine the number of tests passed and failed. Counters are simply variables that are initialized to zero, and incremented at `IncrementCounter` statements. They are different from *Bins*, which have the semantics that only the currently set bin is incremented at the end of the test of a DUT.
8. Next, a series of *Test Conditions* is declared. Each of these specifies a Test Condition Group and a selector. In this example, the Test Condition Groups come from `mytestconditionsgroups.tcg`. However, they could have been inline in the test plan.
9. Next, a series of *Flowables or Tests* is declared. Each of this is of the OASIS known `Test FunctionalTest` which selects a Pattern List and a test condition. Thus for instance, `MyFunctionalTest1Max` selects the test condition `TC1Max` and a pattern list.
10. Following this, three flows are declared, `FlowTest1`, `FlowTest2` and `FlowMain`. Flows run *Flowables.* Flowables include Tests (such as `MyFunctionalTest1Max`) and other flows (such as `FlowTest1` and `FlowTest2`). Each of `FlowTest1` and `FlowTest2` run through the minimum, typical and maximum versions of Test1 and Test2 respectively. The flow `FlowMain` calls the earlier declared flows, `FlowTest1` and then `FlowTest2`.
11. Finally, the TestFlow event is assigned to the FlowMain Flow. Thus the flow `FlowMain` is the one that will be executed by this test plan when a user chooses to Execute this plan.

15

Appendix A: TPL Grammar in Backus-Naur Notation

This appendix presents formal file grammars that can be parsed by a LALR(1) parser[1], such as one generated by the yacc parser generator. The grammars listed make use of a stylized notation for expressing the form of the file contents; this Appendix describes the notation.

When specifying the grammar, terminal symbols are shown in **`bold typewriter`** type, and are to appear in the input exactly as written. Non-terminal symbols are shown in *italic* type; they are spelled beginning with a letter, and can be followed by zero or more letters, digits, or hyphens:

symbol1 symbol-2 symbol-3

Syntactic definitions are introduced by the name of the non-terminal being defined followed by a colon. One or more alternatives then follow on successive lines:

non-terminal :
    alternative-1
    alternative-2

Below is shown a non-terminal with two alternatives, one of which is the empty alternative, and the other a very long alternative. The very long alternative that takes more than a line gets continued to the next line by a '\' at the end of the first line.

non-terminal :
    <empty>
    this-is-a-very-long-non-terminal it-is-followed-by-an-even-longer-non-terminal \
        and-then-there-is-an-even-longer-next-line-non-terminal

When the words "one of" follow the colon, this signifies that each of the terminal symbols following on one or more lines is an alternative definition:

digit : one of
    `0 1 2 3 4 5 6 7 8 9`

Optional components of a definition are signified by appending the suffix *opt* to a terminal or non-terminal symbol:

[1] A parser that uses a *Look Ahead Left Recursive* parsing technique, with a look-ahead limited to *one* token.

non-terminal :
symbol-1 optional-symbol$_{opt}$

The OTPL grammar is broken up into three sections: the *main* grammar, the *Timing file* grammar and the *TimingMap file* grammar in the sections that follow.

15.1 Main OTPL Syntax

OTPL_unit :
version_info import_list OTPL_decl

version_info :
Version *version_identifier* ***';'***

import_list :
<empty>
import_list import_stmt

import_stmt :
Import *import_items* ***';'***

import_items :
file_name
import_items ***','*** *file_name*

file_name :
filename_segment
file_name ***'.'*** *filename_segment*

OTPL_decl :
user_vars_decls
custom_type_decls
levels_decls
named_spec_set_decls
test_condition_group_decls
bin_defs_decl
pre_header_decl
test_plan_decl
sysc_flows_decl
run_result_map_decl

user_vars_decls :
user_vars_decl
user_vars_decls user_vars_decl

user_vars_decl :
user_vars_decl_header ***'{'*** *user_vars_item_list* ***'}'***

user_vars_decl_header :
`UserVars`
`UserVars` *user_vars_name*

user_vars_item_list :
<empty>
user_vars_item_list user_vars_item

user_vars_item :
elementary_type object_name '=' expression ';'
elementary_type object_name array_size '=' array_initial_value ';'
`Const` *elementary_type object_name '=' expression ';'*
`Const` *elementary_type object_name array_size '=' array_initial_value ';'*
custom_type_instance

elementary_type :
`Integer`
`UnsignedInteger`
`Double`
`String`
`Voltage`
`VoltageSlew`
`Current`
`Power`
`Time`
`Length`
`Frequency`
`Resistance`
`Capacitance`

array_size :
'[' INTEGER_TKN ']'

array_initial_value :
'{' others_clause '}'
'{' element_list '}'
'{' element_list ',' others_clause '}'

element_list :
expression
element_list ',' expression

others_clause :
`Others` *'=' expression*

expression_list :
<empty>
expression
expression_list ',' expression

expression :
expression '+' term
expression '-' term
term

term :
*term '***'** factor*
term '/' factor
factor

factor :
*'**(**' expression '**)**'*
qualified
*qualified '**[**' INTEGER_TKN '**]**'*
type_conversion
function_call
'-' factor
literal

literal :
STRING_LIT_TKN
INTEGER_TKN optional_measurement_units_name
DOUBLE_TKN optional_measurement_units_name

qualified :
field_qualified
*collection_name '**.**' field_qualified*

field_qualified :
identifier
*field_qualified **"->"** identifier*

type_conversion :
*elementary_type '**(**' expression '**)**'*

function_call :
*identifier **"::"** identifier '**(**' expression_list '**)**'*

custom_type_decls :
custom_type_decl
custom_type_decls custom_type_decl

custom_type_decl :
custom_type_decl_hdr custom_type_decl_body

custom_type_decl_hdr :
***CustomType** custom_type_name*

custom_type_decl_body :
*'**{**' custom_type_decl_members '**}**'*

custom_type_decl_members :
custom_type_member
custom_type_decl_members custom_type_member

custom_type_member :

custom_type_member_type field_name ***';'***

custom_type_member_type :
identifier
elementary_type

custom_type_instance :
custom_type_instance_decl custom_type_instance_init

custom_type_instance_decl :
custom_type_name object_name

custom_type_instance_init :
'{' *custom_type_init_list* ***'}'***

custom_type_init_list :
custom_type_init_item
custom_type_init_list ***','*** *custom_type_init_item*

custom_type_init_item :
field_qualified ***'='*** *expression*

levels_decls :
levels_decl
levels_decls levels_decl

levels_decl :
levels_header ***'{'*** *level_items_list* ***'}'***

levels_header :
levels_decl_keywords levels_name

levels_decl_keywords :
Levels
DCParametrics
Calibration

level_items_list :
<empty>
level_items_list level_item

level_item :
level_item_ref ***'{'*** *level_param_defn_list* ***'}'***
Delay *expression* ***';'***
MinDelay *expression* ***';'***

level_item_ref :
pin_or_pingroup_name

level_param_defn_list :
<empty>
level_param_defn_list level_param_defn

level_param_defn :
level_param_name '=' expression ';'
level_param_name '=' slew_expression ';'

slew_expression :
***Slew** '(' expression ',' expression ')'*

named_spec_set_decls :
named_spec_set_decl
named_spec_set_decls named_spec_set_decl

named_spec_set_decl :
named_spec_set_header '{' spec_item_list '}'

named_spec_set_header :
***SpecificationSet** spec_set_name '(' spec_param_list ')'*

spec_param_list :
<empty>
spec_param_name
spec_param_list ',' spec_param_name

spec_item_list :
<empty>
spec_item_list spec_item

spec_item :
elementary_type spec_item_name '=' expression_list ';'

flowable_decl :
flowable_header '{' flowable_param_setter_list '}'

flowable_header :
flowable_indicator flowable_type_name flowable_object_name

flowable_indicator :
Flowable
Test

flowable_param_setter_list :
flowable_param_setter
flowable_param_setter_list flowable_param_setter

flowable_param_setter :
identifier '=' flowable_param_value ';'
param_group_identifier '{' param_group_field_associations '}'

flowable_param_value :
identifier
string_or_signed_literal

string_or_signed_literal :

STRING_LIT_TKN
optional_minus INTEGER_TKN optional_measurement_units_name
optional_minus DOUBLE_TKN optional_measurement_units_name

optional_minus :
<empty>
'-'

param_group_identifier :
identifier

param_group_field_associations :
param_group_field_associations ',' param_group_field_association
param_group_field_association

param_group_field_association :
identifier '=' flowable_param_value

test_condition_decl :
TestCondition *test_cond_name '{' test_condition_params '}'*

test_condition_params :
test_condition_group_param selector_param
selector_param test_condition_group_param
test_condition_group_param

selector_param :
Selector *'=' identifier ';'*

test_condition_group_param :
TestConditionGroup *'=' identifier ';'*

test_condition_group_decls :
test_condition_group_decls_item
test_condition_group_decls test_condition_group_decls_item

test_condition_group_decls_item :
test_condition_decl
test_condition_group_decl

test_condition_group_decl :
test_condition_group_header '{' spec_set tcg_levels_items timings '}'

test_condition_group_header :
TestConditionGroup *test_cond_group_name*

spec_set :
<empty>
reference_to_named_spec_set
local_spec_set

reference_to_named_spec_set :

SpecificationSet *spec_set_name* **';'**

local_spec_set :
local_spec_set_header **'{'** *spec_item_list* **'}'**

local_spec_set_header :
SpecificationSet **'('** *spec_param_list* **')'**

tcg_levels_items :
<empty>
tcg_levels_items tcg_levels_item

tcg_levels_item :
reference_to_named_levels
local_levels

reference_to_named_levels :
levels_ref_keywords levels_name **';'**

levels_ref_keywords :
Levels
DCParametrics
Calibration
DynCalibration

local_levels_decl_keywords :
Levels
DCParametrics
Calibration

local_levels :
local_levels_header **'{'** *level_items_list* **'}'**

local_levels_header :
local_levels_decl_keywords

timings :
<empty>
Timings **'{'** *timing_item* **'}'**
Timings **'{'** *timing_item timing_map_item* **'}'**
Timings **'{'** *timing_map_item timing_item* **'}'**

timing_item :
Timing **'='** *identifier* **';'**

timing_map_item :
TimingMap **'='** *identifier* **';'**

identifier_or_file_qualified :
identifier
file_qualified

file_qualified :

file_name ***':'*** *identifier*

bin_defs_decl :
BinDefs ***'{'*** *bin_group_decls sort_bin_group_specifier* ***'}'***

bin_group_decls :
bin_group_decl
bin_group_decls bin_group_decl

bin_group_decl :
BinGroup *bin_header* ***'{'*** *bin_group_items* ***'}'***

bin_header :
bin_group_name

bin_group_items :
bin_group_item
bin_group_items bin_group_item

bin_group_item :
Bin *bin_name INTEGER_TKN* ***':'*** *STRING_LIT_TKN* ***';'***
LeafBin *bin_name INTEGER_TKN* ***':'*** *STRING_LIT_TKN* ***';'***
Bin *bin_name INTEGER_TKN* ***':'*** *STRING_LIT_TKN* ***','*** *bin_name* ***';'***
LeafBin *bin_name INTEGER_TKN* ***':'*** *STRING_LIT_TKN* ***','*** *bin_name* ***';'***

sort_bin_group_specifier :
SortBinGroup ***'='*** *bin_name* ***';'***

pre_header_decl :
flowable_class_prologue flowable_class_params code_template
functions_class_prologue function_def_list
global_enumerations code_template

flowable_class_prologue :
class_name_decl class_dllname_decl class_bases_decl

class_name_decl :
TestClass ***'='*** *identifier_or_test_keywd* ***';'***
FlowableClass ***'='*** *identifier_or_test_keywd* ***';'***

class_dllname_decl :
<empty>
TestClassDll ***'='*** *STRING_LIT_TKN* ***';'***

identifier_or_test_keywd :
identifier
Test

class_bases_decl :
<empty>
PublicBases ***'='*** *public_base_list* ***';'***

public_base_list :
identifier
public_base_list ***','*** *identifier*
Test
public_base_list ***','*** ***Test***

flowable_class_params :
<empty>
Parameters ***'{'*** *flowable_class_param_items* ***'}'***

flowable_class_param_items :
flowable_class_param_item
flowable_class_param_items flowable_class_param_item

flowable_class_param_item :
flowable_class_param_info
flowable_class_param_group_info
flowable_class_enumeration

flowable_class_param_info :
identifier_or_type_name param_name ***'{'*** *flowable_class_param_attr_list* ***'}'***

identifier_or_type_name :
identifier
elementary_type
oasis_class_name

oasis_class_name :
TestCondition
PatternList

flowable_class_param_attr_list :
flowable_class_param_attr
flowable_class_param_attr_list flowable_class_param_attr

flowable_class_param_attr :
cardinality_attr
Attribute *'='* *identifier* ***';'***
SetFunction *'='* *identifier implement_directive* ***';'***
Choices *'='* *choice_list* ***';'***
GuiType *'='* *STRING_LIT_TKN* ***';'***
Default *'='* *flowable_param_value* ***';'***
Description *'='* *STRING_LIT_TKN* ***';'***

cardinality_attr :
Cardinality *'='* *cardinality_value* ***';'***

cardinality_value :
"1"
"0-1"
"1-n"
"0-n"

implement_directive :
    <empty>
        Implement

choice_list :
    optional_minus INTEGER_TKN
        STRING_LIT_TKN
        choice_list ',' optional_minus INTEGER_TKN
        choice_list ',' STRING_LIT_TKN

flowable_class_param_group_info :
    param_group_header '{' param_group_attr_list param_group_field_infos '}'

param_group_header :
    ParamGroup *param_group_name*

param_group_attr_list :
    param_group_attr
        param_group_attr_list param_group_attr

param_group_attr :
    cardinality_attr
        Attribute *'=' identifier ';'*
        SetFunction *'=' identifier implement_directive ';'*

param_group_field_infos :
    param_group_field_info
        param_group_field_infos param_group_field_info

param_group_field_info :
    *identifier_or_type_name param_group_field_name *
        *'{' **Description** '=' STRING_LIT_TKN ';' '}'*

flowable_class_enumeration :
    Enum *enum_name '=' identifier_list ';'*

functions_class_prologue :
    Functions *'=' identifier ';'*

function_def_list :
    function_def
        function_def_list function_def

function_def :
    return_type function_name '(' typed_param_list ')' ';'

return_type :
    Void
        elementary_type

typed_param_list :
    <empty>

typed_param
typed_param_list ',' typed_param

typed_param :
elementary_type param_name
elementary_type param_name '[' ']'

global_enumerations :
global_enumeration
global_enumerations global_enumeration

global_enumeration :
enum_with_external_directive enum_name '=' identifier_list ';'

enum_with_external_directive :
`Enum`
`ExternalEnum`

code_template :
`CodeTemplate`

counters_decl :
`Counters` *'{' counters_list '}'*

counters_list :
identifier
counters_list ',' identifier

plist_defs_decl :
`PListDefs` *'{' plist_defs_list '}'*

plist_defs_list :
file_qualified
plist_defs_list ',' identifier_or_file_qualified

socket_def_decl :
`SocketDef` *'=' file_name ';'*

offline_def_decl :
`OfflineDef` *'=' file_name ';'*

test_flow_decl :
flow_header '{' flow_item_list '}'

flow_header :
`Flow` *flow_name*

flow_item_list :
<empty>
flow_item_list flow_item

flow_item :

flow_item_header '{' result_clause_list '}'

flow_item_header :
`FlowItem` *flow_item_name identifier*

result_clause_list :
result_clause
result_clause_list result_clause

result_clause :
`Result` *run_result_list '{' action_list transition '}'*

run_result_list :
run_result_item
run_result_list ',' run_result_item

run_result_item :
run_result
run_result ':' run_result

action_list :
<empty>
action_list action

action :
routine_call_action
increment_counters_action
property_action
set_bin_action

increment_counters_action :
`IncrementCounters` *identifier_list ';'*

identifier_list :
identifier
identifier_list ',' identifier

property_action :
`Property` *identifier '=' STRING_LIT_TKN ';'*

routine_call_action :
identifier **`"::"`** *identifier '(' expression_list ')' ';'*

set_bin_action :
`SetBin` *bin_group_name '.' bin_name ';'*

transition :
`Return` *run_result ';'*
`GoTo` *flow_item_name ';'*

run_result_map_decl :
`RunResultMap` *'{' run_result_map_entries '}'*

```
run_result_map_entries :
        run_result_map_entry
            run_result_map_entry run_result_map_entries

run_result_map_entry :
        run_result '=' STRING_LIT_TKN ';'
            run_result ':' run_result '=' STRING_LIT_TKN ';'
            Default '=' STRING_LIT_TKN ';'

test_plan_decl :
        test_plan_name_decl test_plan_dut_type_decl \
              test_plan_item_list flow_defs_decl

test_plan_name_decl :
      TestPlan test_plan_name ';'

test_plan_dut_type_decl :
      DutType dut_type_name ';'

test_plan_item_list :
        test_plan_item
            test_plan_item_list test_plan_item

test_plan_item :
        user_vars_decl
            counters_decl
            plist_defs_decl
            socket_def_decl
            offline_def_decl
            flowable_decl
            test_condition_decl
            bin_defs_decl
            custom_type_decl
            test_condition_group_decl
            test_flow_decl
            end_sequence_decl
            run_result_map_decl

flow_defs_decl :
        FlowDefs '{' flow_defs_item_list '}'

flow_defs_item_list :
      <empty>
            flow_defs_item_list flow_defs_item

flow_defs_item :
        standard_flow_name '=' flow_name ';'

end_sequence_decl :
        EndSequence '{' end_sequence_test_conditions '}'
```

end_sequence_test_conditions :
test_cond_name
end_sequence_test_conditions ',' test_cond_name

sysc_flows_decl :
sysc_flows_name_decl sysc_flows_item_list flow_defs_decl

sysc_flows_name_decl :
SyscFlows *sysc_flows_name ';'*

sysc_flows_item_list :
sysc_flows_item
sysc_flows_item_list sysc_flows_item

sysc_flows_item :
flowable_decl
test_flow_decl
run_result_map_decl

bin_name :
STRING_TKN

bin_group_name :
STRING_TKN

dut_type_name :
STRING_TKN
STRING_LIT_TKN

enum_name :
STRING_TKN

field_name :
STRING_TKN

filename_segment :
STRING_TKN
STRING_LIT_TKN
Test

flowable_type_name :
STRING_TKN

flowable_object_name :
STRING_TKN

flow_item_name :
STRING_TKN

standard_flow_name :
STRING_TKN

flow_name :
STRING_TKN

function_name :
STRING_TKN

identifier :
STRING_TKN

levels_name :
STRING_TKN

level_param_name :
STRING_TKN

optional_measurement_units_name :
<empty>
STRING_TKN

object_name :
STRING_TKN

param_name :
STRING_TKN

param_group_name :
STRING_TKN

param_group_field_name :
STRING_TKN

custom_type_name :
STRING_TKN

run_result :
INTEGER_TKN
'-' INTEGER_TKN

spec_item_name :
STRING_TKN

spec_param_name :
STRING_TKN

spec_set_name :
STRING_TKN

test_cond_group_name :
STRING_TKN

test_cond_name :
STRING_TKN

test_plan_name :
STRING_TKN

sysc_flows_name :
STRING_TKN

user_vars_name :
STRING_TKN

version_identifier :
STRING_TKN

collection_name :
STRING_TKN

pin_or_pingroup_name :
STRING_TKN

Below are the definitions of symbols that are otherwise undefined above.

1. <empty>: The empty string. This is used when the nonterminal can be replaced by an empty string. It is useful, for instance, to have an optional item such as *optional_minus.*
2. STRING_TKN: a sequence of characters including alphanumerics and the underscore '_' character. It is usually used for names.
3. STRING_LIT_TKN: a sequence of characters in double quotes, representing a string.
4. INTEGER_TKN: standard representation of an integer in decimal notation.
5. DOUBLE_TKN: standard representations of a floating point decimal number in decimal or scientific notation.

15.2 Timing File Syntax

file-contents:
version-info import-list$_{opt}$ *timing-definitions*

version-info:
`Version` *version-identifier* **`;`**

import-list:
import-list import-stmt

import-stmt:
`Import` *import-items* **`;`**

import-items:
file-name
import-items **`','`** *file-name*

file_name:
timing-identifier
file_name **`'.'`** *timing-identifier*

timing-definitions:
timing-definition
timing-definitions timing-definition

timing_definition:
`Timing` *timing-name* **`'{'`** *timing-block* **`'}'`**

timing-name:
timing-identifier

timing-block:
common-section vendor-section$_{opt}$

common-section:
`CommonSection` **`'{'`** *timing-domain-definitions* **`'}'`**

vendor-section:
`VendorSection` **`'{'`** *vendor-definitions* **`'}'`**

vendor-definitions:

timing-domain-definitions:
timing-domain-definition
timing-domain-definitions timing-domain-definition

timing-domain-definition:
`Domain` *domain-name MDR-specification*$_{opt}$ **`'{'`** *domain-block* **`'}'`**

domain-name:

timing-identifier

MDR-specification :
'[' DataRate *data-rate-multiplicity* **']'**

domain-block:
period-table pin-timings

period-table:
PeriodTable '{' *periods* **'}'**

periods:
period
periods period

period:
Period *period-name* **'{'** *time-expression* **';' '}'**

period-name:
timing-identifier

time-expression:
expression

pin-timings:
pin-timing
pin-timings pin-timing

pin-timing:
Pin *pin-name* **'{'** *pin-timing-block* **'}'**

pin-name:
timing-identifier

pin-timing-block:
default-pin-state$_{opt}$ *waveform-table-definitions pin-option-definitions*$_{opt}$

default-pin-state:
DefaultState '=' ForceDown[2] **;**
DefaultState '=' ForceUp ;
DefaultState '=' ForceOff ;
DefaultState '=' D ;
DefaultState '=' U ;
DefaultState '=' Z ;

waveform-table-definitions:
waveform-table-definition
waveform-table-definitions waveform-table-definition

waveform-table-definition:
`WaveformTable` *waveform-table-name* **`'{'`** *waveforms-block* **`'}'`**

waveform-table-name:
timing-identifier

waveforms-block:
waveform-definitions

waveform-definitions:
waveform-definition
waveform-definitions waveform-definition

waveform-definition:
`'{'` *waveform-character MDR-cycle-specification*$_{opt}$ \
`'{'` *single-waveform-definition* **`'}'`** **`'}'`**
`'{'` *waveform-characters MDR-cycle-specification*$_{opt}$ \
`'{'` *multi-waveform-definition* **`'}'`** **`'}'`**

waveform-characters:
waveform-character **`'/'`** *waveform-character*
waveform-characters **`'/'`** *waveform-character*

MDR-cycle-specification:
`'['` *cycle-number* **`']'`**

single-waveform-definition:
event-blocks

event-blocks:
event-block
event-blocks event-block

event-block:
`event` **`@`** *time-expression* **`,`** *edge-resource* **`;`**
`event` **`@`** *time-expression* **`;`**

multi-waveform-definition:
multi-event-blocks

multi-event-blocks:
multi-event-block
multi-event-blocks multi-event-block

multi-event-block:
events **`@`** *time-expression* **`,`** *edge-resources* **`;`**
events **`@`** *time-expression* **`;`**

events:
`event`
events **`/`** **`event`**

edge-resources:
edge-resource
edge-resources **/** *edge-resource*

edge-resource:
E *edge-number*
Edge *edge-number*

pin-options-definitions:
PinOptions '{' *pin-options* **'}'**

pin-options:
pin-options pin-option
pin-option

pin-option:
compare-mode-definition

compare-mode-definition:
CompareMode = Single;
CompareMode = Multi;

expression :
expression **+** *term*
expression **-** *term*
term

term :
term ***** *factor*
term **/** *factor*
factor

factor :
'(' *expression* **')'**
qualified
qualified **'['** *integer* **']'**
string-literal
positive-integer
positive-integer measurement-units-name
double
double measurement-units-name
function-call
'-' *factor*

qualified :
timing-identifier
qualified **.** *timing-identifier*

function-call :
qualified **(** *expression-list* **)**

measurement-units-name :

The following are the descriptions of undefined non-terminals used in the above grammar:

1. *version-identifier*: A sequence of one or more characters from the set `[0-9.]`, where the first character must be a digit. The version identifier is provided to allow the Timing syntax to change and thus allow the system the ability to decipher from the version number what constructs to expect.

2. *timing-identifier*: A sequence of one or more characters from the set `[a-zA-Z_0-9]`, where the first character must be from the set `[a-zA-Z_]`.

3. *vendor-definitions*: Arbitrary text that is meaningful only to a specific vendor's Timing -object parser.

4. *waveform-character* : A single alpha-numeric character that represents the waveform character in the pattern source file.

5. *time-expression* : An expression used to express the period and uses variable names, mathematical symbols, numbers, braces and engineering units.

6. *event* : One of the standard list of supported event types. See **Error! Reference source not found.** for details.

7. *edge-number* : An integer greater than or equal to 1, where the number indicates the resource number associated with that event.

8. *data-rate-multiplicity*: An integer greater than or equal to 1, where the number indicates the multiplicity of the data rate to use in case of a multi-data-rate pattern. For a double data rate timing this number would be set to 2. In case the tester needs to use a single-data-rate the MDR syntax can be omitted. In the case of Advantest's OPENSTAR™ 250MHz digital module, valid values are 1 & 2.

9. *cycle-number:* An optional number from the set [1-9][0-9]+, where the number indicates that this describes the waveform for that section of a multi-data rate pattern. In a double data rate [1] will define the first (odd) half of a *Double-Rate tester cycle* and [2] indicates the second (even) half. The absence of this option indicates that the pin is configured to be operating in *Normal* mode and there is no multiplexing of *pattern* data.

10. *string-literal*: A sequence of one or more characters from the set `[a-zA-Z_0-9]`, where the first character must be from the set `[a-zA-Z_]`.

11. *integer*: A non-negative integer in decimal notation.

12. *positive-integer*: A positive integer in decimal notation.

13. *comments* : A # signifies a comment and all the text following the # character on that line are ignored by the parser

15.3 TimingMap File Syntax

file-contents:
    version-info import-list$_{opt}$ *timing-map-definitions*

version-info:
    `Version` *version-identifier* **`';'`**

import-list:
     import-list import-stmt

import-stmt:
    `Import` *import-items* **`';'`**

import-items:
    file-name
       import-items **`','`** *file-name*

file_name:
    timing-map-identifier
     file_name **`'.'`** *timing-map-identifier*

version-identifier:
    timing-map-version

timing-map-definitions:
    timing-map-definition
     timing-map-definitions timing-map-definition

timing-map-definition:
    `TimingMap` *timing-map-name* **`'{'`** *timing-map-block* **`'}'`**

timing-map-name:
    timing-map-identifier

timing-map-block:
    timing-map-domain-definitions

timing-map-domain-definitions:
    timing-map-domain-definition
     timing-map-domain-definitions timing-map-domain-definition

timing-map-domain-definition:
    `Domain` *domain-name* **`'{'`** *domain-block* **`'}'`**

domain-name:
    timing-map-identifier

domain-block:
    waveform-maps

waveform-maps:
waveform-map
waveform-maps waveform-map

waveform-map:
`WaveformMap '{'` *waveform-map-block* **`'}'`**

waveform-map-block:
pin-format waveform-map-definitions

pin-format:
`PinFormat '{'` *pin-names* **`'}'`**

pin-names:
pin-name
pin-names **`','`** *pin-name*

pin-name:
timing-map-identifier

waveform-map-definitions:
waveform-map-definition
waveform-map-definitions waveform-map-definition

waveform-map-definition:
waveform-selector-name **`','`** *period-name* **`','`** **`'{'`** *waveform-table-names* **`'}'`**

waveform-selector-name:
timing-map-identifier

period-name :
timing-map-identifier

waveform-table-names:
waveform-table-name
waveform-table-names **`','`** *waveform-table-name*

waveform-table-name:
timing-map-identifier

The following are the descriptions of undefined non-terminals used in the above grammar:

1. *timing-map-version*: A sequence of one or more characters from the set `[0-9.]`, where the first character must be a digit.
2. *timing-map-identifier*: A sequence of one or more characters from the set `[a-zA-Z_0-9]`, where the first character must be from the set `[a-zA-Z_]`.
3. *comments* : A # signifies the beginning of a comment and all the text following the # character on that line are ignored by the parser.