IEEE

1450.3™

IEEE Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450™-1999) for Tester Target Specification

IEEE Computer Society

Sponsored by the
Test Technology Standards Committee

IEEE
3 Park Avenue
New York, NY 10016-5997, USA

7 September 2007

IEEE Std 1450.3™-2007

Recognized as an
American National Standard (ANSI)

IEEE 1450.3™-2007

IEEE Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450™-1999) for Tester Target Specification

Sponsor

Test Technology Standards Committee
of the
IEEE Computer Society

Approved 24 August 2007

American National Standards Institute

Approved 8 March 2007

IEEE SA-Standards Board

Abstract: The STIL environment supports transferring tester-independent test programs to a specific automated testing equipment (ATE) system. Although native STIL data are tester independent, the actual process of mapping the test program onto tester resources may be critical, and it is necessary to be able to completely and unambiguously specify how the STIL programs and patterns are mapped onto the tester resources. TRC (which stands for either tester resource constraints or tester rules checking, depending on the usage) is an extension to the STIL language to facilitate this operation.

Keywords: Tester rules checking (TRC), tester resource reporting, tester resource targeting, tester resource loading

The Institute of Electrical and Electronics Engineers, Inc.
3 Park Avenue, New York, NY 10016-5997, USA

Print: ISBN 0-7381-5517-9 SH95624
PDF: ISBN 0-7381-5518-7 SS95624

Introduction

This introduction is not part of IEEE 1450.3-2007, Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450™-1999) for Tester Target Specification.

STIL is a collection of standards with the base standard being 1450 and the dotted extensions used to define additional syntax for addressing additional areas; i.e., this standard addresses tester rules.

The extensions follow the same conventions as the base standard. The base and the extensions are developed so as to work together; i.e., STIL is a single language that is defined (and has been developed) as separate IEEE standards.

Notice to users

Errata

Errata, if any, for this and all other standards can be accessed at the following URL: http://standards.ieee.org/reading/ieee/updates/errata/index.html. Users are encouraged to check this URL for errata periodically.

Interpretations

Current interpretations can be accessed at the following URL: http://standards.ieee.org/reading/ieee/interp/index.html.

Patents

Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. The IEEE shall not be responsible for identifying patents or patent applications for which a license may be required to implement an IEEE standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Participants

The following is a list of participants in the STIL Working Group.

Contents

IEEE Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450™-1999) for Tester Target Specification

1. Overview

The STIL environment supports transferring tester-independent test programs to a specific automated testing equipment (ATE) system. Although native STIL data are tester independent, the actual process of mapping the test program onto tester resources may be critical, and it is necessary to be able to completely and unambiguously specify how the STIL programs and patterns are mapped onto the tester resources. TRC (which stands for either tester resource constraints or tester rules checking, depending on the usage) is an extension to the STIL language to facilitate this operation.

Figure 1 shows the usage model for tester targeting. The four ways that the TRC statements come into play in the flow of data from design to test are indicated by the circled numbers in the diagram. These four uses are defined as follows:

a) **Tester rules checking:** As early as possible in the process of inserting “Design for Test” logic and generation of test patterns, the rules of the target tester are identified by means of the TRC file defining the target tester.

b) **Tester resource reporting**: As part of the pattern generation process, a report of resources required for the pattern may be created in TRC format. This information is available for test planning purposes, such as 1) when a pattern is for an embedded core to be integrated into a chip or 2) for tester scheduling purposes. Each resource report is associated with a particular STIL file/stream. The resource report data may be a separate file (as implied in the above diagram) or may be included in the STIL pattern file.

c) **Tester resource targeting**: The process of tester targeting is that of adding additional information into the STIL file/streams that specifies how the resources of a given tester are to be assigned. Note the bars on the left side of the diagram, which indicate that this targeting operation can be done in one of three places: 1) by the EDA software that generates the patterns, 2) by software created by the test user, or 3) by the ATE sofware that loads the STIL patterns.

d) **Tester resource loading**: The tester loader is a process that maps the device-oriented STIL data to the resources of the tester. There may or may not be targeting information provided. If targeting information is not present, then the loader is expected to do the job of assigning resources. If targeting information is present, then it is to be used to direct the resource assignment.

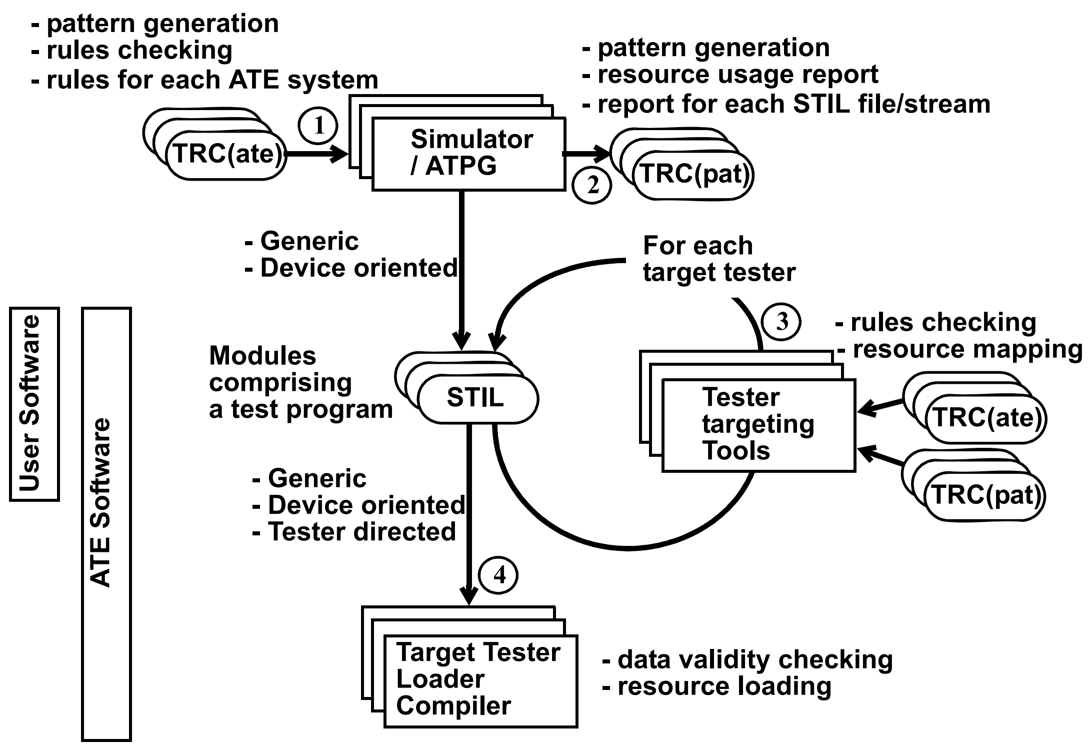


Figure 1—STIL.3 usage model (tester targeting)

Note that the semantics of the various TRC statements change, based on which flow is being addressed; i.e., in a constraint flow, the TRC statements provide a set of rules that are to be applied to a set of data and an error shall be generated if the test pattern data does not conform to the rules; in a report flow, the same TRC statements are providing a summary of test pattern data regardless of whether it conforms to any given TRC constraints. Most TRC statements can be used in either context. The definitions in this standard are written from the constraint point of view.

1.1 Scope

- Define structures in STIL for the specification of resource mapping of ATE hardware architectures. An example of resource mapping is the assignment of tester resources to waveform characters that are used in STIL vectors.
- Define structures in STIL for including ATE-specific instructions in-line with the STIL data.
- Define structures in STIL that allow for “incremental processing” whereby, a set of STIL files may be targeted to multiple ATE systems by allowing separately identified ATE data to coexist.
- Define structures in STIL for defining tester rules checks to ensure that the set of generated STIL files conform to the selected resources on one or more ATE systems.
- Define structures in STIL for the specification of the resources required for the execution of a set of STIL files on a given ATE system.

1.2 Purpose

Transferring “tester independent” test program/pattern data as represented in STIL to a specific ATE system is a desired capability. It is required to be able to completely and unambiguously specify how the STIL program/patterns are mapped onto a specific tester’s resources. Because of the various different use models for the creation and consumption of test data, it is necessary to enable certain operations (such as rules checking) very early in the process. Likewise it is desirable to allow other operations (such as resource allocation) to be done very late in the process. The STIL language extensions enable the user/creator a standard way of specifying and controlling the application of test program/pattern data to specific ATE systems to the extent necessary for each use model scenario.

1.3 TRC limitations

In setting the scope for any standard, some issues are identified and determined not to be defined. The following is a list of issues that are not in the scope of this standard:

a) Violations: The tester-targeting operation naturally leads to situations where a given set of constraints cannot be met. It is not in the scope of this standard to define the content or format of such constraint rule violations.
b) Completeness: The variations in architecture of various ATE systems makes it virtually impossible to address all the nuances and corner cases of each design. It is the intent of this standard to provide a common way of describing key common attributes. This standard is intended to address 80% of the issues involved with TRC operations, because the effort to address the remaining 20% is too complex and would make the standard unwieldy to apply in general.

2. Normative references

The following referenced documents are indispensable for the application of this Standard. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEEE Std 1450™ (STIL.0), IEEE Standard Test Interface Language (STIL) for Digital Test Vectors.[1, 2]

IEEE Std 1450.1™ (STIL.1), IEEE Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450™-1999) for Semiconductor Design Environments.

IEEE Std 1450.2™ (STIL.2), IEEE Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450™-1999) for DC Level Specification.

IEEE Std 1450.6™ (STIL.6), IEEE Standard IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data-Core Test Language (CTL).

3. Definitions

For the purposes of this standard, the following terms and definitions apply. *The Authoritative Dictionary of IEEE Standards Terms* should be referenced for terms not defined in this clause. Additional terminology specific to this standard is found in Annex A.

STIL.0: Refers to IEEE Std. 1450. This base STIL standard is commonly referred to as “dot 0”.

[1]The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[2]IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).

STIL.1: Refers to IEEE Std 1450.1. This extension to the STIL base standard is commonly referred to as "dot 1": Design information specification.

STIL.3: Refers to IEEE Std 1450.3 (i.e., this standard). This extension to the STIL base standard is commonly referred to as "dot 3": Tester target specification.

STIL.6: Refers to IEEE Std 1450.6. This extension to the STIL base standard is commonly referred to as "dot 6": Core test language specification.

TRC: (**A**) (Tester resource constraint) a set of STIL statements (i.e., this standard) that are used to specify either 1) the resources available in a tester or 2) the resources needed in support of a STIL pattern. (**B**) (Tester rule check) an operation performed by an application to verify the consistency between a pattern and a set of tester resource constraints.

4. Structure of this standard

This document is an adjunct to IEEE Std 1450. The conventions established and defined in IEEE Std 1450 are used in this document and are included verbatim below.

Many clauses in this document add additional constructs to existing clauses in the IEEE Std 1450 document and are so identified in the title. The constructs defined in this document are limited to the Environment block as defined by IEEE Std 1450.1. All clauses in this document are normative unless specifically identified as "Informative." Example code is provided within each clause. More complete examples are provided in the annexes, which are informative.

4.1 Formats from STIL.0

The following discussion is a copy of the conventions as defined in STIL.0 and followed by this document.

Different fonts are used as follows:

a) SMALL CAP TEXT is used to indicate user data.
b) `Courier` text is used to indicate code examples.

In the syntax definitions:

a) SMALL CAP TEXT is used to indicate user data.
b) **Bold** text is used to indicate keywords.
c) *Italic* text is used to reference metatypes.
d) () indicates optional syntax that may be used 0 or 1 time.
e) ()+ indicates syntax that may be used 1 or more time.
f) ()* indicates optional syntax that may be used 0 or more times.
g) <> indicates multiple choice arguments or syntax.

In the syntax explanations, the verb "shall" is used to indicate mandatory requirements. The meaning of a mandatory requirement varies for different readers of the standard:

a) To developers of tools that process STIL (readers), “shall” denotes a requirement that the standard imposes. The resulting implementation is required to enforce this requirement and to issue an error if the requirement is not met by the input.
b) To developers of STIL file/stream (writers), “shall” denotes mandatory characteristics of the language. The resulting output must conform to these characteristics.
c) To the users of STIL, “shall” denotes mandatory characteristics of the language. Users may depend on these characteristics for interpretation of the STIL source.

The language definition clauses contain statements that use the phrase “it is an error” and “it may be ambiguous.” These phrases indicate improperly defined STIL information. The interpretation of these phrases will differ for the different readers of this standard in the same way that shall differs, as identified in the list above.

4.2 Additional formatting conventions

The following items are new conventions that are used in the various paragraph types:

a) In the Syntax definitions, each statement or group of statements is identified by a syntax line number (in paranthesis at the right side of the page). These numbers are to be referenced to the definitions that follow, which contain the syntax line numbers in parenthesis on the left side of the page.
b) In the Syntax definitions, underlined attribute keywords indicate the default when none in a group is selected.
c) In the code example sections of the document, the text is in `Courier` font and each line contains a line number followed by a colon (:) at the left-hand side of the page. This line number is for reference only and is not part of the code.
d) In the code examples, user-defined names (for things like signalnames, signal group names, block names, etc.) are in all uppercase with under_score separators. This is to distinguish them from STIL keywords.
e) In the tutorial examples (Annex D through Annex G), circled numbers on the right-hand side of the code examples are used to indicate that a note about the code is included immediately after the example.

4.3 Dependencies on IEEE Std 1450.1

This standard is built on IEEE Std 1450 and relies on some of the syntax as defined in the Design extension (IEEE Std 1450.1). The following list provides the IEEE Std 1450.1 statements that are used by this standard:

— **Clause 5, Expressions constructs:** The additional expression capabilities are used for defining relationships between various TRC attributes (see Annex B).
— **Clause 7, STIL statement:** This statement, which is required at the beginning of every STIL file, is extended to allow specification of the STIL extension standards.
— **Clause 9, Variables block:** This new block is required when fluid attributes are used (see Annex B).
— **Clause 18: Environment block:** The block allows for the definition of STIL information pertaining to the environment of application in which a set of test pattern data is used. TRC is one such environment, and hence, all TRC statements are contained within an Environment block.
— **18.2 “MAP_STRING” syntax and 18.3 NameMaps example:** As part of the general Environment block, syntax is defined that defines naming relationships to other environments. Tester channel information makes use of this facility (see Annex C).

5. STIL syntax description

This clause contains extensions to Clause 6 of STIL.0. All constructs and restrictions for Clause 6 of IEEE Std 1450 are in effect here, with the following additions.

5.1 Additional reserved words

Table 1 lists new STIL reserved words defined by this standard and not defined in IEEE Std 1450. Subsequent clauses in this standard identify the use and context of each of these additional reserved words.

Table 1—Additions to STIL reserved words

Assert
ConfigConstant
ParamConstant
Resource

5.2 Keywords used in a TRC block

Table 2 lists new STIL keywords that are used with a TRC block or TRC sub-block. These keywords are not reserved in the general STIL context, but they are available only inside a TRC block or a TRC sub-block.

Table 2—TRC keywords

Accuracy, AllowedWhen, AppliesTo
Base, Block
CharacterContent, CompareEvents, ConditionalStatements, Contents, CoreUsageReady
DeltaChangeVectorData, DriveEvents
AllowedScanPadWaveforms
DCLimits, DCResourceAttributes, DifferentialConfiguration
Environment
FanOut, FormatSelect
Infinite, InOut, InstructionAttributes
Length, LoopAttributes
Macro, Max, MaxCaptureMemory, MaxData, MaxEdgeTime, MaxIO, MaxMask, MaxPeriodGenerators, MaxPeriods, MaxShapes, MaxScanChainLength, MaxIteration, MaxLength, MaxNest, MaxRunTime, MaxScanMemory, MaxSignals, MaxTimeSets, MaxTimingGenerators, MaxVectorMemory, MinAfter, MinBefore, MinCompareWindow, MinCompareToDriveOn, MinDriveOffTime, MinDriveOffToCompare, MinDriveOnTime, MinDrivePulse, MinEdgeReTrigger, MinIteration, MinLength, MinTimeAfterMatch, MinVectorsAfter, MinVectorsAfterMatch, MinVectorsBefore, MinVectorsBetween, Modulus, MultiBitData, MultipleDevices, MultipleSites

Table 2—TRC keywords *(continued)*

NameChecks, NonCyclized, NumberCaptureCycles, NumberData, NumberDCLevels, NumberIO, NumberLevels, NumberMask, NumberPatternUnits, NumberPeriods, NumberShapes, NumberSignals, NumberTesterChannels, NumberTimeSets, NumberVectorsPerShift
PatternAttributes, PatternVariables, PeriodAttributes, PeriodSelectMemory, PerPinConfiguration, PerTimingGenerator, Pragma, ProcedureCalls
Resolution
Scope, SelectWithPeriod, Shape, SignalAttributes, SignalsPer, STILPatterns, SubWaveformDuration, SubWaveformIteration, SystemAttributes
TimeLimits
TRC
UndrivenInOut, Units, Usage
VectorCompression
WaveformDescriptions, WaveformAttributes, WaveformSelectMemory

6. Statement usage and organization by flow

As shown in Figure 1, there are four ways in which TRC blocks and statements can be utilized to accomplish various objectives. Table 3 shows the blocks of data that are pertinent to each of these four usage models for TRC data.

There are no special requirements for sequencing of blocks within a TRC block. A reader is expected to process an entire TRC block prior to interpreting.

A TRC block that contains a PatternReport is typically part of the STIL file/stream for that pattern (or a referenced Include file). The typical placement of this block (or Include) would be near the beginning, right after the definition of Signals, Groups, Specs, and Variables.

A TRC block that contains Constraint data is typically a separate STIL file/stream from the STIL file/stream that contains the pattern data. The constraint data file and the pattern data file would be separately identified by the tool or ATE software that it processing the STIL file/stream(s).

Table 3—STIL/TRC block usage

Block/Statement/Function	Purpose	Constraint[a]	Report[b]	Target[c]	Load[d]
STIL	file header id	X	X	X	X
Environment {}	container block	X	X		X
Environment {TRC { Usage Constraints; } }	specify to use for constraints	X			X
Environment {TRC { Usage PatternReport; } }	specify that data is a report		X		
Environment { TRC { Category {} }	select one or more categories	X			X
Environment { TRC { SystemAttributes {} }}	define system application	X			X
Environment { TRC { NameChecks {} }}	define name constraints	X			
Environment { TRC { PatternAttributes {} }}	define pattern information	X	X		X
Environment { TRC { PeriodAttributes {} }	define period information	X	X		X
Environment { TRC { SignalAttributes {} }}	define signal information	X	X		X
Environment { TRC { WaveformAttributes {} }	define waveform information	X	X		X
Environment { TRC { WaveformDescriptions {} }}	define wave shapes	X	X		X
STIL blocks	any block in STIL			X	X
Environment { NameMaps {} }	map STIL names to other form			X	X
Pragma {}	define ATE native statements			X	X
<<resource_id>> within STIL blocks	define ATE resource mapping			X	X

[a]Constraint = Information that is used to convey constraints for an ATE system, or else a set of arbitrary instructions to be used to constrain the implementation of a chip design or chip test patterns.

[b]Report = Information that is used to report the actual parameters of a given pattern or pattern burst.

[c]Tester Target = Information that is added to a STIL file/stream to specify how it is to be loaded into the resources of a given ATE system.

[d]Tester Loading = Information that is used in a STIL file/stream to control the loading of the resources of an ATE system.

6.1 TRC usage for ATE constraint specification

An ATE specification file is typically a separate file that specifies the attributes of one or more ATE systems. It may contain a set of attributes that is the intersection of several ATE systems, thereby ensuring that a pattern that meets the constraints will run on any ATE system. The file may define a class of ATE systems by utilizing category variables or expressions. The structure of an ATE constraint file is as follows:

```
1: STIL 1.0 { Design 2005; TRC 2007; }
2: Header {
3:   Source "IEEE Std 1450.3-2007";
4:   Ann {* clause 6.1 *}
5: }
6: Environment {
  TRC ATE_1 {
    Usage Constraints;
  }
  TRC ATE_2 {
    Usage Constraints;
  }
} // end Environment
```

6.2 TRC usage for design/pattern constraints

A TRC file that is to be used for constraining the creation of a design or pattern set for that design looks much the same as the ATE constraint file. The difference is that typically only one set of constraints will exist and hence only one Environment block in the file. If there are multiple Environment blocks, then the name of the block is an identifier for a set of constraints that are selected by the design tool. The structure of a file with a single set of constraints is as follows:

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
  Source "IEEE Std 1450.3-2007";
  Ann {* clause 6.2 *}
}
Environment {
  TRC {
    Usage Constraints;
  }
} // end Environment
```

6.3 TRC usage for pattern reporting

The same syntax as is used for specifying resource constraints on a Pattern (as defined in STIL.0) can also be used to report the resource utilization for that Pattern (or PatternBurst). The different application is identified as such by the “Usage PatternReport;” statement in the TRC block. Some resource constraint blocks do not apply in this application, i.e., System and NameChecks. The information could be included in the same STIL file as the pattern or in a separate file. The structure of this type of file is as follows:

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
  Source "IEEE Std 1450.3-2007";
  Ann {* clause 6.3 *}
}
Environment {
  TRC PATNAME_1 {
    Usage PatternReport;
  }
} // end Environment
Environment {
  TRC PATNAME_2 {
    Usage PatternReport;
  }
} // end Environment
```

6.4 TRC usage for tester targetting

Tester targetting is accomplished by annotating a STIL file/stream with additional information that tells an ATE loader/translator how to map the STIL constructs onto the hardware resources of the tester. This information can take many forms, but the following example shows one possibility:

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
  Source "IEEE Std 1450.3-2007";
  Ann {* clause 6.4 *}
}
Signals { SIG[1..5] InOut; }
Environment ATE1 {
  NameMaps { } // specify signal to channel mapping
}
SignalGroups { SIGS = SIG[1..5]; }
Timing basic {
   WaveformTable ONE {
      <<PER1>> Period 500ns; // tag the period resource
      DIR { <<SEQ1>> 01 { 0ns D/U; }} // tag the per pin waveform resource
   }
   WaveformTable TWO {
      <<PER1>> Period 500ns; // use same resourse as wft one
      DIR { <<SEQ2>> 01 { 0ns D/U; }} // use different resource from wft one
   }
Pragma ATE1 {*
  MAP1: ...
  MAP2: ...
*}
Pattern P {
  <<MAP1>> V { SIGS = 10101; } // select MAP1 from pragma ATE1
  <<MAP2>> V { SIGS = LHLHL; } // select MAP2 from pragma ATE1
  <<MAP1>> V { SIGS = 01010; } // select MAP1 from pragma ATE1
}
```

7. STIL statement

This clause contains extensions to Clause 8 of STIL.0.

The STIL statement identifies the primary version of IEEE Std 1450 information contained in a STIL file and the presence of one or more standard Extension constructs. The primary version of STIL is defined in IEEE Std 1450.

The extension to the STIL statement allows for a block containing extension identifiers that allow for additional constructs in the STIL file. Multiple Extension statements may be present to identify the presence of multiple extension environments. The extension name and the extension statements are defined in the individual documents for those standards.

All other constructs and restrictions for Clause 8 of IEEE Std 1450 are in effect here.

7.1 STIL syntax

STIL IEEE_1450_0_IDENTIFIER **{** (1)
    (**TRC** EXT_VERSION;) (2)
} *// end STIL*

(1) **STIL:** A statement at the beginning of each STIL file.

IEEE_1450_0_IDENTIFIER: The primary version of STIL, as identified by IEEE Std 1450.

(2) **TRC**: The specific name of this Extension.

EXT_VERSION: The version of this extension to STIL. This standard is identified by the identifier 2006**.**

7.2 STIL example

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
  Source "IEEE Std 1450.3-2007";
  Ann {* clause 7.2 *}
}
```

8. Variables block extensions

This clause defines extensions to Clause 9, STIL.1.

This clause defines additional statements to the Variables block to support fluid constraint definitions. All statements and capabilities as defined in STIL.1 are unchanged.

Fluid constraints are used in this standard to define how a set amount of a resource may be allocated or shared between multiple uses of that resource. Each use may require different amounts of that resource, and there is an identified limit to the amount of that resource available. Fluid constraints may be used to specify aspects of a tester that are optional (i.e., present or not present) or have varying size options (e.g., memory size, or number of channels). Fluid constraints may be used to specify tradeoffs that may be made at load time to support the specific needs of the application. For example, a tester may be able to use two slow-speed pin channels to make one high-speed pin channel; a tester may be able to trade off vector memory for scan memory. See Annex B for an example of the application of this capability.

This standard does not specify how the tradeoff computation is to be accomplished or what might be the consequences of different tradeoffs. However, the Assert statement (defined below) allows for the checking of the fluid assignments to ensure that they satisfy the requirements of the target test system.

8.1 Variables block syntax

Variables (VARIABLES_DOMAIN) {
    Assert *boolean_expr* ; (1)
    ConfigConstant CONFIG_NAME (ALT_CONFIG_NAME)* ; (2)
    ParamConstant PARAM_NAME ; (3)
}

(1) **Assert**: The Assert statement contains a boolean expression that is expected to be TRUE in order for the rules checking to pass successfully. As defined in 5.6 of STIL.1, a boolean expression can be one that contains a boolean operator (e.g., ==, :==, or <>) or it can be an expression that results in an integer value that is interpreted as a boolean result (i.e., one or greater for TRUE or zero or less for FALSE). If the result of the expression evaluation is FALSE, the application shall report this failure to satisfy this statement. The operands of the boolean expression can be as follows:

a) Named constants that are defined in the current Variables block: IntegerConstant, ConfigConstant, ParamConstant

b) Named constants that are defined in a global (unnamed) Variables block

c) Literal integer values

(2) **ConfigConstant**: This statement defines the name of a value that is determined by the "configuration" of the hardware in a tester. The actual value is to be provided to the TRC process based on knowledge of the target test system. Once defined, the value is then used as a constant value. The constant can be of type integer (see 5.7 of STIL.1 for the definition of allowed integer forms) or real number (see 5.9 of STIL.1 for the definition of allowed real expression forms).

CONFIG_NAME: The name of the hardware configuration parameter. This name may be used in expressions as a defined constant.

ALT_CONFIG_NAME: Zero or more ALT_CONFIG_NAME identifiers may be specified. These names shall reference CONFIG_NAME names that are defined in other ConfigConstant statements that are in scope. The name specifies another resource that is available and capable of satisfying the same requirements and may be used when the CONFIG_NAME resource is exhausted.

(3) **ParamConstant**: This statement defines the name of a value that is assignable by the TRC process. The value chosen is typically a tradeoff with other ParamConstant values and allows the TRC process to optimize the use of resources for a particular device test program. Once defined, the value is then used as a constant value. The constant can be of type integer (see 5.7 of STIL.1 for the definition of allowed integer forms) or a real number (see 5.9 of STIL.1 for the definition of allowed real expression forms).

8.2 Variables example

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
  Source "IEEE Std 1450.3-2007";
  Ann {* clause 8.2 *}
}
Variables {
  // the following illustrates usage of the Assert statement
  IntegerConstant K1 := 99;
  IntegerConstant K2 := 100;
  Assert K1 <> K2;
  Assert K1 + K2 <= 200;
  Assert K1;
  Assert K1 :== K2; // FALSE -- report as ERROR

  // the following illustrates usage of the ConfigConstant and IntegerParm statements
  ConfigConstant NUM_MODULES;
  Assert 0 <= NUM_MODULES <= 39;
  IntegerConstant NUM_SIGNAL_SLOTS := NUM_MODULES * 32;

  ConfigConstant NUM_PATTERN_MEMORY_BITS;
  Assert 0 < NUM_PATTERN_MEMORY_BITS <= 4*1024*1024*1024;

  ParamConstant NUM_800MBPS;
  ParamConstant NUM_400MBPS;
  ParamConstant NUM_SCANIN;
  ParamConstant NUM_SCANOUT;

  IntegerConstant NUM_SIGNALS :=
    NUM_800MBPS + NUM_400MBPS + NUM_ScanIn + NUM_ScanOut;
  Assert 2*NUM_800MBPS + NUM_400MBPS + NUM_ScanIn + NUM_ScanOut
    <= NUM_SIGNAL_SLOTS;

  ParamConstant NUM_SCAN_LOAD_UNLOAD;
  ParamConstant LONGEST_SCAN_SHIFT;
  IntegerConstant SCAN_BITS_NEEDED :=
    NUM_SCAN_LOAD_UNLOAD
    * LONGEST_SCAN_SHIFT
    * (NUM_SCANIN + 2*NUM_SCANOUT);

  ParamConstant NUM_PATTERN_VECTORS;
  IntegerConstant MAIN_MEM_NEEDED :=
    NUM_PATTERN_VECTORS * NUM_SIGNALS;

  Assert MAIN_MEM_NEEDED + SCAN_BITS_NEEDED <=
    NUM_PATTERN_MEMORY_BITS;
}
```

9. Resource statement

The resource statement is used to identify on a STIL statement or block basis, the specific resources within a target tester that are to be assigned for this statement or block. This information is optional and may be computed by the software load operation on the tester (see Figure 1). The content of the resource memory itself may be defined in a Pragma block in a format appropriate to the resource for the particular tester. See Annex G for an example of the usage of this facility.

9.1 Resource statement syntax

Resource (TESTER_IDENTIFIER)+ ; *// list of target testers* (1)
<< (RESOURCE_ID)+ >> (2)

(1) **Resource**: The Resource statement is used to define a list of tester names that are to have resource identifiers. The ordering of the tester_identifiers determines the ordering of the resource_identifier tags in the <<RESOURCE_ID>> tags that follow. Typically this statement will be at the beginning of a pattern block. It shall occur prior to the occurence of any <<RESOURCE_ID>> tag. The Resource statement and the <<RESOURCE_ID>> tags may be used in either a Pattern or a Timing block, and only one Resource statement shall be allowed within a Pattern or Timing block.

(2) <<RESOURCE_ID>>: The resource id tag is a string contained in double angle brackets that may optionally be placed prior to any block or statement keyword. The purpose of this <<RESOURCE_ID>> tag is twofold: (a) it is used to identify the tester resource that is to be assigned to support the tagged block or statement, and (b) as a reference to some other block (quite possibly a Pragma block) that defines the loading of the tester resource. The surrounding angle brackets are a required part of the syntax and are

expected to be ignored by parsers that are not involved with loading patterns to a tester. There may be multiple RESOURCE_ID names within the angle brackets, separated by a space; in which case, they are to be correlated with the tester names as listed in a Resource statement that shall be contained within the Timing or Pattern block. If no tag exists in a list of resource id tags, this shall be indicated by the use of the asterisk (*) character.

The resource id tag within the angle bracket is a format as required by the tester loader. The following are examples of allowed formats for resource id tags. The actual format that is used for any given tester loader is as defined by that application and is not defined in this standard:

a) Auser-defined name (see definition of user-defined name characteristics in 6.8 of STIL.0)
b) An integer
c) An asterisk (*) indicating a null resource id tag

10. TRC: TestResourceConstraints block

The TRC block is used to define the constraints for a given test system or the required resouce attributes for a given test program. The TRC block always exists within a Environment block. If a single set of constraints is being defined in a STIL file/stream, then neither the Environment nor the TRC block need be given a name. If the STIL file/stream contains TRC blocks for different classes of tester or different classes of user, then these would typically be in separately named Environment blocks. If the STIL file/stream contains TRC blocks for similar testers (e.g., different models or configurations), then these would typically be in separately named TRC blocks. When a TRC file contains multiple TRC blocks, the application processing that file may process all blocks present or specify which blocks to apply, depending on the application.When a TRC file contains multiple TRC blocks, the application processing that file may process all blocks present or specify which blocks to apply, depending on the application.

Either the Module block or the SignalAttributes block shall serve as the entry point into a set of TRC rules. If all resources are uniform, then a global (unnamed) SignalAttributes block should be used. If resources with different attributes are to be defined, then multiple Modules blocks are used. Refer to the statement syntax defintions in this clause for detailed definition of the rules.

A series of sub-blocks within a TRC block contain attributes of various different aspects of a system: SignalAttributes, PeriodAttributes, or PatternAttributes. Each of these blocks is referenced by either a Module block statement or a Signal block statement to construct the set of constraints that apply to a given instance.

There are facilities in the syntax to allow for sharing of common blocks. Refer to 10.3.

The following are general rules of interpretation that apply to all statements in a TRC block and all blocks that are contained within a TRC block:

a) In the syntax definitions, optional statements are identified with parenthesis ( ). When one optional statement is omitted in a STIL-TRC file, then this means that no checking of this constraint shall be done.
b) An integer value of “-1’” means that the parameter is unbounded and therefore need not be checked. This usage is effectively the same as omitting the statement altogether, however, it indicates a purposeful intent rather than a possible oversight.

10.1 TRC syntax

```
prefix = < E P T G M k m u n p f a > // used by Units statement
eng_unit = < A Cel F H Hz m Ohm s W V > // used by Units statement

Environment (ENV_NAME) {                                                          (1)
   ( TRC (TRC_NAME) {                                                             (2)
      ( STILExtensions ( < Design | DCLevels | TRC | CTL > )+ ; )                 (3)
      ( Usage < Constraints | PatternReport > ; )                                 (4)
      ( Category (CAT_NAME)+ ;)                                                   (5)
      ( Selector (SEL_NAME)+ ;)                                                   (6)
      ( Variables (VAR_NAME)+ ;)                                                  (7)
      ( MultipleDevices integer_expr;)                                            (8)
      ( MultipleSites integer_expr;)                                              (9)
      ( Module (MODULE_NAME) {                                                   (10)
         ( DCResourceAttributes DC_ATTR_NAME; )                                  (11)
         ( MaxNumberModules integer_expr; )                                      (12)
         ( PatternAttributes (PAT_ATTR_NAME)+ ;)                                 (13)
         ( PeriodAttributes PER_ATTR_NAME (<Synchronous | Asynchronous>) ;)      (14)
         ( SignalAttributes (SIG_ATTR_NAME) +; )                                 (15)
         ( WaveformAttributes WAV_ATTR_NAME ; )                                  (16)
         ( WaveformDescriptions WAV_DESC_NAME ; )                                (17)
      } )* // end Module
      ( Units (< (prefix) eng_unit | Integer | Real >)+; )                       (18)
      ( dc_resource_attributes_block )*                                          (19)
      ( name_checks_block )*                                                     (20)
      ( pattern_attributes_block )*                                              (21)
      ( period_attributes_block )*                                               (22)
      ( signal_attributes_block )*                                               (23)
      ( waveform_attributes_block )*                                             (24)
      ( waveform_descriptions_block )*                                           (25)
   })* // end TRC
} // end Environment
```

(1) **Environment**: The Environment block is a general-purpose construct defined in STIL.1 for the purpose of containing application environments for STIL data. In this case, the Environment block is used to contain tester resource constraint data that are used for transferring data from an EDA (pattern generation) environment into an ATE (pattern consumption environment). The environment block (and hence the environment block name) is used to (a) differentiate from all other environment blocks that may be used for other purposes and (b) to identify classes of TRC blocks (e.g., all like testers from a given manufacturer, or all testers on a test floor).

(2) **TRC**: This statement begins a block describing a set of constraints for a given tester or the set of test requirements for a given test program. The TRC_NAME is typically the name/model of the ATE system that is being defined. If the TRC block is unnamed, then all blocks within it are global (i.e., available within all the named TRC blocks within the containing Environment block). This facility allows for the sharing of common attributes by placing them in the "global TRC block" and referencing them from individual, named, TRC blocks. It is an error to have a block of the same name in both the global and a named TRC block. In the case where multiple blocks are allowed (e.g., Module or PatternAttributes), it is permissable to reference blocks in both the global and the named TRC as long as the names are unique.

(3) **STILExtensions**: This statement is used to indicate which of the STIL extensions are supported by the target tester or test program. If this statement is not present, then only STIL.0 is supported. The extensions available at the time of this writing are **Design**, **DCLevels**, and **CTL**.

(4) **Usage**: This statement is used to indicate the intent of the TRC block. The default if this statement is not present is Constraints. The allowed identifiers are as follows:

Constraints: This keyword specifies that the TRC block is used to define a set of target constraint attributes. These attributes will typically be for a particular model or instance of an ATE system. However, they may also be attributes that a user defines to constrain a test data flow, or constraints that a chip design application applies to a design flow.

PatternReport: Specify that the TRC block contains a report summary of the resources and requirements needed in support of execution of a pattern (or pattern burst).

(5) **Category** (CAT_NAME)+ : This statement is optional and is used to define the category names that are used within the TRC block. The typical usage for variables within TRC is for “fluid” constraints, i.e., constraints that are coupled one to another. See Annex B for information about fluid constraints.

(6) **Selector** (SEL_NAME)+ : This statement is optional and is used to choose a selector block that picks the min, typ, max values for spec-variables. If a selector block is not referenced, then all spec.-variables use the “Typical” values.

(7) **Variables** (VAR_NAME)+ : This statement is optional and is used to choose a variables block. If a named variables block is referenced, the names shall not conflict with the names defined in the global variables block (if one exists). The variables in the global block and the variables in the referenced block(s) are available in the current TRC block.

(8) **MultipleDevices**: This statement defines the number of like devices that may be tested simultaneously in a given tester. If this statement is omitted, then only single device testing is allowed.

(9) **MultipleSites**: This statement defines the number of un-like devices that may be tested simultaneously in a given tester. If this statement is omitted, then only single device type testing is allowed.

(10) **Module**: This statement begins a block that contains attribute information for one module or instrument of a tester. The allowed statements within a Module are defined immediately below. The name of a Module block is optional and is for documentation purpose only (a module is never referenced). There may be multiple unnamed and named Module blocks that together define a system. If an unnamed SignalAttributes exists, then it is an error to have any Module blocks.

(11) **DCResourceAttributes**: This statement references the block that defines the dc resource attributes for the module. There shall be only one DCResourceAttributes statement in the Module block. This statement in the Module block takes precedence over a global, unnamed block.

(12) **MaxNumberModules**: This statement specifies the maximum number of like instances of the module that are allowed. If this statement is omitted, the default value is 1.

(13) **PatternAttributes**: This statement references the block that defines the pattern attributes for the module. There shall be only one PatternAttributes reference statement in a Module block. There may be multiple names in the statement indicating that any one of the referenced blocks may be used; i.e., this is an “OR” function indicating that one OR the other of the named blocks may satisfy the rule checking. “ANDing” of rules sets that are contained in multiple blocks can be accomplished using the Inherit statement. The following code example indicates how the desired block is accomplished:

```
TRC {
  PatternAttributes PA1 { }
  PatternAttributes PA2 { }
}
TRC TRC2 {
  PatternAttributes PA3 { }
```

```
     PatternAttributes PA4 { }
     Module MOD1 {
       PatternAttributes PA1; // select PA1 from global TRC
     }
     Module MOD2 {
       PatternAttributes PA3 ; // select PA3 from the local TRC
     }
     Module MOD3 {
       PatternAttributes PA1 PA2 PA3 PA4 ; // select from all four
     }
   }
```

(14) **PeriodAttributes**: This statement references the block that defines the period attributes for the module. There shall be only one PeriodAttribute statement in a Module block. This statement in the Module block takes precedence over a global, unnamed block. This reference may be used either at the module level indicating that all signals share the same attributes or at the signals level if the attributes vary by signal. The following optional keywords are allowed:

a) **Synchronous**: Signals of this type shall be synchronous with all other signals blocks that specify the same period attributes block. This is the default value.
b) **Asynchronous**: Signals of this type shall be asynchronous with all other signals blocks; i.e., these signals shall use a private reference period generator.

(15) **SignalAttributes**: This statement references the block that defines the signal attributes for the module. There shall be only one SignalAttributes statement in a Module block. There may be multiple names in the statement indicating that any one of the referenced blocks may be used; i.e., this is an "OR" function indicating that one OR the other of the named blocks may satisfy the rule checking. "ANDing" of rules sets that are contained in multiple blocks can be accomplished using the Inherit statement.

(16) **WaveformAttributes**: This statement references the block that defines the waveform attributes for the module. There shall be only one WaveformAttributes statement in a Module block. It is allowed to have both a WaveformAttributes block and a WaveformDescriptions block; in which case, both sets of constraints must be met.

(17) **WaveformDescriptions**: This statement references the block that defines the waveform descriptions for the module. There shall be only one WaveformDescriptions statement in a Module block. It is allowed to have both a WaveformAttributes block and a WaveformDescriptions block; in which case, both sets of constraints must be met.

(18) **Units**: This statement defines allowed forms of unit representations that are allowed in the representation of literal numbers. If this statement is omitted, then all forms of numbers are allowed. If any eng_unit, Integer, or Real is specified, then all others are disallowed. If, for any given engr_unit, a prefix is specified, then all other prefixes are disallowed for this engr_unit. The following are some examples:

```
Units ns; // allow only ns; disallow all other forms
Units ns Integer; // allow ns and integer; disallow us, ms, etc; disallow exponential forms
Units s mV Integer Real; // allow ms, us, ns, etc.; also mV, integers, and exponential
```

(19) *dc_resource_attributes_block*: This named or unnamed block contains dc resource attribute statements. There shall be only one DCResourceAttributes statement associated with a Module block, either the global block or a referenced, named block. Refer to Clause 12 for details.

(20) *name_checks_block*: The NameChecks block is optional and contains statements describing naming rules for the objects within a STIL file/stream. Refer to Clause 17 for details. The name checks apply to all blocks in the current TRC block.

(21) *pattern_attributes_block*: This named or unnamed block contains pattern attribute statements. This block may be referenced by either (a) a PatternAttributes statement within a Module block, (b) a PatternAttributes statement with a SignalAttributes block, or (c) an Inherit statement within another PatternAttributes block. Refer to Clause 16 for details.

(22) *period_attributes_block*: This named or unnamed block contains attribute statements with respect to the period. There shall be only one PeriodAttributes statement associated with a Module block, either the global block or a referenced, named block. Refer to Clause 13 for details.

(23) *signal_attributes_block*: This named or unnamed block contains signal attribute statements. There shall be only one SignalAttributes statement associated with a Module block, either the global block or a referenced, named block. Refer to Clause 11 for details. If the SignalAttributes block is unnamed, then it serves as the top level reference to the system and all signal resources shall be the same. If the system contains resources with different attributes, then the Module statement shall be used.

(24) *waveform_attributes_block*: This named or unnamed block contains waveform attribute statements. There shall be only one WaveformAttributes block associated with a Module block, either the global block or a referenced, named block. Refer to Clause 14 for details. Note that whereas this block describes attributes of the waveforms, the WaveformDescriptions block describes actual waveforms that can be represented on an ATE system. Either method of describing waveforms may be used as appropriate to a given ATE system. It is allowed to have both a WaveformAttributes block and a WaveformDescriptions block; in which case, both sets of constraints must be met.

(25) *waveform_descriptions_block*: This named or unnamed block contains waveform description statements. There shall be only one WaveformDescriptions block associated with a Module block, either the global block or a referenced, named block.. Refer to Clause 15 for details. Note that whereas this block describes actual waveforms that can be represented on an ATE system, the WaveformAttributes block describes attributes or capabilities of the waveforms. Either method of describing waveforms may be used as appropriate to a given ATE system. It is allowed to have both a WaveformAttributes block and a WaveformDescriptions block; in which case, both sets of constraints must be met.

10.2 TRC example

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause 10.2 *}
}
Variables {
   ConfigConstant T1;
}
Environment ATE_CLASS {
 TRC ATE_NAME {
   STILExtensions Design DCLevels;
   Usage Constraints;
   Category MY_CAT;
   MultipleDevices 4;
   MultipleSites 16;

   NameChecks { Length 12; }
   SignalAttributes MY_SIGS { MaxSignals 256; }
   DCResourceAttributes MY_DC { DCLimits VIH (0 <= @@ <= 5V); }
   PatternAttributes MY_PATS { Max Locations 4096; }
   PeriodAttributes { TimeLimits (5ns <= @@ <= T1); }
```

```
  WaveformAttributes MY_WF_CHAR { Resolution 100ps; }

  Module M1 {
    MaxNumberModules 2;
    SignalAttributes MY_SIGS;
    DCResourceAttributes MY_DC;
    PatternAttributes MY_PATS;
    WaveformAttributes MY_WF_CHAR;
  } // end Module
} // end TRC
} // end Environment
```

10.3 TRC block sharing rules

Sharing of blocks is a technique for defining a set of rules that are then referenced and used in other blocks. The following are general rules that apply to block sharing:

a) There is NO implicit use of global (unnamed) blocks in this standard. All blocks are referenced by name.
b) Each block type has its own name space; i.e., all PatternAttributes block names are in one name space, all PeriodAttributes block names are one name space, and so on.
c) Blocks may reside in a global (unnamed) TRC block. The names of these blocks are included along with all block names in the current TRC block, and they are therefore available for reference by name.

11. TRC: SignalAttributes

The SignalAttributes block is used to define the number of signals and attributes of each type of signal. Multiple SignalAttributes blocks may exist, with each defining signals with different attributes. This block may contains references to other blocks that together with the explicit statements herein define the attributes of the signals.

11.1 TRC: SignalAttributes—syntax

signal_attributes_block =

SignalAttributes (SIG_ATTR_NAME) { (1)
(**Inherit** SIG_ATTR_NAME;)* (2)
(**AllowedScanPadWaveforms** WAV_DESC_NAME (< **ScanIn** | **ScanOut** >) ;) * (3)
(**DCResourceAttributes** DC_ATTR_NAME;) (4)
(**FanOut** *integer_expr*;) (5)
(**InOut** < **WithinCycle** | **OnCycleBoundary** | **Static** | **InOnly** | **OutOnly** >;) (6)
(**MaxScanMemory** *integer_expr*;) (7)
(**MaxCaptureMemory** *integer_expr* (<**FailOnly** | **Result**>)+ ;) (8)
(**MaxScanChainLength** *integer_expr* ;) (9)
(**MaxSignals** *integer_expr*;) (10)
(**PatternAttributes** (PAT_ATTR_NAME)+ ;) (11)
(**PeriodAttributes** PER_ATTR_NAME (<**Synchronous** | **Asynchronous**>) ;) (12)
(**UndrivenInOut** < **Yes** | **No** >;) (13)
(**WaveformAttributes** WAV_ATTR_NAME ;) (14)
(**WaveformDescriptions** WAV_DESC_NAME ;) (15)

} *// end SignalAttributes*

(1) **SignalAttributes**: The SignalAttributes block defines the attributes of a set of signals that have like functionality. There may be multiple SignalAttributes blocks if there are different types of tester channels (pins) on an ATE system. This block contains the reference to other block types that make up a system, i.e., PeriodAttributes and PatternAttributes. If the SignalAttributes block is unnamed, then it shall be referenced by a Module block to become effective. See Clause 10 for the definition of Module versus the SignalAttributes block for defining the TRC rules for a system.

(2) **Inherit**: This statement allows reference to another SignalAttributes block. All rules in the inherited block and the current block shall be satisfied; i.e., this is an "AND" of the statements in both blocks. See Clause 10 for examples of referencing global and named blocks in either the current or the other TRC blocks.

(3) **AllowedScanPadWaveforms**: This statement references a named WaveformDescriptions block that contains the allowed wave shapes to be used for scan padding. Optional attributes allow the specification of unique waveforms for ScanIn and ScanOut. The allowed waveforms are defined using the syntax as defined in Clause 15. A typical set of allowed pad wavefroms would look as follows:

```
WaveformDescriptions SCANPAD Explicit {
  Shape { Z; X; }
  Shape { Z; L; }
  Shape { X; }
}
```

(4) **DCResourceAttributes:** This statement references the block that defines the dc resource attributes for the containing module. There shall be only one DCResourceAttribute statement in the Module block. This statement in the SignalAttributes block takes precedence over one in the Module block.

(5) **FanOut** *integer_expr*: The FanOut statement is used to define the number of tester pin channels that may be driven by a given signal.

(6) **InOut**: The InOut statement is used to specify the in/out switching capabilities of a tester channel. The following are allowed:

a) **<u>WithinCycle</u>**: This keyword specifies that in/out switching may occur within a cycle (period). The exact time of the switch is specifed by the WaveformTable and the drive/compare event times within the waveform definition.

b) **OnCycleBoundary**: This keyword specifies that in/out switching may occur only on period boundaries. This effectively means that each waveform definition may have only drive or compare events, and that the first one must occur at T0 of the cycle.

c) **Static**: This keyword specifies that the in/out cannot be switched during the execution of a pattern. For a static signal, it is typically established at the beginning of the pattern exec. See also the pattern statement "Fixed" as defined in STIL.1.

d) **InOnly**: This keyword specifies that the signal is always an input to the DUT.

e) **OutOnly**: This keyword specifies that the signal is always an output from the DUT.

(7) **MaxScanMemory** *integer_expr*: This statement is used to specify that a scan state memory is available for this signal type and the number of scan states that can be defined for the signal. If this statement is omitted, it may still be possible to load scan states using vector memory.

(8) **MaxCaptureMemory** *integer_expr*: This statement is used to specify that there is a capture memory associated with the signal and the size of the capture memory. An additional keyword is used to specify how the memory is to be used. If both keywords are present, then the tester is capable of both.

a) **FailOnly**: The capture memory can be used to capture fail data.

b) **Result**: The capture memory can be used to capture result data.

(9) **MaxScanChainLength**: This statement is used to specify the maximum length of each scan chain.

(10) **MaxSignals** *integer_expr*: This statement is used to specify the maximum number of signals of a given type that are available on an ATE system.

(11) **PatternAttributes**: This statement is used to reference a named block that defines the pattern/vector attributes across a set of signals. See the definition of PatternAttributes in Clause 16 for more detail.

(12) **PeriodAttributes** PER_ATTR_NAME: This statement is used to reference a named block that defines the PeriodAttributes for this type of signal. An additional keyword indicates whether to share period attributes with other signal types. If both keywords are present, then the tester is capable of both.

a) **Synchronous**: Signals of this type shall be synchronous with all other signals blocks that specify the same period attributes block.

b) **Asynchronous**: Signals of this type shall be asynchronous with all other signals blocks; i.e., these signals shall use a private reference period generator.

(13) **UndrivenInOut**: This statement is used in pattern reporting to indicate that there are times within the pattern where signals are not driven by either the tester or the device. This situation can cause unreliable test conditions if not managed appropriately by the tester.

(14) **WaveformAttributes** WAV_ATTR_NAME: This statement is used to reference a waveform attributes block (within the current TRC block) that defines the waveform generation capabilities of this signal type. Note that the waveforms may also be defined as shapes (via the WaveformDescriptions block) as well as by their attributes. If both blocks exist, then the rules of both blocks shall be satisfied.

(15) **WaveformDescriptions** WAV_DESC_NAME: This statement is used to reference a waveform descriptions block (within the current TRC block) that defines the waveshapes that can be created by this signal type. Note that the waveforms may also be defined as attributes (via the WaveformAttributes block) as well as by their waveforms. If both blocks exist, then the rules of both blocks shall be satisfied.

11.2 TRC: SignalAttributes—examples

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause 11.2 *}
}
Environment { TRC {

   PeriodAttributes MY_PERS { TimeLimits 5ns <= @@ <= 1us; }
   WaveformAttributes MY_WF_CHARS { Resolution 100ps; }

   SignalAttributes {
     FanOut 4;
     InOut OnCycleBoundary;
     MaxScanMemory 512*1024*1024;
     MaxCaptureMemory 65_536;
     MaxScanChainLength 1302;
     MaxSignals 95;
     PeriodAttributes MY_PERS Synchronous;
     WaveformAttributes MY_WF_CHARS;
   } // end SignalAttributes
}} // end Environment-TRC
```

12. TRC: DCResourceAttributes

The DCResourceAttributes block is used to define the number and attributes of each type of dc resource for an ATE tester. If there is a single DCResourceAttributes block, then it may be unnamed and it applies to all Module and SignalAttributes blocks. There may be multiple DCResourceAttributes blocks; in which case, the name is required and is used as a reference in the Module or SignalAttributes block.

12.1 TRC: DCResourceAttributes—syntax

```
DCResourceAttributes DC_RESOURCE_NAME {                                              (1)
    ( Inherit DC_RESOURCE_NAME; )*                                                   (2)
    ( PerPinAttributes                                                               (3)
        (<Supply | PMU | Driver | Comparator | Load | Termination | Clamp>)+ {
            ( DifferentialConfiguration (<Driver | Comparator>)+;)                   (4)
            ( NumberTesterChannels integer_expr;)                                    (5)
            ( DCLimits                                                               (6)
                    (< VIH | VIL | VICM | VID | VIHD | VILD | VIHSlew | VILSlew
                    | VOH | VOL | VOCM | VOD | VOHD | VOLD
                    | IOH | IOL | LoadVRef | ClampHi | ClampLo
                    | ResistiveTermination | TermVRef
                    | VForce | IClamp | IForce | VClamp >)+ boolean_expr; )*
            ( NumberLevels (< VIH | VIL | VOH | VOL >)+ integer_expr; )*              (7)
    } )* // end PerPinAttributes
    ( NumberDCLevels integer_expr < ; | {                                            (8)
        SignalsPer integer_expr;
    } >)
    ( NumberDCSets integer_expr < ; | {                                              (9)
        SignalsPer integer_expr;
    } >)
    ( DCSequenceAttributes                                                           (10)
        (< Supply | PMU | Driver | Comparator | Load | Termination | Clamp >)+ {
            ( Shape {                                                                (11)
                ( (boolean_expr) (< Connect | Disconnect | Apply | Ramp >); )+
            } )+ // end Shape
    } ) // end DCSequenceAttributes
} // end DCResource
```

(1) **DCResourceAttributes:** The DCResourceAttributes block defines the attributes of the dc resources. There may be multiple DCResourceAttributes blocks if there are different types of dc resources on an ATE system. The DCResourceAttributes block is referenced by either a Module block, a SignalAttributes block, or an Inherit statement.

DC_RESOURCE_NAME: The name of the DCResourceAttributes block that is used as a reference.

(2) **Inherit**: This statement allows reference to another DCResourceAttributes block. All rules in the inherited block and the current block shall be satisfied; i.e., this is an “AND” of the statements in both blocks. See Clause 10 for examples of referencing global and named blocks in either the current or the other TRC blocks.

(3) **PerPinAttributes**: This statement defines a block of per-pin dc resources with like attributes. The dc resources (**Supply**, **PMU**, **Driver, Comparator**, **Load, Termination**, **Clamp**) are defined in STIL.2. If there are multiple PerPinAttributes block, then each must be for a different set of DCResources; there can be one for Supply and one for PMU, but the cannot be two for Supply.

(4) (**DifferentialConfiguration**: This statement specifies which differential dc resources are used for the signals in the referencing SignalAttributes block. The differential dc resources (**Driver, Comparator**) are defined in STIL.2.

(5) **NumberTesterChannels**: This statement specifies the number of tester channels required to implement the specified PerPinConfiguration or DifferentialConfiguration. The default is no limit.

(6) **DCLimits**: This statement defines the minimum and maximum dc levels for each specified dc resource. The boolean-expression defines the allowed limits of the identified levels. Delimiters are required around the boolean-expression. The dc resources are defined in STIL.2.

a) **VIH**: This statement defines the VIH voltage capability of the tester driver.
b) **VIL**: This statement defines the VIL voltage capability of the tester driver.
c) **VICM**: This statement defines the VICM voltage capability of the tester differential driver.
d) **VID**: This statement defines the VID voltage capability of the tester differential driver.
e) **VIHD**: This statement defines the VIHD voltage capability of the tester differential driver.
f) **VILD**: This statement defines the VILD voltage capability of the tester differential driver.
g) **VIHSlew**: This statement defines the VIH slew rate capability of the tester driver.
h) **VILSlew**: This statement defines the VIL slew rate capability of the tester driver.
i) **VOH**: This statement defines the VOH voltage capability of the tester comparator.
j) **VOL**: This statement defines the VOL voltage capability of the tester comparator.
k) **VOCM**: This statement defines the VOCM voltage capability of the tester differential comparator.
l) **VOD**: This statement defines the VOD voltage capability of the tester differential comparator.
m) **VOHD**: This statement defines the VOHD voltage capability of the tester differential comparator.
n) **VOLD**: This statement defines the VOLD voltage capability of the tester differential comparator.
o) **IOH**: This statement specifies the IOH current capability of the tester dynamic load.
p) **IOL**: This statement specifies the IOL current capability of the tester dynamic load.
q) **LoadVRef**: This statement specifies the LoadVRef voltage capability of the tester dynamic load.
r) **ClampHi**: This statement specifies the ClampHi voltage capability of the tester voltage clamp circuit.
s) **ClampLo**: This statement specifies the ClampLo voltage capability of the tester voltage clamp circuit.
t) **ResistiveTermination**: This statement specifies the resistive termination capability of the tester driver.
u) **TermVRef**: This statement specifies the TermVRef voltage capability of the tester driver termination.
v) **VForce**: This statement specifies the VForce voltage capability of the tester supply or PMU.
w) **IClamp**: This statement specifies the IClamp current capability of the tester supply or PMU.
x) **IForce**: This statement specifies the IForce current capability of the tester supply or PMU.
y) **VClamp**: This statement specifies the VClamp voltage capability of the tester supply or PMU.

(7) **NumberLevels**: This statement defines the maximum number of dc level values for each of the specified dc resources. The dc resources are defined in STIL.2. The default is 1.

a) **VIH**: This statement defines the number of different VIH levels provided by the tester, when used to switch dc levels within a cycle.
b) **VIL**: This statement defines the number of different VIL levels provided by the tester, when used to switch dc levels within a cycle.

c) **VOH**: This statement defines the number of different VOH levels provided by the tester, when used to switch dc levels within a cycle.
d) **VOL**: This statement defines the number of different VOL levels provided by the tester, when used to switch dc levels within a cycle.

(8) **NumberDCLevels** *integer_expr*: This statement specifies the number of DCLevels that may be referenced in a DCSets block (i.e., the number of levels that can be selected on-the-fly from a pattern). If not specified, the default is no limit. The following optional statement may be specified:

SignalsPer *integer_expr*: This statement specifies the incremental number of signals that are associated with each DCLevels block. If not specified, then the incremental number of signals allocated shall be one.

(9) **NumberDCSets** *integer_expr*: This statement specifies that the number of DCSets that may be referenced in a PatterExec block (i.e., the number of levels that can be selected on-the-fly from a pattern). If not specified, the default is no limit. The following optional statement may be specified:

SignalsPer *integer_expr*: This statement specifies the incremental number of signals that are associated with each DCSets block. If not specified, then the incremental number of signals allocated shall be one.

(10) **DCSequenceAttributes**: This statement defines the allowed sequencing of dc resources. One or multiple of the following identifiers indicate to which resources the block applies to: Supply, PMU, Driver, Comparator, Load, Termination, or Clamp. If there are multiple DCSequenceAttributes blocks, each shall be for a different set of resources, i.e., at most one block defining Supply sequencing and one block defining PMU sequencing. Each block shall contain one of more Shape blocks.

(11) **Shape**: This block defines the allowed action sequencing and allowed timing of these actions in controlling the associated dc resources.

a) *boolean_expr*: The boolean timing expression is optional. If not defined, then only the action sequence is defined. If the boolean expression is defined, then it is interpreted as an assert; i.e., the expression must evaluate to a true for the action being constrained to be valid. The following are the allowed tokens in a timing expression (see 6.13 of STIL.0 and 5.10 of STIL.1):
   i) absolute numbers that refer to the time from the beginning of the sequence
   ii) the @ label, which refers to the time of the prior action
   iii) @n, which refers to the time of the n'th action (where first action is numbered @1)
b) **Connect** | **Disconnect** | **Apply** | **Ramp**: This list of actions is allowed to construct a Shape.

12.2 TRC: DCResourceAttributes—example

```
Environment {
    TRC {
      DCResourceAttributes LOGIC {
        PerPinConfiguration Driver Comparator Load;
        NumberTesterChannels 1;
        NumberLevels VIH 2;
       NumberDCLevels 16 { SignalsPer 64; }
       DCLimits VIH (1V <= @@ <= 5V);
       DCLimits VIL (-2V <= @@ <= 3V);
       DCLimits VOL VOH (0V <= @@ <= 5V);
       DCLimits IOH (@@ > -10mA);
       DCLimits IOL (@@ < 20mA);
       DCLimits LoadVRef (@@ < = 4V;}
      }
    }
  }
```

13. TRC: PeriodAttributes

The PeriodAttributes block contains information about the timing control system of a tester. If there is only a single PeriodAttributes block, then it may be unnamed and it applies to all Module and SignalAttributes blocks. There may be multiple PeriodAttributes blocks; in which case, the name is required and is used as a reference in the Module or SignalAttributes block.

13.1 TRC: PeriodAttributes—syntax

```
period_attributes_block =
    PeriodAttributes (PER_ATTR_NAME) {                                        (1)
        ( Inherit PER_ATTR_NAME; )*                                           (2)
        (Accuracy time_expr;)                                                 (3)
        (MaxPeriods integer_expr;)                                            (4)
        (MaxPeriodGenerators integer_expr (< Dynamic | Static >) < ; | {      (5)
            ( SignalsPer integer_expr; )
        } >) // end MaxPeriodGenerators
        ( PeriodSelectMemory integer_expr < ; | {                             (6)
            ( SignalsPer integer_expr; )
        } >) // end PeriodSelectMemory
        ( TimeLimits boolean_expr; )                                          (7)
        (Resolution time_expr;)                                               (8)
    } // end PeriodAttributes
```

(1) **PeriodAttributes** PER_ATTR_NAME: The period attributes block contains statements defining the attributes of the period (or cycle) generator of an ATE system. The name (PER_ATTR_NAME) of this block is used by a SignalAttributes block to reference the capabilities that apply to the signals.

(2) **Inherit**: This statement allows reference to another PeriodAttributes block. All rules in the inherited block and the current block shall be satisfied; i.e., this is an "AND" of the statements in both blocks. See Clause 10 for examples of referencing global and named blocks in either the current or the other TRC blocks.

(3) **Accuracy** *time_expr*: Specify the accuracy of the ATE system in generating the period. The accuracy means that the actual period shall be within (period – time_expr < t > period + time_expr). (Note that there are multiple factors that must be considered in the overall timing: (a) period accuracy and resolution, (b) drive event accuracy and resolution, and (c) compare event accuracy and resolution. This statement specifies only the accuracy of the period.)

(4) **MaxPeriods** *integer_expr*: This statement specifies the maximum number of period values that may be specified. The periods are to be selected on a vector basis by means of the associated waveform table.

(5) **MaxPeriodGenerators** *integer_expr*: This statement specifies the number of independent periods that may be specified simultaneously. These independent period generators are to be assigned to blocks of signals that will then execute independently according to the separate period generation sequence of each one. This capability is typically used to support multiport devices with different timing domains. The following additional parameters shall be specified:

a) **Dynamic**: This keyword specifies that the signal assignment may be changed from one pattern exec to the next. This is the default attribute.
b) **Static**: This keyword specifies that the signal assignment to a period generator is fixed according to the architecture of the ATE system and cannot be changed.

The MaxPeriodGenerator may optionally be defined as a block and contain the following statement:

SignalsPer *integer_expr*: This statement specifies the incremental number of signals that are associated with a period generator. If not specified or if SignalsPer statement set to 1, then the incremental number of signals allocated shall be one.

(6) **PeriodSelectMemory** *integer_expr*: This statement specifies that the period selection is accomplished by means of an indirect selection memory. The integer value specifies the size of the indirect memory. The following optional statement may be specified:

SignalsPer *integer_expr*: This statement specifies the incremental number of signals that are associated with each period selection. If not specified or if SignalsPer statement set to 1, then the incremental number of signals allocated shall be one.

(7) **TimeLimits**: This statement specifies the limits of the period generator. The @@ symbol is used to represent the value to be programmed by the period generator, and a boolean expression is used to specify the min/max limits allowed.

(8) **Resolution** *time_expr*: This statement is used to specify the minimum increments that the period can be set to. Note that this is different from accuracy, which defines the relation between the value programmed and the resulting effect on an ATE system.

13.2 TRC: PeriodAttributes—examples

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause 13.2 *}
}
Environment {TRC {
   PeriodAttributes {
     Accuracy 200ps;
     MaxPeriods 16;
     MaxPeriodGenerators 2 Static {SignalsPer 64;}
     PeriodSelectMemory 16 { SignalsPer 32; }
     TimeLimits (2ns <= @@ <= 10us);
     Resolution 100ps;
   } // end PeriodAttributes
}} // end Environment-TRC
```

14. TRC: WaveformAttributes

The WaveformAttributes block contains information about the timing attributes of the signals. If there is only a single WaveformAttributes block, then it may be unnamed and it applies to all Module and SignalAttributes blocks. There may be multiple WaveformAttributes blocks; in which case, the name is required and is used as a reference in the Module or SignalAttributes block.

14.1 TRC: WaveformAttributes—syntax

```
waveform_attributes_block =
    WaveformAttributes (WAV_ATTR_NAME) {                                          (1)
        ( Inherit WAV_ATTR_NAME; )*                                               (2)
        ( Accuracy <Edge|EdgeToEdge> time_expr;)*                                 (3)
        ( CompareEvents                                                           (4)
            (<H|L|X|V|T|h|l|x|v|t|full_name>(/<H|L|X|V|T|h|l|x|v|t|full_name>)*)+
            (integer_expr (integer_expr)) ; )* // number events, number substitutes
        ( DriveEvents                                                             (5)
            (<U|D|Z|P|full_name>(/<U|D|Z|P|full_name>)*)+
            (integer_expr (integer_expr)) ; )* // number events, number substitutes

        // use the following syntax to define MaxShapes
        < ( FormatSelect < In | Out | InOut > {                                   (6)
            (< MaxShapes integer_expr (< Dynamic | Static >) < ; | {              (7)
                | MaxShapes integer_expr (< Dynamic | Static >) {
                    ( SignalsPer integer_expr; )
            } >)
        } )* // end FormatSelect with MaxShapes

        // use following syntax to define format attribute statements
        | ( FormatSelect < In | Out | InOut > {
            (< MaxTimeSets integer_expr (< Dynamic | Static >) < ; | {            (8)
                    ( SignalsPer integer_expr; )
                    ( PerTimingGenerator; )
                } >)
            ( MaxTimingGenerators integer_expr (< Dynamic | Static >) < ; | {     (9)
                    ( SignalsPer integer_expr; )
                } >)
            ( MaxData <Drive|Compare|DriveCompare> integer_expr (<Dynamic | Static>) < ; | {   (10)
                    ( SignalsPer integer_expr; )
                } >)
            ( MaxIO integer_expr (< Dynamic | Static >) < ; | {                   (11)
                    ( SignalsPer integer_expr; )
                } >)
            ( < MaxMask integer_expr (< Dynamic | Static >) < ; | {               (12)
                    ( SignalsPer integer_expr; )
                } >)
        })* // end FormatSelect with format attribute statements
        >
        ( MaxEdgeTime time_expr;)                                                 (13)
        ( MinCompareWindow time_expr;)                                            (14)
        ( MinCompareToDriveOn time_expr;)                                         (15)
        ( MinEdgeReTrigger time_expr (<Drive | Compare>)* ;)*                     (16)
        ( MinDriveOffTime time_expr;)                                             (17)
        ( MinDriveOffToCompare time_expr;)                                        (18)
        ( MinDriveOnTime time_expr;)                                              (19)
        ( MinDrivePulse time_expr;)                                               (20)
        ( SubWaveformIteration integer_expr;)                                     (21)
        ( SubWaveformDuration boolean_expr;)                                      (22)
        ( Resolution time_expr;)                                                  (23)
        ( TimeLimits boolean_expr; )                                              (24)
        ( WaveformSelectMemory integer_expr < ; | {                               (25)
            ( SignalsPer integer_expr; )
            ( SelectWithPeriod; )                                    (26)
        } >)
    } // end WaveformAttributes
```

(1) **WaveformAttributes** WAV_ATTR_NAME: This block describes key attributes of the waveforms. The WAV_ATTR_NAME defines a name for this block that is used within a SignalAttributes block to reference the appropriate waveform attributes block. Note: See the SignalAttributes block for the relationship between the WaveformAttributes block and the WaveformDescriptions block

(2) **Inherit**: This statement allows reference to another WaveformAttributes block. All rules in the inherited block and the current block shall be satisfied; i.e., this is an "AND" of the statements in both blocks. See Clause 10 for examples of referencing global and named blocks in either the current or the other TRC blocks.

(3) **Accuracy**: This statement is used to specify the accuracy of each edge (i.e., each event in a waveform).

a) **Edge**: This keyword specifies that the accuracy is relative to the beginning of the period.

b) **EdgeToEdge**: This keyword specifies that the accuracy is relative to any other timing event.

c) *time_expr*: This is the value to be assigned to the accuracy. The occurence of the event shall be within (edge_time - accuracy_time) < t > (edge_time + accuracy_time).

(4) **CompareEvents**: This statement defines all compare events and compare event sequences that shall be allowed within a waveform. If the list contains only uppercase event identifiers, then only the edge strobe is allowed. Likewise, if only lowercase, then only the window strobe is allowed. If it contains both types of events, then either edge or window is allowed. The single character event identifiers or the full name indentifiers may be used interchangeably. The single character event identifiers or the full name indentifiers may be used interchangeably. The events represent what may be used to make up a WFT on a tester. The tester WFT may contain either single events (e.g., H L X) or substitutable set of events (e.g., H/L).

a) <**H|L|X|V|T|h|l|x|v|t|***full_name*>: A list of the allowed single events shall be defined (i.e., H L X h l). Also allowed are the *full_name equivalents for the single character event names (e.g., CompareHigh = H).*

b) /<**H|L|X|V|T|h|l|x|v|t|***full_name*>: A list of allowed compound compare events shall be defined (i.e., H/L, H/L/T). Also allowed are the *full_name equivalents for the single character event names (e.g., CompareHigh = H).*

c) *integer_expr*: The first integer expression defines the maximum number of compare events that shall occur in any cycle. This parameter is optional with the default of 1. This is typically determined by the number of timing edge generators in the tester, and it may be a function of tester speed or degrees of multiplexing.

d) *integer_expr*: The second integer expression defines the number of compound compare events that shall exist in a given waveform. This parameter is optional with the default of 1. For example, if this integer is a "2," then two unique compound events requiring two WFCs are allowed. The waveform definition would look like AB{10ns L/H[0]; 15ns X; 20ns L/H[1]; 25ns X;}. Note the use of the square bracketed index number. If there is no index number, then the default is a single substitute (see STIL.0). Multiple instances of the same compound data still count as one usage.

(5) **DriveEvents**: This statement defines all drive events and drive event sequences that shall be allowed within a waveform. This parameter is optional with the default of 1. The single character event identifiers or the full name indentifiers may be used interchangeably. The events represent what may be used to make up a WFT on a tester. The tester WFT may contain either single events (e.g., U D Z) or substitutable set of events (e.g., U/D).

a) <**U|D|Z|P|***full_name*>: A list of allowed drive events shall be defined (e.g., U D Z). Also allowed are the *full_name equivalents for the single character event names (e.g., DriveUp = U).*

b) /<**U**|**D**|**Z**|**P**|*full_name*>: A list of allowed compound drive events shall be defined (e.g., D/U). Also allowed are the *full_name equivalents for the single character event names (e.g., DriveUp = U).*

c) *integer_expr*: The first integer expression defines the maximum number of drive events that shall occur in any cycle. This parameter is optional with the default of 1. This is typically determined by the number of timing edge generators in the tester, and it may be a function of tester speed or degrees of multiplexing.

d) *integer_expr*: The second integer expression defines the number of compound (i.e., substitute) drive events that shall exist in a given waveform. This parameter is optional with the default of 1. For example, if this integer is a "2," then the following waveform is allowed: pP{0ns P; 10ns D/U[0]; 15ns D; 20ns D/U[1]; 30ns Z;}, and two WFC definitions are expected from the pattern to control each event instance.

(6) **FormatSelect**: This statement begins a block that defines the format selection attributes of a waveform. See Annex I for the explanation of the terms and concepts as used herein. There are two forms of this block: one that defines only shapes and one that defines specific attributes. These two forms are mutually exclusive; i.e., they use only one form or the other. The total number of possible waveforms is the product of all the integers within this block. If select operations are combined (e.g., timing and I/O select is a common select operation on a given ATE system), then use the MaxShapes statement only. The In|Out keywords in this statement shall coordinate with the keyword on a referencing SignalCharacteristic block (i.e., if a SignalAttributes block is of type "In," then only FormatSelect of "In" shall be allowed).

a) **In**: This keyword specifies that this block contains waveforms with drive events only.

b) **Out**: This keyword specifies that this block contains waveforms with compare events only.

c) **InOut**: This keyword specifies that this block contains waveforms with both drive and compare events.

(7) **MaxShapes** *integer_expr*: This statement specifies the total number of shapes that shall be allowed. See Annex I for the explanation of the terms and concepts as used herein. If this statement is used, then it shall be the only one within this block. If this statement is omitted, then the other statement shall be used to specify the individual selections criteria.

a) **Dynamic**: Specify that shapes are selectable on a vector by vector basis.

b) **Static**: Specify that shapes are selectable only at the beginning of a pattern exec.

c) **SignalsPer** *integer_expr*: This statement specifies the incremental number of signals that are associated with each shape selection. If not specified or if SignalsPer statement is set to 1, then the incremental number of signals allocated shall be one.

(8) **MaxTimeSets** *integer_expr*: This statement specifies the number of time sets that shall be allowed. See Annex I for the explanation of the terms and concepts as used herein. This determines the number of timing values that may be assigned to a given waveform in order to create formats that are of the same shape but different timing.

a) **Dynamic**: Specify that time sets are selectable on a vector by vector basis.

b) **Static**: Specify that time sets are selectable only at the beginning of a pattern exec.

c) **SignalsPer** *integer_expr*: This statement specifies the incremental number of signals that are associated with each time set selection. If not specified or if SignalsPer statement is set to 1, then the incremental number of signals allocated shall be one.

d) **PerTimingGenerator**: Specify that time set selection is done on a per-TG basis. See the MaxTimingGenerators statement for further definition.

(9) **MaxTimingGenerators** *integer_expr*: This statement specifies the number of timing generators that shall be allowed. See Annex I for the explanation of the terms and concepts as used herein. A timing generator is a function that can generate multiple sets of timing events (i.e., time sets) for multiple signals.

a) **Dynamic**: Specify that TGs are selectable on a vector by vector basis.

b) **Static**: Specify that TGs are selectable only at the beginning of a pattern exec.

c) **SignalsPer** *integer_expr*: This statement specifies the incremental number of signals that are associated with each timing generator selection. If not specified or if SignalsPer statement is set to 1, then the incremental number of signals allocated shall be one.

(10) **MaxData**: This statement specifies the number of data values that are used to make up a waveform. See Annex I for the explanation of the terms and concepts as used herein. This is typically referred to as pattern memory data. For example, if the pattern memory has one bit per pin for both drive and compare, then this condition is defined by "DriveCompare 2;", whereas if the pattern memory has separate data bits for drive and compare, then it would be defined by "Drive 2; Compare 2;".

a) **Drive** *integer_expr*: Specify the number of data states available for drive.

b) **Compare** *integer_expr*: Specify the number of data states available for compare.

c) **DriveCompare** *integer_expr*: Specify the number of data states to be shared by drive and compare.

d) **Dynamic**: Specify that data states are selectable on a vector by vector basis.

e) **Static**: Specify that data states are selectable only at the beginning of a pattern exec.

f) **SignalsPer**: This statement specifies the incremental number of signals that are associated with each data selection. If not specified or if SignalsPer statement is set to 1, then the incremental number of signals allocated shall be one.

(11) **MaxIO** *integer_expr*: This statement specifies the number of input/output select values that are used to make up a waveform. See Annex I for the explanation of the terms and concepts as used herein.

a) **Dynamic**: Specify that input/output selection is on a vector by vector basis.

b) **Static**: Specify that input/output selection is only at the beginning of a pattern exec.

c) **SignalsPer**: This statement specifies the incremental number of signals that are associated with each I/O selection. If not specified or if SignalsPer statement is set to 1, then the incremental number of signals allocated shall be one.

(12) **MaxMask** *integer_expr*: This statement specifies the number of compare mask select values that are used to make up a waveform. See Annex I for the explanation of the terms and concepts as used herein.

a) **Dynamic**: Specify that compare mask selection is on a vector by vector basis.

b) **Static**: Specify that compare mask selection is only at the beginning of a pattern exec.

c) **SignalsPer**: This statement specifies the incremental number of signals that are associated with each mask selection. If not specified or if SignalsPer is statement set to 1, then the incremental number of signals allocated shall be one.

(13) **MaxEdgeTime** *time_expr*: Specify the maximum allowed time that an edge can be programmed from T0 (i.e., the beginning of the period).

(14) **MinCompareWindow** *time_expr*: Specify the minimum allowed strobe width (i.e., from L/H to X). Note that the next occurence may be in a following period.

(15) **MinCompareToDriveOn** *time_expr*: Specify the minimum allowed time from doing a compare strobe to turning the driver on (i.e., from L/H to P/U/D).

(16) **MinEdgeReTrigger** *time_expr*: Specify the minimum time that shall exist prior to the next occurence of a like event. Note that the next occurence may be in a following period. The following optional keywords are allowed:

a) **Drive**: Specify that the retrigger time applies to drive events.

b) **Compare**: Specify that the retrigger time applies to compare events.

c) default if Drive or Compare not specified is that retrigger applies to all events.

(17) **MinDriveOffTime** *time_expr*: Specify the minimum allowed drive off time (i.e., from Z to next U/D/P). Note that the next occurence may be in a following period.

(18) **MinDriveOffToCompare** *time_expr*: Specify the minimum allowed time from turning the driver off to doing a compare strobe (i.e., from Z to L/H/T).

(19) **MinDriveOnTime** *time_expr*: Specify the minimum allowed drive on time (i.e., from U/D, or P/Z). Note that the next occurence may be in a following period.

(20) **MinDrivePulse** *time_expr*: Specify the minimum allowed drive pulse width (i.e., from U/D, or D/U). Note that the next occurence may be in a following period.

(21) **SubWaveformIteration** *integer_expr*: Specify the number of times that the waveform may be repeated. Refer to 18.1 of STIL.0 for the definition of sub-waveforms. The *integer* is the maximum number of iterations allowed. The integer shall be a literal integer or a constant integer expression.

(22) **SubWaveformDuration** *boolean_expr*: Specify the limits of the allowed time duration of each iteration. Refer to 18.1 of STIL.0 for the definition of sub-waveforms. The time expression is an assert; i.e., the expression must evaluate to a true for the waveform being constrained to be valid. For more explanation and examples of edge time expressions, see the Shape block in Clause 15.

(23) **Resolution** *time_expr*: This statement defines the minimum increment for specifying a time value of a waveform event. Note that there is a separate statement for specifying the period resolution.

(24) **TimeLimits**: This statement specifies the limits of the waveform edge generators. The @@ symbol is used to represent the value to be programmed by the edge generator, and a boolean expression is used to specify the min/max limits allowed. Time edges may extend beyond the end of the period. A negative time value can be used to specify time that begins prior to T0 of the period.

(25) **WaveformSelectMemory** *integer_expr*: This statement defines the size of a indirect memory that has the function of accessing the specific waveforms. If this statement is not present, then no indirect memory is present.

SignalsPer *integer_expr*: This statement specifies the incremental number of signals that are associated with each waveform selection. If not specified or if SignalsPer statement set to 1, then the incremental number of signals allocated shall be one.

(26) **SelectWithPeriod**: This statement defines that the indirect memory specified by the WaveformSelectMemory statement is to be accessed in common with the indirect memory specified by the PeriodSelectMemory.

14.2 TRC: WaveformAttributes—examples

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause 14.2 *}
}
Environment {
 TRC {
   WaveformAttributes {
     Accuracy 1ns;
     CompareEvents X T L/H 4;
     DriveEvents Z P D/U 3;
     FormatSelect In {
       MaxShapes 8 Dynamic { SignalsPer 32; }
```

```
      } // end FormatSelect
      FormatSelect Out {
        MaxTimesets 4 Dynamic { SignalsPer 32; }
        MaxTimingGenerators 2 Static;
        MaxData Compare 2;
        MaxIO 2;
        MaxMask 2;
      } // end FormatSelect
      MaxEdgeTime 4us;
      MinCompareWindow 2ns;
      MinEdgeRetrigger 5ns Drive Compare;
      MinDriveOffTime 2ns;
      MinDriveOnTime 4ns;
      MinDrivePulse 1ns;
      SubwaveformInteration 1024;
      SubwaveformDuration 5ns <= @@ <= 100ns;
      Resolution 500ps;
      TimeLimits (0ns <= @@ <= 1us);
      WaveformSelectMemory 4096 { SelectWithPeriod; }
    } // end WaveformAttributes
  } // end TRC
} // end Environment
```

15. TRC: WaveformDescriptions

The WaveformDescriptions block contains definitions of waveform shapes applied to the signals. If there is only a single WaveformDescriptions block, then it may be unnamed and it applies to all Module and SignalAttributes blocks. There may be multiple WaveformDescriptions blocks; in which case, the name is required and is used as a reference in the Module or SignalAttributes block.

15.1 TRC - WaveformDescriptions—syntax

waveform_descriptions_block =

WaveformDescriptions (WAV_DESC_NAME) (< **Rule** | **Explicit** >) { (1)

(**Inherit** WAV_DESC_NAME;)* (2)

WFNAME { (3)

(**NumberData** *integer_expr* ;) (4)

(**NumberIO** *integer_expr* ;) (5)

(**NumberMask** *integer_expr* ;) (6)

(**NumberPeriods** *integer_expr*;) (7)

(**NumberShapes** *integer_expr* ;) (8)

(**NumberSignals** *integer_expr*;) (9)

(**NumberTimeSets** *integer_expr* ;) (10)

Shape { (11)

((TRC_LABEL:) (*boolean_expr*) < EVENT | EVENT_LIST> ;)*

} *// end Shape*

})* *// end WFNAME*

} *// end WaveformDescriptions*

(1) **WaveformDescriptions** WAV_DESC_NAME: This block contains descriptions of allowed waveform shapes and times. It also defines the waveform select resources consumed by each waveform. This block is referenced by a SystemAttributes block by means of the WAV_DESC_NAME identifier. Note: See the SystemAttributes block for the relationship between the WaveformAttributes block and the WaveformDescriptions block

Rule: This optional parameter means that the shape definitions are rules that must be met by the waveform definitions. All waveforms shall conform to all rules. "Rule" is the default if neither "Rule" nor "Explicit" is specified.

Explicit: This optional parameter means only defined waveforms are allowed. For a waveform to be allowed, it must conform exactly to one of the definitions in the Shape block; i.e., it shall contain all events and in the same order; it shall conform to the time expression assertions.

(2) **Inherit**: This statement allows reference to another WaveformDescriptions block. All rules in the inherited block and the current block shall be satisfied; i.e., this is an "AND" of the statements in both blocks. See Clause 10 for examples of referencing global and named blocks in either the current or the other TRC blocks.

(3) WFNAME: This statement begins a block that defines a waveform and the resources needed to create that waveform.

(4) **NumberData** *integer_expr*: Specify the number of data states used to create this waveform. Default = no check. Typically, a waveform has two data states (1/0) that are used to select U/D for a drive waveform or H/L for a compare waveform.

(5) **NumberIO** *integer_expr*: Specify the number of I/O states used to create this waveform. Default = no check. A bidirectional waveform typically has two I/O states: to select drive on/off.

(6) **NumberMask** *integer_expr*: Specify the number of mask states used to create this waveform. Default = no check. Typically an output waveform has two mask states: compare/don't compare

(7) **NumberPeriods** *integer_expr*: Specify number of periods needed to create this shape. Default = no check. This statement is for tester architectures that can create complex shapes that encompass multiple periods.

(8) **NumberShapes** *integer_expr*: Specify the number of shape selects used to create this waveform. Default = no check. This is an alternative way of specifying shape selection. If this statement is used, then it should combine NumberData, NumberIO, and NumberMask into this one attribute.

(9) **NumberSignals** *integer_expr*: Specify the number of tester pins needed to create this shape. Default = no check. This statement is to support special tester architecture where multiple pin channel resources can be tied together to create a complex waveform.

(10) **NumberTimeSets** *integer_expr*: Specify the number time sets selects used to create this waveform. Default = no check.

(11) **Shape**: This statement begins a block to specify the events that make up the shape and the timing limits of the waveform. Note that the syntax follows the same form as for a waveform as defined in Clause 18 of STIL.0. The difference between a STIL.0 waveform and this waveform is that the definition is to be used as a constraint for a valid waveform rather than for a waveform itelf. The following is the definition of the components that make up a shape:

a) EVENT_LABEL: The label is optional and, if present, allows for reference by other shapes in the current WaveformDesctiptions block to be in reference to this timing lable. The ending colon is required syntax.

b) *boolean_expr*: The boolean time expression is optional. If not defined, then only the shape of the waveform is defined (for this one event time). If the boolean time expression is defined, then it is interpreted as an assert; i.e., the expression must evaluate to a true for the waveform being constrained to be valid. The following are the allowed tokens in a timing expression (see 6.13 of STIL.0 and 5.10 of STIL.1):
    i) Absolute numbers that refer to the time from T0
    ii) The @ label, which refers to the time of the prior event
    iii) @n, which refers to the time of the n'th event (where first event is numbered @1)
    iv) An event-label within the current WaveformDescriptions block
    v) @Tm, which refers to the beginning of the m'th cycle of this waveform definition (where the @T0 is the beginning of the first cycle of this waveform definition)
    vi) @Tm.n, which refers to the n'th event in the m'th cycle of this waveform definition

c) < EVENT | EVENT_LIST>: The event of the event list defines the waveform shape attributes in a similar manner as the waveform shape definitions of a Timing block in STIL.0.

15.2 TRC: WaveformDescriptions—examples

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause 15.2 *}
}
Environment {
 TRC {
   WaveformDescriptions Explicit {
     WF1 {
       NumberData 2;
       NumberIO 2;
       NumberMask 2;
       NumberPeriods 1;
       NumberShapes 2;
       NumberSignals 1;
       NumberTimeSets 1;
       Shape {
         (@1 :== 0ns) P;
         (@2 >= @1+2ns) D/U;
         ((@3 >= @2+2ns) && (@3 < 200ns)) D;
       } // end Shape
     } // end WF1
     WF2 {
       Shape {
         (@1 >= 10ns) D/U;
         (@2 >= @+5ns) D;
         (@T1 >= 25ns);
         (@T1.1 >= @T1+10ns) D/U;
         (@T1.2 >= @+5ns) D;
       } // end Shape
     } // end WF2
   } // end WaveformDescriptions
 } // end TRC
} // end Environment
```

16. TRC: PatternAttributes

The PatternAttributes block contains information that applies to the vector generation process. If there is only a single PatternAttributes block, then it may be unnamed and it applies to all Module and SignalAttributes blocks. There may be multiple PatternAttributes blocks; in which case, the name is required and is used as a reference in the Module or SignalAttributes block.

Within this block, the term "vectors" and the term "locations" are used to refer to two different aspects of the pattern generation system. The term "vectors" refers to the V statements in a STIL file/stream, whereas the term "locations" refers to the pattern memory on an ATE system that stores the vector information.

16.1 TRC: PatternAttributes—syntax

```
instruction_enum =
    < Condition | GoTo| IddqTestPoint| Loop | Macro | MatchLoop | Call | Shift | BreakPoint >   (1)
pattern_attributes_block =
    PatternAttributes (PAT_ATTR_NAME) {                                                        (2)
      ( Inherit PAT_ATTR_NAME; )*                                                              (3)
      ( Base < Hex | Dec > INTEGER; )                                                          (4)
      ( InstructionAttributes ( instruction_enum )+ < ; | {                                    (5)
        ( MinAfter < Locations | Vectors > integer_expr; )                                     (6)
        ( MinBefore < Locations | Vectors > integer_expr; )                                    (7)
        ( WithParameters < Yes | No > )                                                        (8)
        ( LoopAttributes (< Loop | MatchLoop | Shift | BreakPoint >)* {                        (9)
          ( Infinite; )
          ( MaxIteration integer_expr; )
          ( MaxLength integer_expr; )
          ( MaxNest integer_expr; )
          ( MinIteration integer_expr; )
          ( MinLength integer_expr; )
          ( MinTimeAfterMatch time_expr; )
          ( MinVectorsAfterMatch integer_expr; )
          ( Modulus < Locations | Vectors > integer_expr; )
        } )* // end LoopAttributes
      } > )* // end InstructionAttributes
      ( MaxRunTime time_expr;)                                                                 (10)
      ( Max < Locations | Vectors > integer_expr;)                                             (11)
      ( Modulus < Locations | Vectors > integer_expr; )                                        (12)
      ( MultiBitData (< InWaveforms | InPatterns | No >)+ ;)                                   (13)
      ( NonCyclized < Yes | No >;)                                                             (14)
      ( NumberCaptureCycles (integer_expr ( integer_expr)) ;)                                  (15)
      ( NumberPatternUnits integer_expr ;)                                                     (16)
      ( NumberVectorsPerShift (integer_expr) ( integer_expr);)                                 (17)
      ( PatternVariables                                                                       (18)
        (< Integer | IntegerConstant | SignalVariable | WFCConstant | Spec | All | No >)* ;)
      ( VectorCompression integer (< PerVectorMemory | VectorMemoryPer >) ; )                  (19)
    } // end PatternAttributes
```

(1) *instruction_enum*: This list of enums is used on the InstrurctionAttributes statement, which is defined below.

(2) **PatternAttributes** (PAT_ATTR_NAME): This block specifies the attributes associated with running a pattern. This block may also be used to document the attributes of a given pattern. The perspective of the

descriptions herein is from the tester constraint perspective. Note that some statements have little or no significance as a tester hardware constraint and are so described.

(3) **Inherit**: This statement allows reference to another PatternAttributes block. All rules in the inherited block and the current block shall be satisfied; i.e., this is an "AND" of the statements in both blocks. See Clause 10 for examples of referencing global and named blocks in either the current or other TRC blocks.

(4) **Base** < **Hex** | **Dec** > INTEGER: Specify whether vector data and parameters in radix other than WFC are supported. By default, only WFC data are supported. This pattern data characteristic is not typically a tester hardware constraint, but it may be a constraint of the ATE software. The INTEGER defines the maximum number of WFCs that can can be mapped from hex or decimal.

(5) **InstructionAttributes**: This statement specifies the instructions that are allowed. The form of this statement may be either semicolon terminated; in which case, it indicates only the availability of the instruction. This statement may be a block statement; in which case, it specifies the capabilities, limitations, and attributes of the named instructions types (*instruction_enum*) within a pattern. A series of keywords follow that indicate which specific pattern instructions are defined in the block. There may be multiple blocks, with each pattern instruction being defined, at most, in one block. There are no defaults for pattern instructions. If a pattern instruction is not defined in one InstructionAttributes block, then it is not allowed.

(6) **MinAfter** < **Locations** | **Vectors** > *integer_expr*: This statement specifies the minimum number of vectors (or the minimum vector memory locations) that must exist after this instruction type with no intervening instructions. The actual number of STIL vectors (for each "location") is determined by the VectorCompression statement.

(7) **MinBefore** < **Locations** | **Vectors** > *integer_expr*: This statement specifies the minimum number of vectors (or the minimum vector memory locations) that must exist before this instruction type with no intervening instructions. The actual number of STIL vectors (for each "location") is determined by the VectorCompression statement.

(8) **WithParameters** < **<u>Yes</u>** | **No** >: This statement is allowed for Macro and Call instructions and specifies whether parameters are supported. This pattern data characteristic is not typically a tester hardware constraint, but it may be a constraint of the ATE software. By default no checking or limitations are applied in the usage of macros or calls; hence, parameters are allowed. Note that whereas Macros are typically supported by expanding the pattern statements in-line, procedures are typically supported by transferring control to a preloaded pattern sequence. Also note that it is possible to expand procedures to in-line pattern statements; however, the rules for procedure calls shall be maintained.

(9) **LoopAttributes**: This statement is to be used only within an InstructionAttributes block describing one or more of the looping statements: **Loop**, **MatchLoop**, **Shift**, **BreakPoint**. If this statement is not present, for one looping statement, then the defaults as defined in the following LoopAttributes statements apply. The following statements are allowed in a LoopAttributes block:

a) **Infinite**: The number of interations in the loop may be infinite. An infinite loop overrides any value specified in a MaxIteration statement.
b) **MaxIteration** *integer_expr*: The statement specifies the maximum number of times the loop may be executed. The default value is no limit.
c) **MaxLength** *integer_expr*: This statement specifies the maximum number of Vector statements that may occur within the loop. The default value is no limit.
d) **MaxNest** *integer_expr*: This statement specifies the maximum number of loops that may be contained within a loop; i.e., "MaxNest 8;" means that there may be an outer loop and up to seven more loops embedded within. This statement also serves to specify when to expand vectors. If "MaxNest 0;" is specified, then all loops must be converted into in-line instruction sequences. The default value is no limit.

e) **MinIteration** *integer_expr*: This instruction specifies the minimum number of iterations allowed. The default value is one.

f) **MinLength** *integer_expr*: This statement specifies the minimum number of Vector statements that may occur within the loop. The default value is one.

g) **MinTimeAfterMatch** *time_expr*: This statement specifies the minimum time after a match before the pattern can continue. This time is defined in a BreakPoint block or statement within the MatchLoop. Due to pipelining, an ATE system typically cannot respond with valid compare results immediately after a match is found. This statement specifies the minimum amount of time that is required before a compare can be done. The default value is zero. See also MinVectorsAfterMatch.

h) **MinVectorsAfterMatch** *integer_expr*: This statement specifies the minimum number of vectors after a match before the pattern can continue. These vectors are defined in a BreakPoint block or statement within the MatchLoop. The default value is zero. See also MinTimeAfterMatch.

i) **Modulus** < **Locations** | **Vectors** > *integer_expr*: The statement specifies the incremental number of vectors (or locations) that shall be defined within the loop; i.e., “Modulus Vectors 7;” means that only loops of size 7 vectors, 14 vectors, and so on are allowed. The default is one.

(10) **MaxRunTime** *time_expr*: This statement specifies the maximum run time for a pattern execution. By default, there is no limit to the run time.

(11) **Max** < **Locations** | **Vectors** > *integer_expr*: This statement specifies the maximum number of vectors (or the maximum vector memory locations) that are allowed in this pattern type. The actual number of STIL vectors is determined by the VectorCompression statement. The default is to provide no limit to the number of vectors.

(12) **Modulus** < **Locations** | **Vectors** > *integer_expr*: The statement specifies the incremental number of vectors (or locations) that shall be defined within the pattern type; i.e., “Modulus Vectors 7;” means that only patterns of size 7 vectors, 14 vectors, and so on are allowed. The default is one.

(13) **MultiBitData** (< **InWaveforms** | **InPatterns** | **<u>No</u>** >: Ths statement specifies whether patterns with multibit vectors are supported. By default, multibit data are not allowed.

a) **InWaveforms**: Multiple bits used to select events in waveforms.

b) **InPatterns**: Multiple bits in pattern parameters to Macros and Procs.

(14) **NonCyclized** < **Yes** | **<u>No</u>** >: This statement specifies whether patterns with non-cyclized data are supported. By default, only cyclized patters are supported.

(15) **NumberCaptureCycles** (*integer_expr* (*integer_expr*)): This statement specifies the number of vectors in ATPG generated capture cycles. This pattern data characteristic is not typically a tester hardware constraint, but it is used for documenting a pattern. There is no default.

a) *no integers*: All capture sequences have the same number of cycles.

b) *first integer*: Minumum number of capture cycles.

c) *second integer*: Maximum number of capture cycles.

(16) **NumberPatternUnits** *integer_expr*: This statement specifies the number of ATPG generated pattern sequences (i.e., scan patterns). This pattern data characteristic is not typically a tester hardware constraint, but it is used for documenting a pattern. There is no default.

(17) **NumberVectorsPerShift** (*integer_expr*) (*integer_expr*): This statement specifies the number of vectors in ATPG scan shift sequences. The number represented is the number of DUT vectors that contain a parameter substitute on a scan-in or scan-out signal (either a #, %, sig-var, or wfc-string) after the Shift block is unrolled. This pattern data characteristic is not typically a tester hardware constraint, but it is used for documenting a pattern. There is no default.

a) *no integers*: All shift sequences have the same number of cycles.

b) *first integer*: Minumum number of shift cycles.

c) *second integer*: Maximum number of shift cycles.

(18) **PatternVariables**: This statement specifies whether variables are supported and which types of variables. Each of the various variable types can be individually specified or “All” allows for any type. By default, all are allowed. One or more of the following identifiers are allowed:

a) **Integer**: Allow integer variables to be used in pattern statements.

b) **IntegerConstant**: Allow integer constants to be used in pattern statements.

c) **SignalVariable**: Allow signal variables to be used in pattern statements.

d) **WFCConstant**: Allow WFC-constants to be used in pattern statements.

e) **Spec**: Allow spec-variables to be used in pattern statements.

f) **All**: Allow all of the above to be used in pattern statements (this is the default).

g) **No**: Do not allow any of the above to be used in pattern statements.

(19) **VectorCompression**: This statement specifies the number of vector memory locations that are to consumed by each vector of the pattern. If the attribute, PerVectorMemory, is specified, then each vector memory location generates the number of device vectors (i.e., STIL vectors) as specified by the integer attribute. If the attribute, VectorMemoryPer, is specified, then each device vector (i.e., STIL vector) requires the number of vector memory locations as specified by the integer attribute. The default is one STIL vector per vector memory location.

16.2 TRC: PatternAttributes—examples

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause 16.2 *}
}

Environment ATE1 {
 TRC {
   PatternAttributes {
     Base Hex 64;
     NonCyclized No;
     InstructionAttributes Breakpoint IddqTestPoint;
     InstructionAttributes Condition Loop Call Shift {
       MinVectorsBefore 2;
       MinVectorsAfter 2;
     } // end InstructionAttributes
     InstructionAttributes MatchLoop{
       MinVectorsBefore 2;
       MinVectorsAfter 35;
       LoopAttributes MatchLoop {
         MaxIteration 4096;
         MaxLength 65_000;
         MaxNest 3;
         MinIteration 2;
         MinLength 2;
       } // end LoopAttributes
     } // end InstructionAttributes
     Macro WithParameters;
```

```
        MaxRunTime 15s;
        Max Locations 5_000_000;
        MultiBitData InWaveforms;
        PatternVariables SignalVariable;
        ProcedureCalls Yes;
    } // end PatternAttributes
  } // end TRC
} // end Environment
```

17. TRC: NameChecks block

STIL defines a specific set of rules for naming objects and identifies namespaces that contain each type of object. Different environments and applications of STIL data often define different naming constructs from the STIL environment.

When STIL names are passed to contexts outside of STIL, additional restrictions may be imposed on these names. The NameChecks block is used to validate STIL names against a limited set of additional naming restrictions. These limited checks support checking names against an environment that supports reduced naming flexibility. It may be necessary in some circumstances to define restrictions using these rules that are a subset of the complete external context in order to simplify the checks to a naming environment that overlaps both the basic STIL constructs and the external environment. Obviously a name in an external context cannot be defined that is more flexible than the incoming STIL capabilities.

NameChecks operations come in three forms: character-content, length, and scope. They are specified against the names defined either inside a STIL block (for instance, the contents of the STIL Signals block) or against the blocknames of a STIL block (for instance, across the names of all WaveformTable blocks). Character-content checks validate that the set of characters contained in a name are appropriate; length checks confirm restrictions on the length of a name; and scope constraints check that this name is uniquely defined across the specified set of STIL blocks.

NameChecks shall be performed under an executable STIL context, that is, after STIL name resolution processes have identified a set of names that are actually used under a PatternExec context.

17.1 NameChecks block—syntax

name_checks_block =
  NameChecks { (1)
    (**Contents** (STIL_BLOCK_NAME)+ **;**) (2)
    (**Block** STIL_BLOCK_NAME **;**)* (3)
    (**CharacterContent** *regular_expr* **;**) (4)
    (**Length** integer_expr ;) (5)
    (**Scope** (STIL_BLOCK_NAME)+ **;**) (6)
  }

(1) **NameChecks** NAM_CHECKS_NAME: An optional STIL block inside TesterResourcesConstraints to identify a set of STIL naming restrictions for this TesterResourcesConstraints block. Name checks shall be unique across all NameChecks blocks in one TRC block.

(2) **Contents** STIL_BLOCK_NAME: Defines a set of STIL names, defined by user-defined keyword statements, in this STIL_BLOCK_NAME, to be checked against the CharacterContent, Length, and Scope requirements identified in this block. STIL_BLOCK_NAME is any STIL block that contains user-defined name keywords (such as Signals). At least one Contents statement or Block statement shall be present in a

NameChecks block. If the keyword **AllNames** is specified for STIL_BLOCK_NAME, then All STIL namespaces will be checked against these constraints.

(3) **Block** STIL_BLOCK_NAME: Defines a set of STIL names, defined by the set of all blocknames defined under STIL_BLOCK_NAME, to be checked against the CharacterContent, Length, and Scope requirements identified in this block. STIL_BLOCK_NAME is any STIL block. At least one Contents statement or Block statement shall be present in a NameChecks block

(4) **CharacterContent** *regular_expr*: Defines a regular expression construct to identify any character restrictions on a name. See annex K for details about regular expressions.

(5) **Length** integer_expr: Defines an absolute limit on the number of characters a name may contain.

(6) **Scope** (STIL_BLOCK_NAME)+: Defines one or more STIL blocks to check for a unique name. This statement is only required when the TesterResourcesConstraints namespace context is different than the STIL environment, typically containing multiple namespaces. If the keyword **AllNames** is specified for STIL_BLOCK_NAME, then each name checked here will be checked against all the names contained in all STIL namespaces. Be aware that STIL namespaces that are defined as unique under another name (for instance, ScanChain names inside a ScanStructures block) will not maintain that restriction if that block is referenced in a Scope statement.

17.2 NameChecks—examples

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause 17.2 *}
}
Environment {

TRC WGL {
 NameChecks {
   // partial WGL name checks for Signals, Groups
   // in WGL, Signals and SignalGroups share the same namespace
   Contents Signals;
   Contents SignalGroups;
   CharacterContent (\".*\"|[A-Za-z][A-Za-z0-9_\-]*);
     // Names are either enclosed in double-quotes,
     // or start with an alphabetic character, followed by
     // any number of alphanumeric characters, underscore, or dash.
     // Does not trap use of WGL reserved words
   Scope Signals SignalGroups;
 } // end NameChecks

 NameChecks {
   // partial WGL name checks for ScanChains
   Block ScanChain;
   CharacterContent (\".*\"|[A-Za-z][A-Za-z0-9_\-]*);
     // Does not trap use of WGL reserved words
   Scope ScanChains;
     // Note STIL ScanChain names are scoped under ScanStructures.
     // This statement assures unique names for all chains across
     // all referenced ScanStructure blocks.
 } // end NameChecks
} // end TRC

// Namechecks for a tester that places all names into one big, untyped, pool.
TRC FLAT_TESTER {
 NameChecks FOR_ALL_NAMES {
            // This context supports only one big namespace
   Contents AllNames;
   Length 22;
   Scope AllNames;
 }
} // end TRC

} // end Environment
```

Annex A

(informative)

Glossary

The following glossary contains definitions of terms found in this document. This annex may contain definitions of terms for which syntax is defined in other STIL documents. The annex also contains definitions that are commonly used in reference to ATE systems.

accuracy: Usually used in reference to a resource of a tester, such as a timing or voltage value; refers to the maximum difference between the value specified and the value actually produced on a tester.

bidirectional signal: A signal that can both drive data into a device pin and compare data out of a device pin; this capability usually also involves controlling the strength of the drive signal such that it can be in a high drive current mode for input and high impedance (low current) mode for output; on most testers, signals are normally bidirectional. *See also:* **split I/O**.

boolean_expr : An expression the evaluates to a true/false as defined in STIL.1.

break point: A point in a pattern where the application of data (i.e., vectors) can be interrupted; typically used for reloading patterns or for synchronizing with other patterns/events.

capture/capture memory: The process of capturing test result information on a tester channel; capture memory is used for capturing multiple results at functional test rates; capture may consist of fail data only or may be a capture of all strobe activity

central timing architecture: Atester architecture where the timing generation logic is common across multiple tester channels and is connected/selected by means of some kind of multiplexing scheme; typically found on older test systems since the trend is toward per-pin timing systems. *See also:* **distributed resource tester**.

channel: The electronics in a test system used to create drive and compare data to each pin of the tester and, hence, to a signal of the DUT; typically consists of data generation logic, timing generation circuits, pin electronic circuits; may also contain logic for scan data, algorithmic generation logic, and other.

conditional statement: The ability for a tester to conditionally execute statements in a pattern based on decisions that are defined in the pattern; conditions are specified by the If/Else statements as defined in STIL.1.

core: A submodule of a design that can be tested independantly of the whole design; details of the embedded core definition are defined in STIL.6; the TRC statement CoreUsageReady is used to indicate that the tester supports core testing or to indicate that patterns are constructed for core test.

constraint: *See*: **tester resource constraint**.

cycle/cyclized/noncyclized: A cycle is one period of a tester; a cycle also corresponds to one vector of a pattern; data may be defined in either cyclized form (i.e., periods, waveformtables, or waveformchars) or noncyclized (i.e., sequences of events on a per-signal basis).

data channel: Data channel refers to the data generation part of a tester channel. *See*: **channel**.

delta change vector data: The definition of vector data by defining only the signals that change from the prior vector; this is defined in STIL.0.

distributed resource/distributed resource tester: A tester architecture where the resources exist on a channel basis; typically refers to the timing and logic control functions of the tester. *See also*: **central timing architecture**.

device under test (DUT): Refers to the physical device to which a test system connects for the purpose of validation, characterization, or failure diagnosis.

edge/drive edge/compare edge: The points in time that are created by a test system to determine when drive events or compare events can occur; whereas events refer to the STIL defined waveform, edges refer to the tester actions that implement these STIL events. *See also*: **events**.

edge time: The timing placement of a drive or compare edge relative to the beginning of the period.

environment: Refers to things that are related to the application or usage of STIL data rather than the STIL pattern data itself; STIL.1 defines a new block called Environment for containing this information; TRC blocks are contained in environment blocks since they are not necessary for interpreting the STIL data, but only for applying it to a specific test system.

Event Sequence Store Memory: A term used on some testers to refer to a memory that holds sequences of a waveform event that generates a waveform; usually contains both shape and time values.

events/compare events/drive events/compound events: The components that make up a cyclized waveform; drive events define the operation of input to a DUT; compare events define the operation of monitoring the output from a DUT; compound events are multiple events that can occur at a given edge time based upon control and selection mechanisms. *See also*: **edge.**

flow: The user flow in which the STIL data are used; the path of transporting/translating data from one environment to another; refer to Clause 1 for definition of the flows that use TRC data.

fluid: The use of variables and expressions to define relationships between various resources of a tester.

functional tester: A tester architecture that is designed to test a DUT in a manner that mimics the application of the mission mode of the DUT. *See also*: **structural tester**.

iddq/iddq test points: A point in a pattern where an iddq measurement is to be performed; iddq refers to the quiescent current on a device power pin.

instruction/instruction attributes: Refers to a statement used in the high-speed functional pattern generation system of a tester to control the sequence of vectors; typically, condition statements are across all channels of a tester or all channels of a segment; example of instructions are Loop, Call, GoTo, etc.; instruction attributes is a block of TRC statements that defines the allowed instructions.

integer/integer_expr: An integer or an integer expression as defined in STIL.1.

label: An identifier in a pattern or waveform that is used as a reference by another statement. *See also*: **tag**.

local memory: A tester term that is used to refer to a pattern data memory that can be accessed according to the vector-period timing; this term historically differentiates from earlier testers that relied on CPU memory to produce tester data.

loop/loop attributes: Refers to a series of vectors that are to be repeated; loop attributes are the set of capabilities/limitations that exist with regard to looping.

macro: Refers to a set of pattern statements that are defined in a MacroDefs block and are invoked by the **STIL.0** statement “Macro”; marco statements are to be interpreted as if they were in-line statements.

non return to zero (NRZ)/delayed non return to zero (DNRZ): NRZ is a tester term for a waveform that does not go back to a low state within the a tester cycle; typically NRZ data are applied at the beginning of the period. ADNRZ is one that applies data after the beginning of the period; in STIL, this waveform is represented as {U/D;}. *See also*: **RZ**, **RO**, and **SBC**.

map/mapping: The action of translating STIL data to the resources of a tester; theTRC statements are defined to assist in this process.

mask: The capability of a tester to ignore the state of an output signal of a DUT.

memory/vector memory/scan memory/subroutine memory: Memory resources in a tester for containing pattern data; memory is often assigned for specific usage (i.e., storage of pattern vectors, storage of scan patterns, storage of subroutine, or procedure vectors).

multibit data: The definition of multiple data states in a pattern; can be assigned to a WFC; can be passed to either a procedure, macro, or waveform. *See also*: **events/compare events/drive events/compound events**.

on-the-fly: The ability of a tester to process an action at pattern speeds; contrast this definition with “static.”

pattern attributes: The set of attributes that defines how a tester processes/creates pattern data.

pattern report: A set of TRC statements that define the attributes of a specific STIL pattern, pattern burst, or set of patterns.

pattern unit: A term used for a sequence of vectors that make up a test as created by an automatic test pattern generation (ATPG) tool; typically used in the context of scan and composed of a scan-in, some activity in DUT pins, followed by a scan-out.

pattern variable: A variable, as defined in STIL.1, that is used to control the flow or data within a pattern.

period: The time from the start of a tester cycle (T0) to the beginning of the next tester cycle (next T0); the time that it takes to execute one pattern vector.

period attributes: The attributes of a tester that are used to implement the period processing function.

period generator: The electronic circuitry in a tester that generates a period; a period generator may be associated with a single tester channel, or it may be shared across multiple tester channels.

period select memory: An indirect memory that is used to select a period generator.

per-pin architecture: The tester architecture where resources are allocated on an individual channel basis; this typically applies to timing resources but can also apply to pattern formatting resources. *See also*: **central timing** and **shared resource architecture**.

pin: The point of contact on either a DUT or a tester; the electronics behind the tester pin is typically referred to as the tester channel; the abstract waveform on a DUT and as defined in STIL is referred to as a signal.

postprocess: The processing of STIL data after they are initially created by some generation process; the act of tester targetting is a postprocess.

pragma: A set of instructions for loading tester resources that is defined in a non-STIL or tester-specific form. Refer to Clause 19 of STIL.1 for the definition of pragma.

procedure call: Refers to a set of pattern statemements that are defined in a Procedures block and are invoked by the STIL.0 statement “Call”; a procedure differs from a macro in that it is defined stand-alone, no information is carried over from the prior sequence of vectors, and no information is carried back to the pattern vectors to follow.

range: Refers to the common technique used in test systems of using ranges to cover the needed spread and resolution for resources such as time, voltage, and current.

real / real_expr: A real number or a real expression as defined in STIL.1. A real number may be expressed in exponent form (i.e., 23.5E-9) or in engineering unit form (i.e., 23.ns). An integer may be used anywere a real number is called for.

regular expression/*regular_expr*: A software technique for the manipulation of text strings. See Annex K for details.

resolution: Usually used in reference to a tester resource; refers to the incremental value to which a resource can be programmed. *See also*: **range**.

resource: Refers to a capability of a test system; resources typically have a size attribute associated (i.e., number of pin channels or number of vectors).

return to one (RO/RTO): A tester term for a waveform that returns to a high state within the period; in STIL, this waveform is represented as {U/D; U;}. *See also*: **RZ**, **RO**, and **SBC**.

return to zero (RZ/RTZ): A tester term for a waveform that returns to a low state within the period; in STIL, this waveform is represented as {U/D; D;}; *See also*: **RZ**, **RO**, and **SBC**.

run time: The time it takes to execute a pattern or pattern burst.

surround by complement (SBC): A tester term for a waveform that has the complementary state before and after the desired data state; in STIL, this waveform is represented as {D/U; U/D; D/U;}; *See also*: N**RZ**, **RO**, and **RZ**.

scan/scan chain/scan memory: Refers to the structural design technique of connecting sequential elements together to allow for load/unload of data; also used to refer to special resources on a tester for the purpose of storing/loading/unloading scan chain data. *See also*: **shift**.

scope: Term used in STIL to refer to the area of affectivity over the STIL file/stream; i.e., global procedures and global variables blocks are available for use throughout all blocks of the file, whereas named procedures and named variables blocks are available only when specifically referenced.

segment: A partition of a tester that has common resource attributes.

select memory: Memory on a test system that is used to address a resource such as timing, formatting, or level select; this is a technique for optimizing the select capabilities by using an indirect memory to select a limited resource.

sequence control memory: Term for a tester resource that controls the flow of execution of pattern statements; a necessary resource to support loop and conditional statements in a pattern.

shape: The sequence of events that make up a waveform; typically the shape is defined independent of the timing. *See also*: **NRZ**, **RZ**, **RO**, and **SBC**.

shared resource architecture: The tester architecture where resources are shared or multiplexed to multiple channels; this typically applies to timing resources but can also apply to pattern formatting resources. *See also*: **central timing** and **per-pin architecture**.

shift: A STIL pattern statement that is used to load/unload scan data; also refers to the capability on a test system that implements the load/unload of scan data.

signal attributes: The set of attributes that define the capabilities of a tester channel.

signals per: A term used in this standard to identify the incremental number of signals that are associated with a given tester resource.

split I/O: A tester term for the ability to connect the input part of a pin channel to one tester pin and the output part of the pin channel to a second (usually adjacent) tester pin.

static: Attributes of a tester that are defined prior to the beginning of pattern execution. *Con*: **on-the-fly**.

STIL file/stream: A term used to refer to either a single STIL file or a set of STIL files, called a stream, that are connected by means of "Include <file-name> IfNeed;" statements. Although the term "stream" was not used in STIL.0, all statements necessary to create a stream are defined therein.

strobe/edge strobe/window strobe: Terms used to define the compare capabilities of a tester; strobe refers to the ability to do a compare on an output signal of a DUT; edge strobe refers to the ability to do a point compare on an output signal of a DUT (requires only one compare event: L/H/X/V/T); window strobe refers to the ability to do a compare over a time interval on a DUT output signal (requires two events: l/h/t/v followed by x).

structural tester: A type of tester that is designed with resources to test the elements that make up the design (i.e., the structural elements of a DUT); typical resources are scan memory and shift hardware. *See also*: **functional tester**.

synchronous: Refers to the execution of two or more patterns such that they start at the same time.

tag: An identifier in a pattern or waveform that is used for the specific purpose of tester targeting. *See also*: **label**.

target tester: Refers to the tester or tester family for which a set of rules (i.e., a TRC block) is created.

tester resource: *See*: **resource**.

tester resource constraint (TRC): Refers to a set of rules that must be adhered to in order for a STIL file/ stream to be suitable for load/execution. These rules may be applied at various stages in the generation of a DUT and its associated test program. They may be applied in the design of the DUT, in the generation of the test program by an ATPG tool, in making adjustments to a STIL file for a particular tartet tester, or in the load process on a test system. See Clause 1 for a discussion of these operations.

tester rule: A single statement in a TRC file; the action of applying a tester resource constraint to a test pattern.

tester targetting: Refers to the process of making a STIL file/stream ready for consumption by a test system. The process may involve checking resources, mapping resources, or making adjustments to the STIL file. See Clause 1 for a discussion of these operations.

time_expr: A time value or a time expressions as defined in STIL.0.

time set: A tester resource that defines the period and the timing of all signals across a segment.

timing generator: A tester resource that defines period and signal timing; it may contain multiple time sets; timing values may be statically or dynamically selected.

usage model: The various flows or ways that a TRC file/stream may be used or applied. See Clause 1 for a discussion of these operations.

vector: A STIL statement inside a pattern block that defines the activity (via a WFC) on the signals of the device.

vector rate: The frequency at which vectors are applied to a DUT; the frequency of application of vectors in a pattern as defined by the period statements in the waveform tables.

waveform characteristic: The set of attributes that define the capabilities of a tester in the creation of waveforms; this is opposed to waveform descriptions that define the exact waveform.

waveform description: The definition of exact waveforms that can be created on a tester; this is opposed to waveform attributes that define only the attributes of the waveforms.

waveform generator: Refers to the tester resource that creates the input event sequence and/or output compare event sequence for application to the DUT.

Annex B

(informative)

Fluid concepts in parameter specification

In many cases, a given ATE system can be configured in many ways with optional or modular additions to the hardware of a system. This flexibility is referred to in this document as a “fluid” specification and is handled by means of new statements that have been added to the Variables block: Assert, ConfigConstant, and ParamConstant (see Clause 8 for the syntax definition).

Two types of parameters may be defined: 1) fixed attributes of a tester that relate to a specific model of tester such as max pin size or hardware range options, which are defined with the ConfigConstant statement, and 2) variable attributes that are selectable by a given application and are defined with the ParamConstant statement. The Assert statement is used to specify the allowed relationships between the various parameters.

Subclauses B.1 and B.2 illustrate examples of fluid specification.

B.1 Example of pins in a segment

This is an example of an ATE system that has variable pin assignments. There are two configurations of the tester defined, one with a maximum of 1024 pins and the other with 512 pins. In both cases, the tester may have either 1 or 2 segments with the pins being assigned to the segments in increments of 64. The following TRC/STIL code defines this case:

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause B.1 *}
}
Variables {
 ConfigConstant MAX_PINS;
 Assert (MAX_PINS :== 512) || (MAX_PINS :== 1024);
 ParamConstant SEG_ONE_PINS;
 Assert (SEG_ONE_PINS % 64) :== 0;
 IntegerConst SEG_TWO_PINS := MAX_PINS - SEG_ONE_PINS;
}
Environment TESTER_SPECS {
 TRC {
   SignalAttributes SEG1 {
     MaxSignals SEG_ONE_PINS;
   }
 }
   SignalAttributes SEG2 {
     MaxSignals SEG_TWO_PINS;
   }
 Module MOD1 {
   SignalAttributes SEG1;
 }
 Module MOD2 {
   SignalAttributes SEG2;
 }
 } // end TRC
} // end Environment
```

B.2 Example of multiple ranges on the period

This example of fluid ATE constraints is defining allowed ranges on the period specification. The ATE system has the ability to switch between 16 different period values on-the-fly. Each of these periods must select from one of three ranges based on the value that is to be programmed into the period. Once the period range is selected, the specification for accuracy, resolution, and max edge times are determined from the period. The following shows how this is specified by means of ParamConstant definitions for the three parameters: PER, ACC, and RES, and an Assert statement that defines the allowed combinations of these three parameters.

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause B.2 *}
}
Variables {
 ParamConstant PER;
 ParamConstant ACC;
 ParamConstant RES;
 Assert
   (PER :== 100ns) && (ACC :== 200ps) && (RES :== 100ps) // RNG1
   || (PER :== 1us) && (ACC :== 5ns) && (RES :== 1ns) // RNG2
   || (PER :== 10us) && (ACC :== 10ns) && (RES :== 5ns) // RNG3
 ;
}
Environment TESTER_SPECS {
 TRC {
   PeriodAttributes {
     MaxPeriods 16; // max ranges switched on the fly
     TimeLimits 10ns <= @@ <= PER;
     Resolution RES;
     Accuracy ACC;
   } // end Periods
   WaveformAttributes {
     TimeLimits 0ns >= @@ >= (2*PER-10ns);
     Resolution PER/4096;
     Accuracy PER/8192;
   } )* // end WaveformAttributes
 } // end TestResourceConstraints
} // end Environment
```

Annex C

(informative)

Tester channel map

This clause makes use of the NameMap block as defined in Clause 17 of STIL.1.

In STIL.1, there is the definition of how to specify a mapping of Signals and other objects from the STIL pattern interchange environment to some other environment. This facility can be used to specify the mapping of a STIL test program to a set of tester channels on a tester. This clause is informational only as no new syntax is defined herein.

C.1 Tester channel map—syntax

Environment (ENV_NAME) {
    (**NameMaps** (MAP_NAME) {
        (**Signals** { (SIG_NAME "MAP_STRING";)* })*
    } *// end NameMaps*
} *// end Environment*

C.2 Tester channel map—example

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause C.2 *}
}

Environment ATE_SC212 {
 NameMaps WAFER {
   Signals {
     // sig-name "channel, type";
     SIG1 "13, INPUT";
     SIG2 "45, BIDI";
     SIG3 "46, OUT";
   } // end Signals
 } // end NameMaps WAFER
 NameMaps PACKAGE {
   Signals {
     SIG1 "42, INPUT";
     SIG2 "19, BIDI";
     SIG3 "16, OUT";
   } // end Signals
 } // end NameMaps PACKAGE
 NameMaps MULTISITE {
   Signals {
     SIG1 "INPUT 13 48 96";
     SIG2 "BIDI 45 83 99";
     SIG3 " OUT 46 85 128";
```

```
    } // end Signals
} // end NameMaps MULTISITE
NameMaps MULTISITE_PINGPONG {
  Signals {
    SIG1 "INPUT (13 48) (96 106)";
    SIG2 "BIDI  (45 83) (99 107)";
    SIG3 "OUT   (46 85) (128 126)";
  } // end Signals
} // end NameMaps MULTISITE_PINGPONG
} // end Environment
```

Annex D

(informative)

Example of TRC for a simple tester model

Four usage models (or flows) use the TRC statements (refer to Figure 1): 1) Tester Rule Checking, 2) Tester Resource Reporting, 3) Tester Resource Targeting, and 4) Tester Resource Loading. This example uses a simplified model of a per-pin architecture tester to illustrate the TRC syntax for flow 1 and flow 3. This simplified tester has 256 data channels with two drive and one compare edges per vector period. The vector rate is up to 200 MHz (i.e., 5 ns period). The regular vector memory has 64 000 000 and 2000 vectors of subroutine memory. The data channels can be either inputs or outputs. This tester has a single period generator with both resolution and accuracy of 10 ps for the usage model 1) and 3). Assuming the period value can be programmed between 5 ns to 4 us.

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* Annex D *}
}

Environment TESTER_RULES {                                      1
TRC SIMPLE {
 Usage Constraints;
 PeriodAttributes PERIOD_INFO {
   Accuracy 10ps;
   MaxPeriods 16;
   MaxPeriodGenerators 1;
   TimeLimits (@@ >= 5ns && @@ <= 4us);
   Resolution 10ps;
 } // end PeriodAttributes

 WaveformAttributes DATA_WAV {                                  2
   Accuracy  Edge 500ps;
   Accuracy  EdgeToEdge '1ns';
   CompareEvents L/H/X 1 1;
   DriveEvents D/U Z P 2 1;
   FormatSelect InOut {
     MaxShapes 64 Dynamic { SignalsPer 1 };
     MaxData DriveCompare 3 Dynamic;
     MaxTimeSets 64 Dynamic { SignalsPer 1 };
   } // end Formats DataWav
   MaxEdgeTime 4*4us;
   Resolution 40ps;
 }   // End Data Waveform Attributes

 PatternAttributes PATTERN_MEMORY {                             3
   Max Locations 16*1024*1024;
 }

 PatternAttributes SUBROUTINE_MEMORY {
   Max Locations 2*1024;
}

SignalAttributes {
  MaxSignals 256;
  InOut WithinCycle;
  PeriodAttributes PERIOD_INFO Synchronous;
  WaveformAttributes DATA_WAV;
  PatternAttributes PATTERN_MEMORY;
  PatternAttributes SUBROUTINE_MEMORY;
} // end SignalAttributes
} // end TRC Rules
} // end Environment
```

NOTES

1—The **PeriodAttributes** block contains the properties of the period generator of the corresponding ATE or ATE family. This specific ATE has a single period generator with 10 ps accuracy and resolution. The period value range is bounded within 5 ns to 4 us. It also identifies that the tester has the potential of 16 different period values that can be switched on-the-fly. There is only one period generator for this tester, so all signals are synchronized.

2—The **WaveformAttributes** block contains the data channel's waveform attributes of this tester. The data channel only has a single event per cycle with 500 ps edge accuracy. There are 64 timing sets available for each channel. Each channel is allowed to switch between I/O on-the-fly. Supported Output measure operations are Compare High, Compare Low, or Mask. Input states allowed are Drive high, Drive low, or tri-state.

3—The **SignalAttributes** block defines the properties of the data signals. If applied according to the usage model 1) Tester Rules Checking, it defines the data channels of a particular tester's capability. If applied according to the usage model 3) Tester Resource Targeting, it defines the data channels of a target tester (the configuration information). This tester has 256 data channels available. This tester has up to 16 Meg vectors behind each data channel.

Annex E

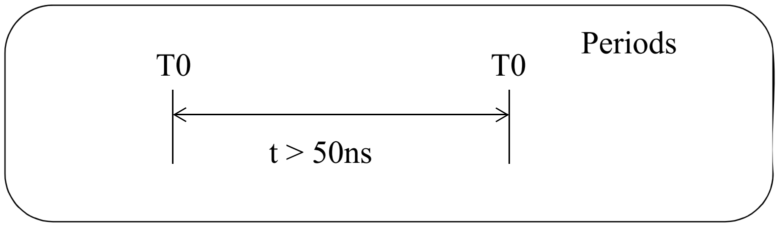
(informative)

Example of TRC used to define waveforms and timing[3]

Four usage models (or flows) use the TRC statements (refer to Figure 1): 1) Tester Rule Checking, 2) Tester Resource Reporting, 3) Tester Resource Targeting, and 4) Tester Resource Loading. This example shows the use of TRC to define explicit waveforms and the allowed timing for the waveforms. This example might represent actual waveforms on a specific tester. It might also represent an arbitrary set of waveforms that are to be allowed by a design or manufacturing process.

The examples in this annex use timing expression syntax that is defined in 5.1 of dot1. Refer to dot1 for definition of timing symbols such as: @@, @1, @T1, @T1.1.

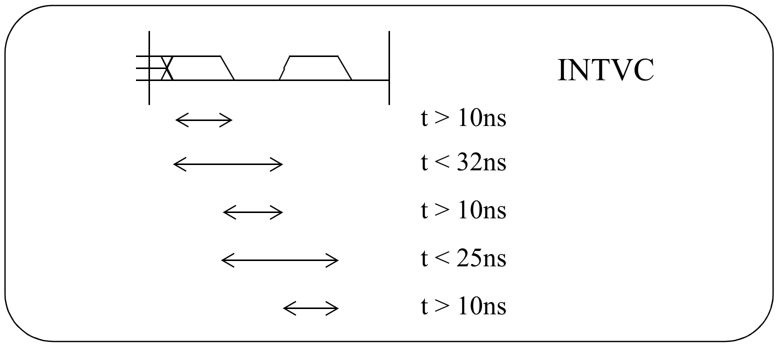
```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause  *}
}
Variables {
 ParamConstant MAX_VECTORS;
 ParamConstant MAX_FREQUENCY;
 Assert
   ( MAX_FREQUENCY <= 5MHz && MAX_VECTORS :== 65536)
   || ( MAX_FREQUENCY <= 10MHz && MAX_VECTORS :== 32768)
   || ( MAX_FREQUENCY <= 20MHz && MAX_VECTORS :== 16384);
 }
Environment TESTER_RULES {
TRC T3320 {
 Usage Constraints;
```



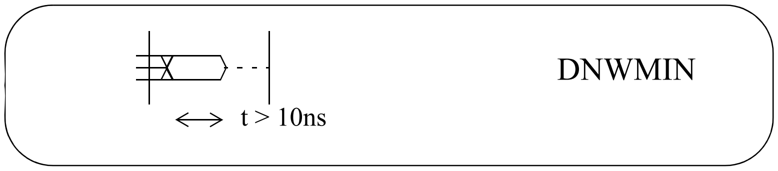
```
 PeriodAttributes PER_CHAR {
   MaxPeriods 1;
   TimeLimits @@ >= 1/MAX_FREQUENCY;
   Resolution 1ns;
 }
 WaveformAttributes WAV_CHAR {
   Resolution 1ns;
   FormatSelect InOut {
     MaxTimeSets 4 Dynamic;
```

[3]This annex is derived from information provided by Toshiba Corporation.

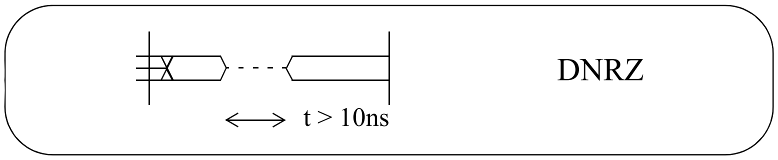
```
  }
}

WaveformDescriptions WAV_DESC Explicit {
```



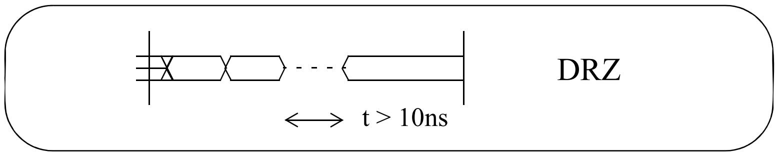
```
  INTVC {   // two cycle double pulse
    Shape {
      U/D/P;
      (@@>=@1+10ns) D;
      (@@>=@2+10ns && @@<@1+32ns) U/D;
      (@@>=@3+10ns && @@<@2+25ns) D;
    }
  }
```



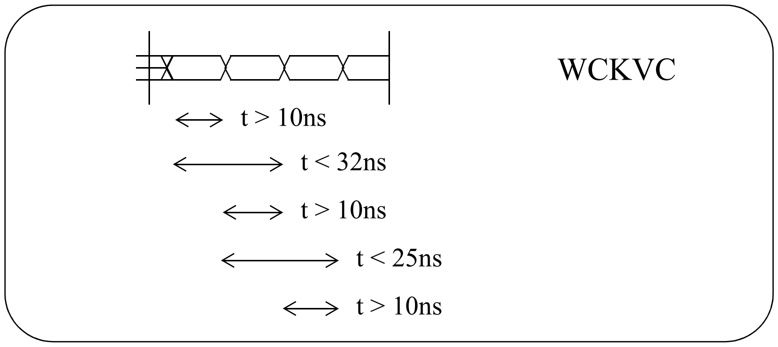
```
  DNWMIN {   // delayed non-return to zero
    Shape {
      U/D/P;
      (@@>@+10ns) Z;
    }
  }
```



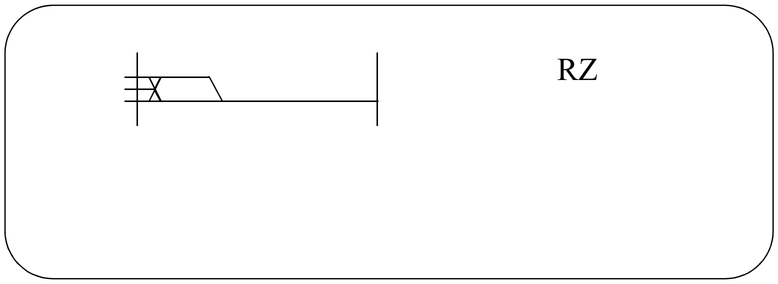
```
  DNRZ {   // DNRZ Bidir
      Shape {
      U/D/P;
      Z;
      (@@>@+10ns) U/D;
    }
  }
```



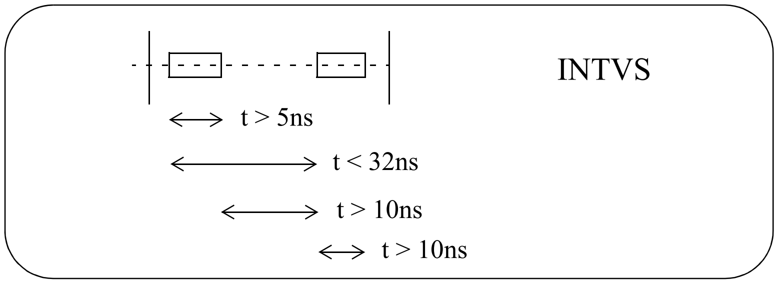
```
DRZ {   // DRZ Bidir
  Shape {
    U/D/P;
    U/D;
    Z;
    (@@>@+10ns) U/D/P;
  }
}
```



```
WCKVC {   // single cycle double pulse
  Shape {
    U/D/P;
    (@@>@+10ns) U/D;
    (@@>@2+10ns && @@<=@1+32ns) U/D;
    (@@>@3+10ns && @@<=@2+25ns) U/D;
  }
}
```



```
RZ {   // return to zero
  Shape {
    U/D/P;
    D;
  }
}
```



```
  INTVS {   // window compare
    Shape {
      L/H;
      (@@>@+5ns) X;
      (@@>@2+10ns && @@<=@1+32ns) L/H;
      (@@>@+5ns) X;
    }
  }
}     // end of WaveformAttributes

PatternAttributes {
  Max Vectors MAX_VECTORS;
}
SignalAttributes {
  MaxSignals 256;
  PeriodAttributes PER_CHAR;
  WaveformAttributes WAV_CHAR;
  WaveformDescriptions WAV_DESC;
}
}     // end of TestResourceConstraints T3320

}     // end Environment
```

Annex F

(informative)

Example of TRC used for resource reporting

Four usage models (or flows) use the TRC statements (refer to Figure 1): 1) Tester Rule Checking, 2) Tester Resource Reporting, 3) Tester Resource Targeting, and 4) Tester Resource Loading. The LS245 test pattern (refer to the Figure 3 of STIL.0, page 9 for the STIL pattern) is used as a basis for the TRC pattern resource report herein to illustrate flow 2 and flow 4. Details of this design can be found in the Annex E of STIL.0.

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause  *}
}

Environment PATTERN_LS245 {
TRC LS245_RESOURCES {
 Usage PatternReport;

 PeriodAttributes PERIOD_INFO {                                             1
   Accuracy 1ns;
   TimeLimits (@@>=500ns);
 } // end PeriodAttributes

 WaveformAttributes RTO_WAV { // OE_ needs be RTO format                    2
   Accuracy  Edge '500ps';
   DriveEvents U D/U 2 1;
   FormatSelect In {
     MaxShapes 1 Static;
     MaxData Drive 2 Dynamic;
   } // end FormatSelect
 } // End WaveformAttributes

 WaveformAttributes NRZ_WAV { // DIR and ABUS only have RTZ format          3
   DriveEvents U/D 1 1;
   FormatSelect In {
     MaxShapes 1 Static;
     MaxData Drive 2 Dynamic;
   } // end FormatSelect
 }   // End WaveformAttributes

 WaveformAttributes WINDOW_WAV { // BBUS uses window compare                4
   CompareEvents x h/l/x 2 1;
   DriveEvents Z;
   FormatSelect Out {
     MaxData Compare 2 Dynamic;
   } // end FormatSelect
 }   // End Data WaveformAttributes

 WaveformDescriptions LS245_CLOCK Explicit {                                5
```

```
  RTO_FORMAT {
    Shape {
      D/U;
      '@+100ns' U;
    } // end Shape
  }  // end In RTO Format
} // end WaveformDescriptions

SignalAttributes CLOCK {                                    (6)
  MaxVectorMemory 9;
  PeriodAttributes PERIOD_INFO Synchronous;
  WaveformAttributes RTO_WAV;
  WaveformDescriptions LS245_CLOCK;
} // end SignalAttributes

SignalAttributes DATA_IN {                                  (7)
  MaxSignals 9;
  MaxVectorMemory 9;
  PeriodAttributes PERIOD_INFO Synchronous;
  WaveformAttributes NRZ_WAV;
} // end SignalAttributes

SignalAttributes DATA_OUT {                                 (8)
  MaxSignals 8;
  MaxVectorMemory 9;
  PeriodAttributes PERIOD_INFO Synchronous;
  WaveformAttributes WINDOW_WAV;
} // end SignalAttributes
Module MOD {
  SignalAttributes CLOCK;
  SignalAttributes DATA_IN;
  SignalAttributes DATA_OUT;
}
} // end TRC Usage of LS245
} // end Environment
```

NOTES

1—The **PeriodAttributes** block contains the requirements of period for the LS245's STIL pattern. The period value required by the DUT and this STIL pattern is 500 ns.

2—The **WaveformAttributes RTO_WAV** block contains the signal OE_'s attributes of LS245. The OE_ signal needs to have a single RTO format that has a drive to data substitute event (D/U) and a drive up event (U).

3—The **WaveformAttributes NRZ_WAV** block contains the signal attributes of DIR and ABUS of LS245. These signals only need to have a single event (or NRZ format).

4—The **WaveformAttributes WINDOW_WAV** block contains the signal attributes of BBUS of LS245. The BBUS signal needs to have a window strobe with two timing values. It needs the X state to turn off the window strobe. It may need to have the **DriveEvents** statement, if the signal BBUS is declared as an InOut signal (as page 14 of STIL.0). The **DriveEvents** statement can be omitted, if the signal BBUS is declared as an Out signal (as page 9 of STIL.0).

5—The **WaveformDescriptions** block defines the OE_'s formats of LS245.

6—The **SignalAttributes** DATA_CLOCK block defines the corresponding property of the signal OE_. In this example, it has nine vectors for this signal (**MaxVectorMemory** statement). The rest of the statements refer to the period and waveform attributes. The keyword Synchronous indicates all signals are always synchronized together.

7—The **SignalAttributes** DATA_IN block defines the corresponding property of the data input signals: DIRS and ABUS. The **MaxSignals** statement indicates the STIL pattern has nine input signals that will be treated as data inputs, e.g., the DIR and ABUS signals. It refers to a predefined WaveformAttributes named as “NRZ_WAV”.

8—The **SignalAttributes** DATA_OUT block defines the corresponding property of the BBUS signals. The **MaxSignals** statement indicates the STIL pattern has eight signals. The **WaveformAttributes** statement refers to a predefined WaveformAttributes named “WINDOW_WAV”.

Annex G

(informative)

Examples of tester targeting and tester loading[4]

Four usage models (or flows) use the TRC statements (refer to Figure 1): 1) Tester Rule Checking, 2) Tester Resource Reporting, 3) Tester Resource Targeting, and 4) Tester Resource Loading. This is an example of flow 3 and flow 4.

A STIL test program must be loaded onto a tester for execution. This requires mapping the various statements of the STIL file/stream to the resources available on a given tester. This resource allocation process (as depicted graphically in Figure 1) can occur as a preload operation wherein the resource information is placed in the STIL file/stream, flow 3, or it can happen at the time the file is loaded onto the tester, flow 4.

G.1 Example of a resource assignment in a Pattern block

The following example shows the use of <<resource_id>> tags within a pattern block. Refer to Clause 9 for the syntax and definitions of the statements.

Two pragmas are contained in this example (refer to Clause 19 of STIL.1 for the definition of pragma). The content of the pragma is entirely dependent on the format/syntax of the consuming tester loader or translator. The pattern (P1) contains references to the content of these pragmas by means of the information in angle brackets. It is assumed in the pragma for “TESTER1” that there is some kind of tagging capability. In the pragma for “TESTER2,” the reference is by index number, which possibly, could correspond to a memory assignment in the tester memory.

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause G.1 *}
}

Signals { s[1..100] InOut; }

 // Load resource mapping memory for TESTER1
Pragma TESTER1 {*
 L1: ... tester statement ...                              ①
 L2: ... tester statement ...
 "L3-x": ... tester statement ...
*} // end Pragma TESTER1

// Load resource mapping memory for TESTER2
Pragma TESTER2 {*
 (0) ... tester statements ...                             ②
 (5) ... tester statements ...
 (13) ... tester statements ...
```

[4]This annex is derived from information provided by Credence Corporation.

```
*} // end Pragma TESTER2

Pattern P1 {
 Resource TESTER1 TESTER2;
         W wft1;                                        3
         C { s[1..100] = \r100 0; }
 <<L1 0>> V { s[5] = 1;
 <<L2 13>> V { s[1..4] = 1011; }                        4
 <<L1 5>> V { s[10,11] = HH; }
 <<"L3-x" 13>> V { s[1..4] = 0011; }
 << * 17 >> IddqTestPoint;
} // end Pattern P1
```

NOTES

1—This pragma block contains the loading instructions for “TESTER1”. For this example, there are three references that need to be resolved, named: L1, L2, and “L3-x”. The content of this block is arbitrary, as far as STIL is concerned, as long as it it bounded by the {* *} characters. In fact, even the tag syntax is arbitrary and is shown here only for informational purposes.

2—This second pragma block is included to show how a pattern can be targetted for two different testers within a single file. This second pragma is indexed by number rather than by tag as in the previous example.

3—The Resource statement at the beginning of pattern P1 serves two purposes. It indicates that the pattern has embedded within it the resource allocation tags for the purpose of tester targetting. It also indicates that the first identifier in the resource tag is to be associated with “TESTER1” and the second identifier is to be associated with “TESTER2”.

4—The pattern contains the resource tags as required by the loader/translator application. These resource tags are contained in angle brackets. Note also the use of * in the tag reference field wich indicates for TESTER1 that there is no reference for this statement.

G.2 Example of resource assignment in a Timing block

The following example shows the use of <<resource_id>> tags within a timing block. See also G.3 for an example of how to implicitly define common resources using the Inherit statement.

This example is different from the previous one in that no pragma information is defined. In this case it is showing only the waveforms and periods that are expected to be assigned to the same tester resource.

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause G.2 *}
}

Timing BASIC {
    WaveformTable ONE {
       <<PER1>> Period 500ns;
                                                                1
       DIR { <<SEQ1>> 01 { 0ns D/U; }}
       OE_ { 01 { 0ns U; 200ns D/U; 300ns U; }}                 2
       ABUS { <<SEQ1>> 01 { 10ns D/U; }}
       BBUS { <<SEQ1>> LHX { 0ns Z; 260ns L/H/X; 280ns T; }}
    } // end WaveformTable ONE
    WaveformTable TWO {
       <<PER1>> Period 500ns;
       DIR { <<SEQ2>> 01 { 0ns D/U; }}                              3
       OE_ { 01 { 0ns U; 200ns D/U; 300ns U; }}
       ABUS { <<SEQ2>> LHZX { 0ns Z; 260ns l/h/t/x; 280ns x; }}
       BBUS { <<SEQ2>> 01 { 10ns D/U; }}
    } // end WaveformTable TWO
    WaveformTable THREE {
       <<PER2>> Period 550ns;
       DIR { <<SEQ1>> 01 { 0ns D/U; }}
       OE_ { 01 { 0ns U; 200ns D/U; 300ns U; }}                     4
       ABUS { <<SEQ3>> 01 { 10ns D/U; }}
       BBUS { <<SEQ3>> LHX { 0ns Z; 260ns L/H/X; 280ns T; }}
    } // end WaveformTable THREE
    WaveformTable FOUR {
       <<PER3>> Period 550ns;
       DIR { <<SEQ2>> 01 { 0ns D/U; }}
       OE_ { 01 { 0ns U; 200ns D/U; 300ns U; }}
       ABUS { <<SEQ4>> LHX { 0ns Z; 460ns L/H/X; 480ns T; }}
       BBUS { <<SEQ4>> 01 { 10ns D/U; }}
    } // end WaveformTable FOUR
    WaveformTable FIVE {
       <<PER1>> Period 500ns;
       DIR { <<SEQ1>> 01 { 0ns D/U; }}
       OE_ { 01 { 0ns U; 200ns D/U; 300ns U; }}
       ABUS { <<SEQ2>> LHX { 0ns Z; 260ns L/H/X; 280ns T; }}
       BBUS { <<SEQ1>> LHX { 0ns Z; 260ns L/H/X; 280ns T; }}
    } // end WaveformTable FIVE
} // end Timing BASIC

Pattern BASIC {
 W FIVE; V { ALL = 00ZZZZZZZZXXXXXXXX; }
 W ONE;  V { ABUS = 00000000; BBUS = LLLLLLLL; }
 W THREE;V { ABUS = 10000000; BBUS = LHLLLLLL; }
 W THREE;V { ABUS = 00001000; BBUS = LLLLLHLL; }
 W TWO;  V { DIR = 1; ABUS = LLLLLHLL; BBUS = 00001000; }
 W TWO;  V { ABUS = LHLLLLLL; BBUS = 10000000; }
 W FOUR; V { ABUS = LHLLLLLL; BBUS = 10000000; }
} // End Pattern BASIC
```

NOTES

1—The resource tag <<PER1>> in front of the Period statement indicates that this is a resource that is to be shared with some other statements in the STIL file/stream.

2—The resource tag <<SEQ1>> in front of the two WFCs (0 and 1) indicates that this waveform resource is to be shared by other waveforms in the STIL file/stream.

3—This is a second reference to the <<PER1>> tag and indicates that the same tester hardware resource is to be used as the prior use of this resource tag.

4—The use of resource tag <<SEQ1>> in waveform table "THREE" indicates that the same tester hardware resource is to be used as in waveform table "one."

G.3 Example of implicit resource allocation in STIL

This example illustrates an alternative method of identifying shared resource assignments within a timing block. Compare and contrast this with the usage of the <<resource id>> tags, which accomplishes a similar purpose (see G.2). The difference between these two approaches is that the <<resource_id>> tags are added in a postprocessing flow, whereas the inheritance technique illustrated in this example is part of the originally generated code.

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause G.3 *}
} // end Header

Timing BASIC {
    WaveformTable PERIOD_ONE {
       Period 500ns;
    }
    WaveformTable PERIOD_TWO {
       Period 550ns;
    }
    WaveformTable PERIOD_THREE {
       Period 550ns;
    }
    WaveformTable SEQUENCE_ONE{
       Waveforms {
          DIR { 01 { 0ns D/U }}
          OE_ { 01 { 0ns U; 200ns D/U; 300ns U;}}
          ABUS { 01 { 10ns D/U; 300ns U;}}
          BBUS { 01 { 10ns D/U; 300ns U;}}
       } // end Waveforms
    } // end SEQUENCE_ONE

    WaveformTable SEQUENCE_TWO {
       Waveforms {
          DIR { 01 { 0ns D/U }}
          ABUS { LHX { 0ns Z; 460ns L/H/X; 480ns T; }}
          BBUS { LHX { 0ns Z; 460ns L/H/X; 480ns T; }}
       } // end Waveforms
    } // end SEQUENCE_TWO

    WaveformTable SEQUENCE_THREE {
       Waveforms {
          BBUS {  LHX { 0ns Z; 260ns L/H/X; 280ns T;}}
       } // end Waveforms
    } // end SEQUENCE_THREE

    WaveformTable SEQUENCE_FOUR {
       Waveforms {
          ABUS { LHX { 0ns Z; 460ns L/H/X; 480ns T; }}
          BBUS { 01 { 10ns D/U; }}
       } // end Waveforms
    } // end SEQUENCE_FOUR

```

```
   WaveformTable ONE {
      InheritWaveformTable PERIOD_ONE;
      Waveforms {
        DIR { InheritWaveform SEQUENCE_ONE.DIR; }
        OE_ { InheritWaveform SEQUENCE_ONE.OE_; }
        ABUS { InheritWaveform SEQUENCE_ONE.ABUS; }
        BBUS { InheritWaveform SEQUENCE_ONE.BBUS; }
      }   // end Waveforms
    } // end ONE

   WaveformTable TWO {
      InheritWaveformTable PERIOD_ONE;
      Waveforms {
        DIR  { InheritWaveform SEQUENCE_TWO.DIR; }
        OE_  { InheritWaveform SEQUENCE_ONE.OE_; }
        ABUS { InheritWaveform SEQUENCE_TWO.ABUS; }
        BBUS { InheritWaveform SEQUENCE_TWO.BBUS; }
      }   // end Waveforms
    } // end WaveformTable

    WaveformTable THREE {
      InheritWaveformTable PERIOD_TWO;
      Waveforms {
        DIR  { InheritWaveform SEQUENCE_ONE.DIR; }
        OE_  { InheritWaveform SEQUENCE_ONE.OE_; }
        ABUS { InheritWaveform SEQUENCE_TWO.ABUS; }
        BBUS { InheritWaveform SEQUENCE_THREE.BBUS; }
      }   // end Waveforms
    } // end WaveformTable

    WaveformTable FOUR {
      Period InheritWaveformTable PERIOD_THREE;
      Waveforms {
        DIR  { InheritWaveform SEQUENCE_TWO.DIR; }
        OE_  { InheritWaveform SEQUENCE_ONE.OE_; }
        ABUS { InheritWaveform SEQUENCE_ONE.ABUS; }
        BBUS { InheritWaveform SEQUENCE_FOUR.BBUS; }
      }   // end Waveforms
    } // end WaveformTable

    WaveformTable FIVE {
      InheritWaveformTable PERIOD_ONE;
      Waveforms {
        DIR  { InheritWaveform SEQUENCE_ONE.DIR; }
        OE_  { InheritWaveform SEQUENCE_ONE.OE_; }
        ABUS { InheritWaveform SEQUENCE_TWO.ABUS; }
        BBUS { InheritWaveform SEQUENCE_THREE.BBUS; }
      }   // end Waveforms
    } // end WaveformTable
  } // end Timing
```

Annex H

(informative)

Example of vector memory checking

A typical application environment will define multiple PatternAttributes blocks. These PatternAttributes blocks must be referenced from a SignalAttributes (where each SignalAttributes block contains a set of signal types) or from a Module block.

For instance:

```
Environment all_tricks {
  TRC trick1 {
    PatternAttributes "single-memory" {...}
    PatternAttributes "bidi-memory" {...}
    SignalAttributes SEPARATE_IO {
      PatternAttributes "single-memory";
    }
    SignalAttributes BIDI {
      PatternAttributes "bidi-memory";
    }
    Module {
      SignalAttributes SEPARATE_IO;
      SignalAttributes BIDI;
    }
  }
}
```

In this example, it is up the the application to determine which of the SignalAttributes blocks are appropriate to each signal.

H.1 Limitations to this approach

ATE contexts may handle one statement type in multiple ways. For instance, Loops of a single Vector statement may be processed differently than Loops containing multiple statements, and Macro and Procedure calls may be processed differently based on the presence and structure of arguments passed into the functions. It is not possible to take into account these types of effects without additional differentiating mechanisms to identify subattributes of each statement type or defining additional MaxVectorsCount statements to indicate sub-behaviors of how these statements may be applied in a set of patterns.

H.2 Application example

The following example demonstrates one application of the proposed constructs. The PatternAttributes block is used to define the maximum number of vectors available for each "type" of Pattern, although more accurately it represents the attributes of a specific type of memory.

The advantage of placing the memory behaviors under PatternAttributes blocks that are then associated with SignalAttributes is the ability to define multiple types of memories.

This example counts “sequencer memory” separately from “parallel vector memory,” with separate limits. A different architecture might place these two blocks into one large contiguous memory. If sequencer and parallel memory are contained in a single large count, the integer values specified in the sequencer block below can be inserted into the statements in the other two memory blocks (remember this syntax allows specifying multiple count attributes per statement) to count both sequencer effects and parallel vector effects against a single value.

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause H *}
}
Environment TRC {
 TRC "_example_" {
   PatternAttributes "sequencer" {
     Max Locations 1024 ;
     VectorCompression 0 ;
       // All macro and procedure bodies are expanded
       // into the Parallel Vector space for each call
     InstructionAttributes Macro Call {
       MaxNest 0 ;
     }
       // Shift and Loop operations generate a seq-jump and
       // iteration operation, and a seq-"return" to the main flow
     InstructionAttributes Shift Loop {
       MinBefore Locations 1 ;
       MinLength Locations 1;
     }
   }        // end PatternAttributes "sequencer"

   PatternAttributes "inlinemem" {
     Max Locations 1_000_000_000 ;
     Modulus Vectors (7 * 4) ;
     VectorCompression 1 ;
       // All macro and procedures are expanded
       // into the Parallel Vector space for each call
     InstructionAttributes Macro Call {
       MaxNest 0 ;
     }
       // Loops and Shifts are defined once
       // iteration is controlled by sequencer operations.
     InstructionAttributes Shift {
       MaxNest 1 ;
     }
     InstructionAttributes Loop {
       MaxNest 1 ;
       Modulus Vectors 7 ;
     }
   }        // end PatternAttributes "inlinemem"

   PatternAttributes "scanmem" {
     Max Locations 10_000_000_000 ;
     Modulus Vectors (7 * 4) ;
     VectorCompression 1 ;
```

```
    InstructionAttributes Macro Call {
      MaxNest 0;
    }
      // Shift data consumes the number of iterations
      // defined by the equivalent Vectors generated
    InstructionAttributes Shift {
      MaxNest 1 ;
    }
    InstructionAttributes Loop {
      MaxNest 1;
      Modulus Vectors 7 ;
    }
  }          // end PatternAttributes
  Module MOD {
    PatternAttributes "inlinemem" ;
    PatternAttributes "scanmem" ;
    PatternAttributes "sequencer" ;
  }
}              // end TRC
}              // end Environment
```

Annex I

(informative)

Waveform generator model

The waveform generation architecture of an ATE system can take on many different forms. The TRC rules as defined in this standard are flexible enough to describe commonly used architectures from waveform generation-per-pin to a central timing architecture. The following list provides the definitions of terms that are used in the syntax statements of 14.1 for specifying waveforms:

shape: This term refers to the wave shape that is created by the combination of the individual attributes listed below (time set, timing generator, data vaues, I/O values, mask values). If a given test system does not represent well using the individual attributes, then the term shape should be used, instead.

time set: This term is used to represent an indirect memory that is used to select timing generators that do the actual work of creating period and wave shape timing edges.

timing generator: This term is used to represent a function (usually a hardware function in a test system) that is used to create timing edges for the creation of period boundaries or waveform events.

data values: This term is used to represent the number of data values in the vector memory that are used for the construction of wave shapes.

I/O values: This term is used to represent the number of values in the vector memory that are used for the control of I/O switching in the formation of wave shapes.

mask values: This term is used to represent the number of values in the vector memory that are used for compare masking in the formation of wave shapes.

I.1 The general timing model

Figure I.1 shows the generic picture of an ATE system and how the constraint rules apply.

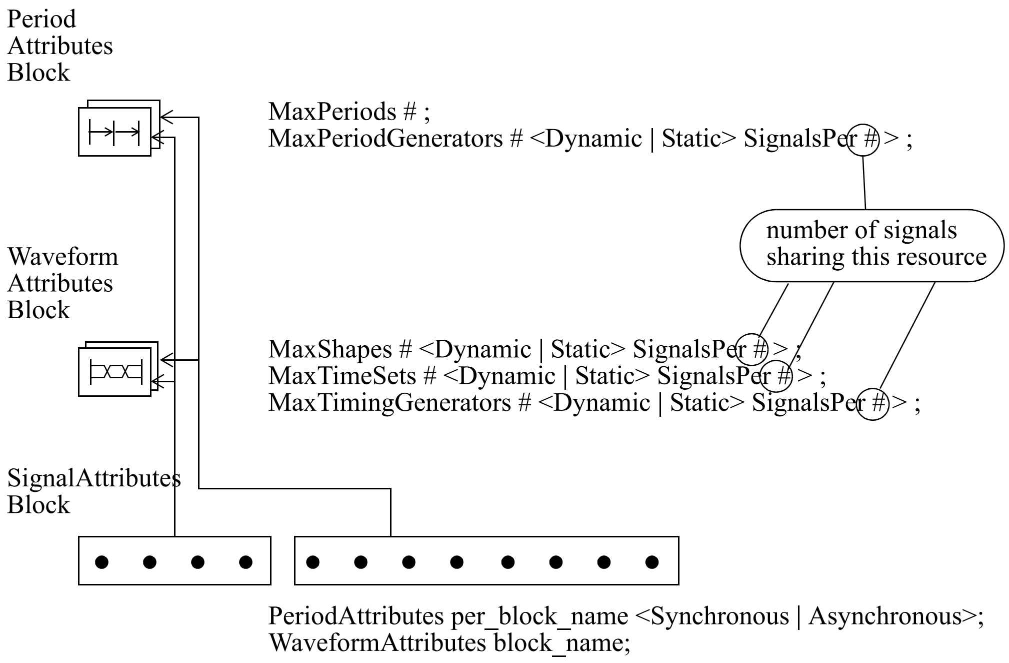

Period Attributes Block
MaxPeriods # ;
MaxPeriodGenerators # <Dynamic | Static> SignalsPer #> ;
number of signals sharing this resource
Waveform Attributes Block
MaxShapes # <Dynamic | Static> SignalsPer #> ;
MaxTimeSets # <Dynamic | Static> SignalsPer #> ;
MaxTimingGenerators # <Dynamic | Static> SignalsPer #> ;
SignalAttributes Block
PeriodAttributes per_block_name <Synchronous | Asynchronous>;
WaveformAttributes block_name;

Figure I.1—General timing model

I.2 Timing model with static timing selection

To better illustrate the timing resource assignment, Figure I.2 shows a system with the following attributes:

- — 120 signals arranged in two groups of 60.
- — 1 period generator for each group of 60 signals. Each period generator has 16 values that are selectable by the per signal time set select.
- — 16 timing generators can be assigned to any of the signals statically. Each TG has 16 values that are selectable by the per signal time set select.
- — 16 time sets for each signal that can be selected dynamically.
- — The selection of the PER and TG blocks is common between the two modules (indicated by the dashed line), thus making the two 60 pin modules run in lock-step as a 120 pin system.

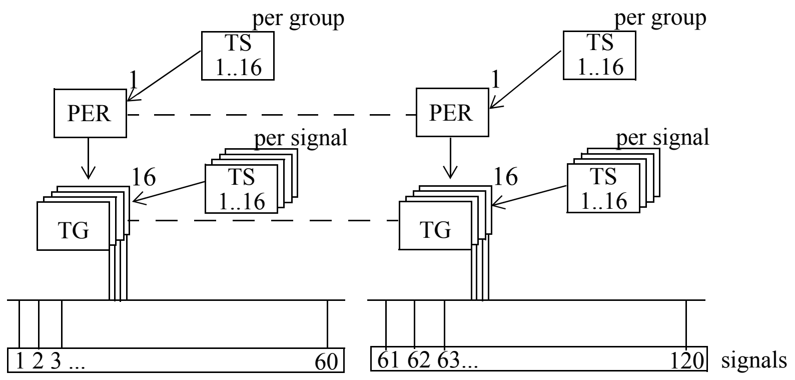


Figure I.2—Example of timing model with static timing select

Here is the code that defines the above architecture.

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause I.2 *}
}
Environment {
 TRC ATE_X {
   PeriodAttributes PER {
     MaxPeriods 16 Dynamic SignalsPer 120;
     MaxPeriodGenerators 1 Static SignalsPer 120;
   }
   WaveformAttributes WAV {
     FormatSelect InOut {
       MaxTimeSets 16 Dynamic SignalsPer 1;
       MaxTimingGenerators 16 Static SignalsPer 120;
     }
   }
   SignalAttributes {
     MaxSignals 120;
     PeriodAttributes PER Synchronous;
     WaveformAttributes WAV;
   }
 } // end TRC
} // end Environment
```

I.3 Timing model with per signal timing

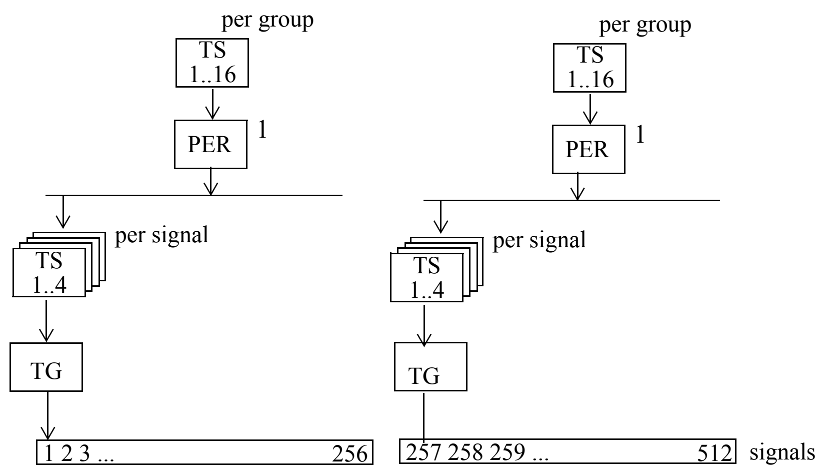


Figure I.3—Example of timing model with per signal timing

Here is the code that defines the above architecture.

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause I.3 *}
}
Environment {
 TRC ATE_Y {
   PeriodAttributes PER {
     MaxPeriods 16 Dynamic SignalsPer 256;
     MaxPeriodGenerators 2 Static SignalsPer 256;
   }
   WaveformAttributes WAV {
     FormatSelect InOut {
       MaxTimeSets 4 Dynamic;
     }
   }
   SignalAttributes {
     MaxSignals 1024;
     PeriodAttributes PER Synchronous Asynchronous;
     WaveformAttributes WAV;
   }
 } // end TRC
} // end Environment
```

I.4 Timing model with waveform select memory

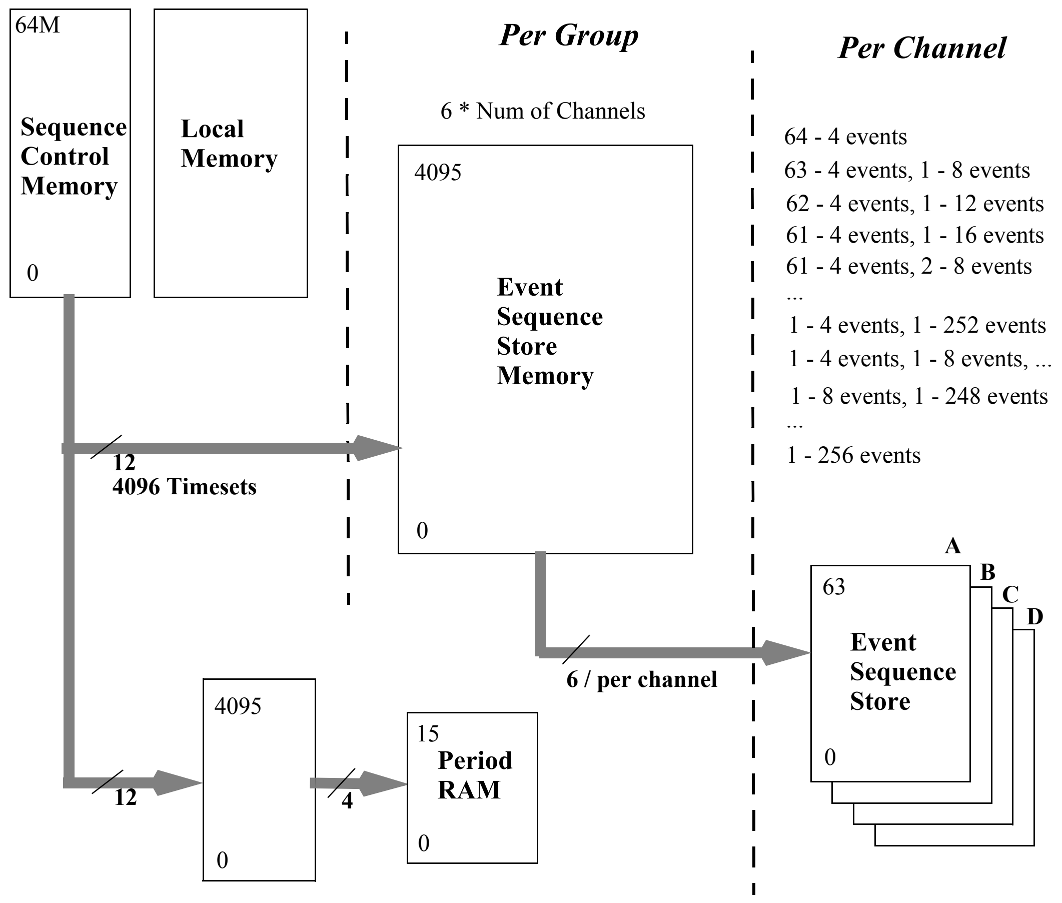

64M
Sequence Control Memory
0
Local Memory
Per Group
6 * Num of Channels
4095
Event Sequence Store Memory
0
12
4096 Timesets
Per Channel
64 - 4 events
63 - 4 events, 1 - 8 events
62 - 4 events, 1 - 12 events
61 - 4 events, 1 - 16 events
61 - 4 events, 2 - 8 events
...
1 - 4 events, 1 - 252 events
1 - 4 events, 1 - 8 events, ...
1 - 8 events, 1 - 248 events
...
1 - 256 events
A
B
C
D
63
Event Sequence Store
0
6 / per channel
4095
0
12
4
15
Period RAM
0

Figure I.4—Example of timing model with waveform/event select memory

The following code shows is for the above architecture with select memory:

```
STIL 1.0 { Design 2005; TRC 2007; }
Header {
 Source "IEEE Std 1450.3-2007";
 Ann {* clause I.4 *}
}
Environment {
 TRC ATE_Z {
   PeriodAttributes PER {
     MaxPeriods 16 Dynamic SignalsPer NUM_PINS;
     MaxPeriodGenerators 1 Static SignalsPer NUM_PINS;
     PeriodSelectMemory 4096 SignalsPer NUM_PINS;
   }
   WaveformAttributes WAV {
     FormatSelect InOut {
       MaxTimeSets 64 Dynamic;
       MaxTimingGenerators 1 SignalsPer 1;
       WaveformSelectMemory 4096 SignalsPer NUM_PINS;
```

```
        SelectWithPeriod;
      }
    }
    SignalAttributes {
      PeriodAttributes PER Synchronous Asynchronous;
      WaveformAttributes WAV;
      MaxSignals NUM_PINS;
    }
  } // end TRC
} // end Environment
```

Annex J

(informative)

File encryption

It is recognized that the TRC information may contain sensitive proprietary information about either a specific tester architecture or a methodology for transferring data to a tester format. It is beyond the scope of this standard to define or recommend an encryption technique. However, the following suggestions may be helpful:

— Tester resource constraint information (i.e., TRC block of STIL data) should be maintained as separate files from the STIL pattern data. This allows for the TRC information to be protected by encryption while allowing the pattern files to be viewed.

— The encryption technique could be established as a tool–tool protection scheme; i.e., tool-A (perhaps an ATE rule generator tool) could produce TRC files that tool-B (perhaps a specific pattern generation tool) can decrypt.

— The encryption technique could be established as a user–user protection scheme; i.e., the user of tool-A produces TRC files that tool-B can only read if the decryption password is known.

Annex K

(informative)

Regular expression reference

A regular expression (abbreviated as regexp or regex, with plural forms regexps, regexes, or regexen) is a string that describes or matches a set of strings, according to certain syntax rules. Regular expressions are used by many text editors and utilities to search and manipulate bodies of text based on certain patterns. Many programming languages support regular expressions for string manipulation. For example, Perl and Tcl have a powerful regular expression engine built directly into their syntax.

For a detailed definition of regular expressions, refer to *Mastering Regular Expressions*, by Jeffrey E. F. Friedl.[5]

To implement regular expressions into a STIL reader, the "Free Software Foundation" software package[6] is recommended. Information on the gnu regular expression software package can be found at:

http://directory.fsf.org/regex.html

The source code is available in a tar file at:

http://ftp.gnu.org/pub/gnu/regex/regex-0.12.tar.gz

The version of this software available at the time of this writing is version 0.12, which was released on 1993-04-12 and is reported as "stable" (meaning the software does not seem to be undergoing revisions).

The actual documentation of the GNU regular expression handler is found on several websites (it is part of the software library package as well). Here are two sites where it can be found:

http://www.codeforge.com/help/GNURegularExpr.html
http://www.gnu.org/software/grep/doc/grep_7.html#SEC7

[5]Friedl, J.E.F. *Mastering Regular Expressions*. Sebastopol, CA: O'Reilly Media, Inc., 2006. ISBN: 0596528124. This publication is available from O'Reilly Media, Inc., 1005 Gravenstein Highway North. Sebastopol, CA 95472, USA (http://www.oreilly.com).
[6]"fsf" is the home location of the "Free Software Foundation," a.k.a. "the GNU home.