IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data—Core Test Language (CTL)

1450.6™

IEEE Computer Society

Sponsored by the
Test Technology Standards Committee

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

5 April 2006

IEEE Std 1450.6™-2005

IEEE Std 1450.6™-2005(R2011)

IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data—Core Test Language (CTL)

Sponsor

Test Technology Standards Committee
of the
IEEE Computer Society

Approved 17 November 2005
IEEE-SA Standards Board

Reaffirmed 16 June 2011
IEEE-SA Standards Board

Approved 29 December 2005
American National Standards Institute

Reaffirmed 26 July 2012
American National Standards Institute

Abstract: The Core Test Language (CTL) is a language created for a System-on-Chip flow (or SoC flow), where a design created by one group is reused as a sub-design of a design created by another group. In an SoC flow, the smaller design embedded in the larger design is commonly called a core and the larger design is commonly called the SoC. The core is a design provided by a core provider, and the task of incorporating the sub-design into the SoC is called Core System Integration.

Keywords: Core Test Language (CTL), Standard Test Interface Language (STIL), System-on-Chip (SoC), wrapped core, unwrapped core

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

Print: ISBN 0-7381-4804-0 SH95373
PDF: ISBN 0-7381-4805-9 SS95373

Introduction

This introduction is not part of IEEE Std 1450.6-2005, IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data—Core Test Language (CTL).

CTL started as a language in the IEEE Std 1500™-2005 standardization activity for core test. This activity provided a representation mechanism for test information that exchanges hands between a core provider and the system integrator. Thus, the language had a charter to provide a mechanism for reuse of test patterns and information that allows for successful design for test and automatic test pattern generator activities on the SoC. As part of IEEE Std 1500-2005, CTL was designed to represent details about the IEEE 1500 wrapper. As CTL and the wrapper technology matured, it became apparent that the two activities should be separated into two standard documents. As a result of this decision, CTL, the language, became IEEE Std 1450.6 activity, and the information model for cores that uses CTL remained in IEEE Std 1500-2005.

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 CTL Working Group:

Rohit Kapur, *Chair*

Mike Collins	Brion Keller	Maurice Lousberg
Douglas Kay		Paul Reuter

When the CTL Working Group approved this standard, it had the following short-term membership:

Bill Chown	Yuhai Ma

Contents

IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data—Core Test Language (CTL)

1. Overview

1.1 General

The Core Test Language (CTL) is a language created for a System-on-Chip flow (or SoC flow), where a design created by one group is reused as a sub-design of a design created by another group. In an SoC flow, the smaller design embedded in the larger design is commonly called a core and the larger design is commonly called the SoC. The core is a design provided by a core provider, and the task of incorporating the sub-design into the SoC is called Core System Integration.

CTL is a language designed to be the transfer mechanism of test knowledge between a core provider and a system integrator to allow for interoperability between the producer and the consumer of the information. It facilitates the reuse of test patterns provided for a core for application from the SoC boundary. Although the language is general (the limitations of CTL are listed in 1.5) and can be used in many different ways, this standard is focused on the use of CTL for SoC designs. Thus, CTL allows for

a) Representation of design constructs and characteristics that are needed to be made visible by the core provider.
b) Representation of test patterns that are to be reused for cores in an SoC test flow.

CTL provides the language for communication of test information. An adjacent IEEE standard activity (IEEE Std 1500™-2005)[1] defines the information requirements of the core provider that are required to be represented in CTL. As a result of this relationship with IEEE Std 1500-2005, CTL has some constructs that

[1]For information on references, see Clause 2.

are there to support IEEE Std 1500-2005's informational requirements for wrapped and unwrapped cores. In an SoC flow (where cores are reused in SoCs), CTL represents all of the test information about the core such that the core can be successfully embedded in the SoC from a test perspective. CTL is a language that provides the information about the structures in the core that are to be reused at the next level of integration. Test reuse is also about the reuse of test patterns. As a result, CTL allows for the description of reusable test patterns (portable tests). CTL leverages existing standard representations. The CTL language is the union of the syntax defined in IEEE Std 1450™-1999, IEEE Std 1450.1™-2005, IEEE Std 1450.2™-2002, and this standard. CTL extends the definitions defined in IEEE Std 1450-1999 and IEEE Std 1450.1-2005 with extensions and exceptions defined in this standard. The reader is assumed to have knowledge of these standards, and their associated content is not repeated or explained in this standard.

CTL has the following characteristics to support the SoC test:

a) CTL can describe constructs to handle a wide variety of cores.
b) CTL does not limit the test methodologies of the core provider.
c) CTL can describe IEEE Std 1500-2005 wrapped and unwrapped cores. (Refer to the associated standard for details.)
    1) Unwrapped cores can be described in CTL to aid in the creation of the wrapper.
    2) Wrapped cores can be described in CTL to allow for the reuse of the wrapper in the integrated SoC.
    2) CTL describes the patterns of the wrapped or unwrapped core such that modifications can be made to the core for application from the SoC boundary.

CTL handles this variety of needs by providing a test mode structure within which primitive concepts are pieced together to describe the test information. Several sets of keywords are provided that form the vocabulary of CTL. The keywords are put together to represent information that are sentences of test information. CTL's ability to handle different designs and their associated test methodologies comes from its reliance on sequences. Predefined sequence types (EstablishMode and TerminateMode) are used to treat the different design configurations (test modes) in a uniform way. Sequence information is leveraged from IEEE Std 1450-1999 and IEEE Std 1450.1-2005 syntax, with the difference that the information is driven from the CTL Environment and test modes.

1.2 SoC flow

As mentioned, an SoC flow is the process in which a sub-design (core) is embedded in a larger design (SoC). The major steps of this flow are shown in Figure 1. Let us assume that CTL exists and it can describe everything needed for the core. Figure 1 shows three major steps in the process of designing the testing mechanism and the test patterns for embedded cores. The first step shows the "Core Design" work. This work is performed in such a manner that the resultant core can be reused in multiple designs with NO change required to either the core design or to the bulk of the test patterns information (data portion of the test patterns) that are developed for testing the core.

The first section in the diagram shows the core design process where the "IP core model" is represented (e.g., a Verilog or VHDL design). Along with the design, a set of "Core Test Patterns" in CTL syntax is generated (syntax from IEEE Std 1450-1999, IEEE Std 1450.1-2005, IEEE Std 1450.2-2002, and this standard). A complete CTL description encompasses test patterns and its associated constructs and a description of the various test modes of the core. Typical cores support multiple configurations that serve different purposes such as internal testing of the core and external testing of the logic outside the core. CTL information is partitioned across these configurations (test modes). The entire description of CTL is centered around the Environment (this is the top-level block of statements in the language), which is shown in Figure 1. The information requirements for cores is defined by IEEE Std 1500-2005. The requirements ensure that enough information is present to create the test for the finished SoC.

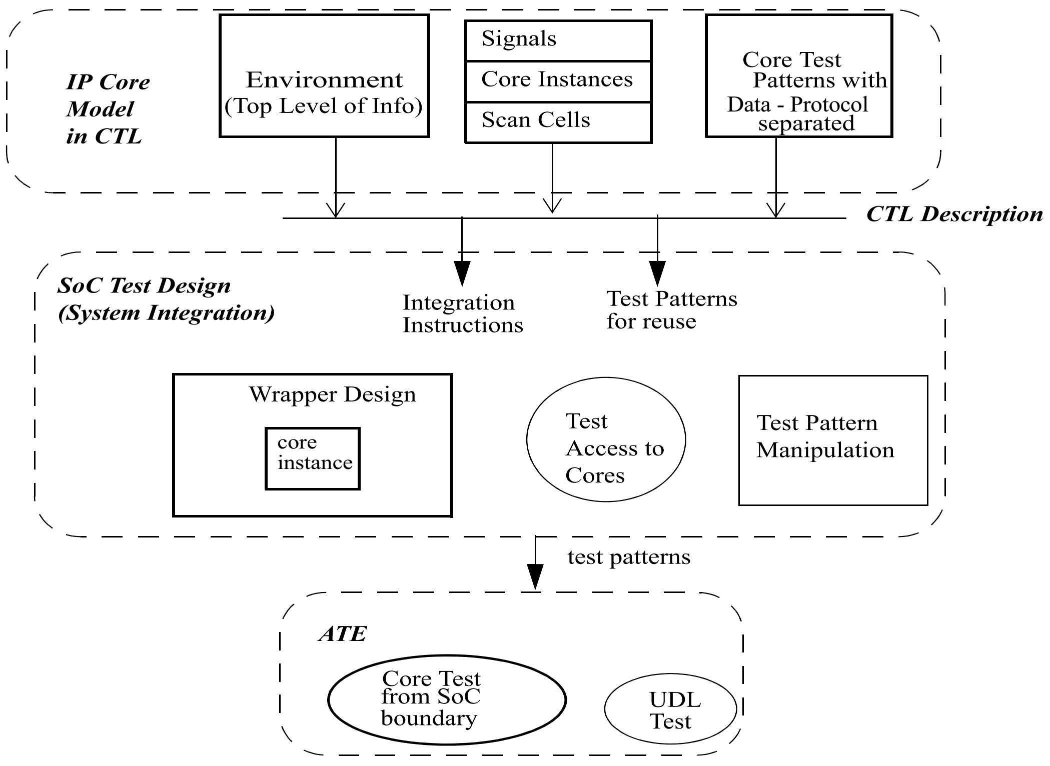


Figure 1—SoC flow showing the CTL use model

The second section of Figure 1 shows the System Integration process. In this process, the SoC integrator makes use of the CTL description of the core design to create the tests for the SoC. This process could involve multiple activities depending on the design and test methodologies in use. The following are some activities that the system integrator may perform:

a) *Wrapper Design*: Wrapper technology represents isolation hardware at the boundary of cores for test purposes. The wrapper isolates the testing of the core logic from the testing of the user-defined logic on the SoC. IEEE Std 1500-2005 defines such technology. Cores may or may not be provided with wrappers. If the test methodology relies on wrapping cores, then the system integration step would look for information in CTL of the core to determine whether a wrapper is provided with the core. If the wrapper is not provided, the CTL information of the core would be used to determine the wrapper cells and other details of the wrapper technology.
b) *Test Access to Cores*: The connections from the core boundaries to the top level of the SoC depends on the different cores embedded in the SoC. The design of the associated design entities depends on the scheduling of the tests of the different cores, which is determined by looking at the CTL that comes with each core.
c) *Test Pattern Manipulation*: The patterns that come with the CTL of every core are written to the boundary of the core. The process of converting these patterns to be applicable at the boundary of the SoC is called test pattern migration or test data porting. The process uses the wrapper technology and test access to cores as relevant to the SoC design. Patterns supplied in CTL have the data and protocol (event sequencing) portion separated. The data represent the 1s and 0s of the test patterns, which represent the bulk of the information. The protocol represents the sequence of the test patterns to apply the data. Test pattern migration is the process of modifiying the protocols of the test patterns

such that the test data are oriented to the SoC boundary. CTL information assists in test scheduling tasks on the SoC.

The last section shows the process of testing the manufactured SoC. The core-level tests are combined with the top-level logic tests to provide for complete testing of the SoC on an ATE. The creation of the top-level patterns of the SoC (UDL test) was not shown in the system integration step of Figure 1, but this task typically uses CTL information for external test or low leakage configurations of the cores on the SoC. If the information in last section is packaged in new CTL, then the SoC can be a core for another level of the design.

1.3 Scope

Unless the logic inside embedded cores can be merged with the surrounding user-defined logic (UDL), the SoC test requires reuse of test data and test structures specific to individual cores (designs) when integrated into larger systems. This standard defines language constructs sufficient to represent the context of a core and of the integration of that core into a system, to facilitate reuse of test data previously developed for that core. The SoC test also requires that the core be embedded in the SoC to allow for efficient testing of the logic external to the core. To that effect, this standard defines constructs that represent the test structures internal to the core for reuse in the creation of the tests for the logic outside the core. This provides constructs that will allow for the wrapping operation of an unwrapped core and the necessary wrapper specific information for a wrapped core. In particular, CTL shall support IEEE Std 1500-2005 for the information needs for wrapped and unwrapped cores. Semantic rules will be defined for the language to facilitate interoperability between the different entities (the core provider, the system integrator, and the automation tools) involved in the creation of an SoC. This standard is limited to SoC testing with multiple and/or hierarchical cores through digital interfaces.

All constructs defined in the CTL shall be consistent with IEEE Std 1450-1999 and extensions (STIL) to support the complete description of the test for cores integrated into SoC environments. Although the preferred syntax for the bulk of the test data is STIL, this language provides constructs for linking other test data representations to incorporate legacy cores. The constructs in the language shall support a vast variety of cores and different test methodologies with particular support for the IEEE 1500 standard for embedded core testing. These constructs shall facilitate the transportation of test information from the core provider to the system integrator and support test automation by providing a consistent and uniform definition of the constructs such that the information provided by a core provider is understood in the same way by the system integrator and the tools developed by EDA.

1.4 Purpose

To develop a language that will provide a sufficient description of a core to support reuse of test data developed for that core after integration into SoC environments, and to enable the creation of test patterns for the logic on the SoC external to the core.

1.5 Limitations of this standard

As the development of the language for complete functionality is a huge task, the first version of this standard is limited in its functionality. CTL, as described in this standard, has limited or has no support for the following areas:

a) Analog testing
b) Memory testing
c) Diagnostics and debug applications
d) ATE interfacing of DFT information

All of these areas are considered important and should be addressed in future extensions of this language.

1.6 Structure of this standard

This standard is to be used with IEEE Std 1450-1999, IEEE Std 1450.1-2005, and IEEE Std 1450.2-2002. The conventions established and defined in IEEE Std 1450-1999 are used in this standard and are included verbatim below.

Many clauses in this document add additional constructs to existing clauses in IEEE Std 1450-1999, IEEE Std 1450.1-2005, and IEEE Std 1450.2-2002 and are so identified in the title. Most constructs defined in this standard are limited to the Environment block, which is defined by IEEE Std 1450.1-2005.

The following is a copy of the conventions as defined in IEEE Std 1450-1999 and followed by this standards.

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 zero or one time.

e) ()+ indicates syntax that may be used one or more times.

f) ()* indicates optional syntax that may be used zero or more times.

g) <> indicates multiple choice arguments or syntax.

h) Defaults where appropriate are underlined in the 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:

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

The language definition clauses contain statements that use the phrase “it is an error”. This phrases indicates improperly defined CTL information.

2. Normative references

CTL encompases syntax defined in IEEE Std 1450-1999, IEEE Std 1450.1-2005, IEEE Std 1450.2-2002, and this standard. The language extends syntax defined by IEEE Std 1450-1999, IEEE Std 1450.1-2005, and IEEE Std 1450.2-2002. Exceptions, if any, to these standards are defined in this standard. The following referenced documents are indispensable for the application of this document. 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™-1999, IEEE Standard Test Interface Language (STIL) for Digital Test Vectors.[2, 3]

IEEE Std 1450.1™-2005, Standard for Extensions to Standard Test Interface Language (STIL) (IEEE Std 1450-1999) for Semiconductor Design Environments.

IEEE Std 1450.2™-2002, IEEE Standard for Extensions to Standard Test Interface Language (STIL) for DC Level Specification.

IEEE Std 1500™-2005, Testability Method for Embedded Core-based Integrated Circuits.

3. Definitions, acronyms, and abbreviations

3.1 Definitions

For the purposes of this standard, the following terms and definitions apply. As CTL is the union of the syntax defined in four documents, the terminology defined in the other documents (IEEE Std 1450-1999, IEEE Std 1450.1-2005, and IEEE Std 1450.2-2002) is applicable in this standard. *The Authoritative Dictionary of IEEE Standards Terms* should be referenced for terms not defined in this clause.

3.1.4 black box: A condition where the netlist of the design refered to is not available. Commonly used in association with a core, namely, a black box core.

3.1.5 core: Embedded hierarchy of the design. This hierarchy could be created by an imaginary boundary that isolates a portion of the logic in a design.

3.1.6 test mode: It is a configuration of the design that is defined in CTL as a CTLMode. This type of configuration in CTL is identified by the TestMode statement within the CTLMode block of statements.

3.1.7 white box: A condition where the netlist of the design refered to is available. Commonly used in association with a core, namely, a white box core.

3.2 Acronyms and abbreviations

1450.0	IEEE Std 1450-1999
1450.1	IEEE Std 1450.1-2005 extension to STIL
1450.2	IEEE Std 1450.2-2002 extension to STIL
ATE	automated test equipment
ATPG	automatic test pattern generator
BIST	built-in self-test
CTL	Core Test Language (aggregate syntax of 1450.6, 1450.1, 1450.2, and 1450.0)
DFT	design for test (it represents circuit modifications such as scan chains that are created for test)
IC	integrated circuit
SoC	System on Chip
STIL	Standard Test Interface Language (combination of 1450.0, 1450.1, and 1450.2)
TAM	test access mechanism used to connect to the (wrapped) cores being tested
UDL	user-defined logic (a moniker for any logic that may surround an embedded core)
WFC	WaveformCharacter

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

4.CTL orientation and capabilities tutorial

4.1 Introduction

Design reuse methodologies have allowed for the partitioning of effort needed to create a complete design at the expense of communication between the teams creating the design. Tasks when partitioned are performed by teams that are in close proximity to each other. In such situations, the teams can get together with frequent formal meetings, share common documentation, or discuss problems informally. This is a normal business process for any company that has a reasonably sized project under way. This breaks down when the design teams are separated in time and space. To avoid unreasonable communication problems, the process needs to be formalized and the interface between the core providers and the integrators needs to be standardized. This is where CTL as an industry-wide standard steps in. CTL is developed to address the information transfer needs of the SoC test.

Through CTL, all test aspects of cores can be described such that a system integrator can integrate a core as a black box into a SoC and perform all of the usual test tasks as though the core was a white box with test patterns to be reused. As shown in Figure 2, CTL would describe all information about the core needed by the system integrator. The language is designed to be manually written and created and/or consumed by test automation tools.

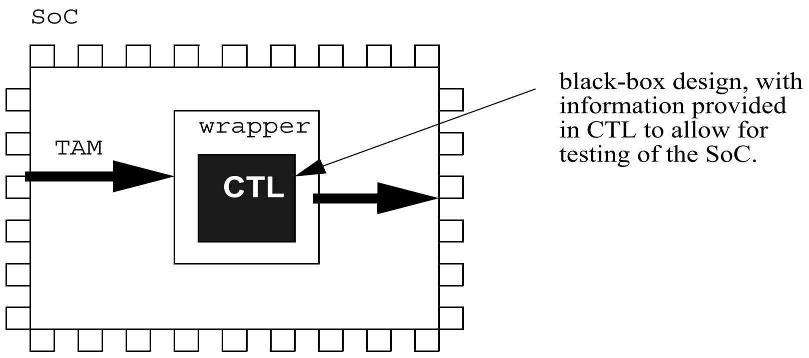

SoC
TAM
wrapper
CTL
black-box design, with information provided in CTL to allow for testing of the SoC.

Figure 2—CTL used to represent the test information of a core instead of a netlist

Figure 2 shows a usage of CTL where the information about the design that is to be embedded as a core is represented in CTL. Using the CTL of the embedded design (the core of the SoC), a wrapper can be constructed, and the appropriate TAM, which creates the pathway for the test data of the core, can be determined based on the test constraints in the CTL of the core. The figure depicts the core represented in CTL as a black box as the netlist is not provided for the core. A box around the black box represents a scan wrapper implemented outside the core, and the arrows represent the access mechanism to the wrapper. Once all structures are in place, the test patterns that are also a part of CTL can be retargeted to the boundary of the SoC. CTL is the language to support all information that the core provider needs to give the system integrator such that the integrator can successfully test the embedded core and any UDL around the core. All language constructs defined in CTL would work with all types of digital cores, their different test methodologies, and the different ways in which they are integrated in the design.

Information in CTL is partitioned by the modes of operation of the design being described. Each test mode has its information as relevant for the design and communication needs. Not all DFT used in a design needs to be described to the system integrator of the design.

In the examples, we show how aspects of the information that describes the test details of a design are presented in CTL. This information is part of the content that passes between the core provider and the system integrator in an SoC flow. The information includes

— Design configuration information
— Structural information
— Test pattern information

These pieces of information are critical to the information needs of the design for the successful testing of the SoC. CTL provides a mechanism to represent test information for the test needs in an SoC flow. The information model is not in the scope of this standard and is addressed in IEEE Std 1500-2005 with complete examples. In this standard, the mechanics to represent information in CTL are described.

NOTE—CTL is part of the STIL series of standards, and the language encompasses the syntax defined by IEEE Std 1450-1999, IEEE Std 1450.1-2005, and IEEE Std 1450.2-2002. The reader is assumed to be familiar with the content of 1450.0, 1450.1, and 1450.2, and thus, they are not reintroduced here. Examples of IEEE Std 1450-1999, IEEE Std 1450.1-2005, and IEEE Std 1450.2-2002 portion of the syntax are given in their corresponding documents.[4]

4.2 CTL for design configurations

When writing the language CTL, one is essentially describing test modes in the Environment block of statements. Within the Environment, each test mode represents a design configuration described in CTLMode blocks (*CTLMode {}*). Each test mode contains information relevant to it. When the information gets bulky (such as test pattern information), the information is only referenced in the appropriate test mode block (*CTLMode {}*) of the Environment. The referenced construct is defined outside the Environment.

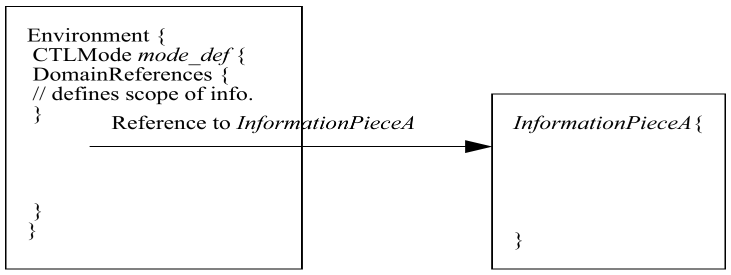


Figure 3—CTL information written outside the Environment and its relationship to the Environment

Figure 3 depicts the informational link between InformationPieceA and the test configuration of the design (*CTLMode*). InformationPieceA is a construct defined outside the Environment and is linked in to the test mode through a statement in the Environment. This mechanism is used for test patterns. The patterns are referenced in the Environment block of statements but defined outside the Environment. The following example shows the reference mechanism as it pertains to patterns:

[4]Notes in text, tables, and figures are given for information only and do not contain requirements needed to implement the standard.

```
Environment {
   CTLMode mymode {
     PatternInformation {
       // reference to pattern P1 as a pattern used to establish the
       // test mode named mymode.
       Pattern P1 { Purpose EstablishMode; Protocol Macro macro_name;}
     }
   }
}
Pattern P1 {
   // definition of pattern P1. All constructs used by P1
   // are required to be in the scope of mymode as defined by
   // DomainReferences.
}
```

The construct being referenced and its supporting constructs should be within the scope of the test mode. CTL uses the concept of local and global entities for grouped constructs defined outside the Environment. Container blocks such as MacroDefs, Procedures, ScanStructures, Timing, Variables, and SignalGroups are defined to allow for the definition of their associated entities. These container blocks can be named with a DOMAIN_NAME to create domains. Domained entities are brought into the scope of the test mode in the Environment block through a construct called DomainReferences. Containers with no domain names are considered to be global pieces of information. Once within the scope of the test mode, the entities within the containers can be referenced or used by referenced constructs as shown in Figure 3. The following example shows the reference mechanism just described:

```
// Nameless container block for signals
Signals {
// signals defined here (global for all modes)
}
// Nameless container block for Macros
MacroDefs {
// global macros
}
// Named container block for Macros
MacroDefs A {
// domain A's macros
}
// Named container block for Macros
MacroDefs B {
// domain B's macros
}
Environment {
// global environment
   CTLMode mymode {
     DomainReferences {
       MacroDefs A;
     }
     // Now macros defined in MacroDefs A can be referenced here as they
     // are in the scope of mymode. See reference mechanism defined
     // in the paragraphs before this example. Macros written in the
     // nameless MacroDefs block are also in the scope of mymode.
   }
}
```

This example shows a test mode named *mymode* identified by the CTLMode block within the Environment block of statements. (This will be described in the following text in this subclause.) The information of *mymode* uses the global Macros and the Macros in the MacroDefs domain *A*.

As mentioned, information in CTL is organized around a test mode (configuration) of the core being represented. Some of the information in each test mode pertains to the externally accessible signals of the core. These signals, defined in the signals block of statements (1450.0), give a point of reference for test-mode-related information. It should be noted that the Signals block is an unnamed block, and hence, its information is global and already included in every test mode.

Designs that are described in CTL typically support multiple configurations or test modes.

Figure 4 shows a design supporting multiple modes of operation. The design has nine input signals and three output signals. The design is implemented with two scan chains both of length 2. One scan chain goes across the boundary of the design capturing values on *signal a* and launching values on *signal b*. The boundary scan chain has a scan input BSI and a scan output BSO and is clocked by BCK. Its scan configuration is achieved by putting a logic-0 on the scan enable BSE. The second scan chain is an internal scan chain with scanin SI, scanout SO, scan-enable SE = logic-1, and clock CLK. The signals *y* and *z* are test mode signals that define the configuration of the design. In this case, *y=0*, *z=0* is the functional configuration of the design; *y=0*, *z=1* puts the design in a test mode where the boundary scan chain can be used to capture and launch values for the purposes of testing the logic outside the design when the design is embedded in a larger design. *y=1*, *z=0* sets the design in a test mode where the logic internal to the design can be tested.

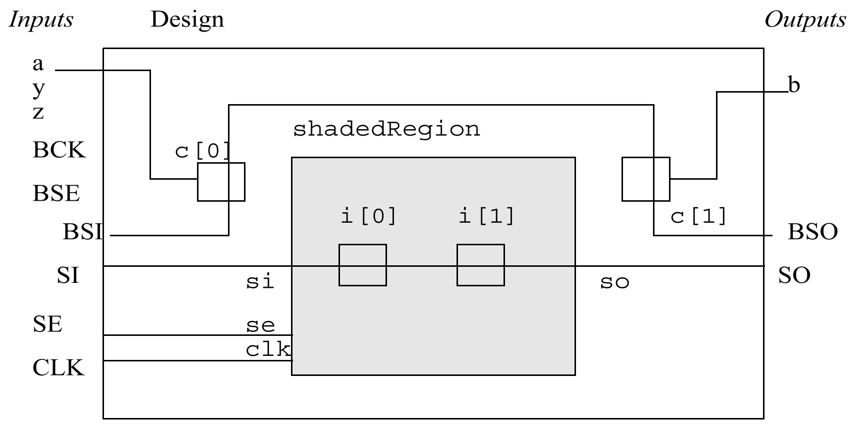


Figure 4—A design for which CTL is written to describe its test-related information

In this subclause on design configurations, the goal is to only describe the configurations of the design. That is, we are interested in only describing how to establish every test mode and not everything that we can describe about the design. The complete CTL-code for the associated modes of Figure 4 is shown as follows. **Bold words** highlight the keywords defined by the standards, which in this case are CTL (1450.6, 1450.0, 1450.1, and 1450.2) constructs. Comments and explanations are embedded in the example:

```
STIL 1.0 {
   Design 2005;
   CTL 2005;
}
```

```
// The information in the environment requires the recognition of the
// boundary of the design. This is described in the Signals block outside
// the environment. Since Signals is a nameless block the signals can
// be then referred to in the Environment. Signals in this example are
// either Inputs (In) or Outputs (Out). Details of Signals are defined
// in 1450.0. Each signal is assigned a default state to denote that if
// no state is specified for that signal on a specific pattern, the
// specified default state would be applied to that signal.
Signals {
   a In { DefaultState Z;}
   b Out;
   y In { DefaultState Z;}
   z In { DefaultState Z;}
   SE In { DefaultState Z;}
   BSE In { DefaultState Z;}
   CLK In { DefaultState D;}
   BCK In { DefaultState D;}
   SI In { DefaultState Z;}
   SO Out;
   BSI In { DefaultState Z;}
   BSO Out;
} // end Signals

// Values in CTL are waveform characters that are defined in the timing
// block as events (D, U, N, L, H, X) associated with timing within
// a clock period. The syntax specification of the Timing block can be
// found in 1450.0. Since the Timing block is a named block it needs to
// be explicitly brought into the scope of the test modes. This would
// be done through a statement in the DomainReferences. The timing
// block is needed to support the definition of the sequences that
// configure the design into the test mode. The design is configured
// into a test mode by a sequence of clock periods as defined by the
// Macros, and the values are waveform characters that are supplied by
// the patterns to the Macros.
Timing T1 {
   WaveformTable W1 {
     Period '100ns';
     Waveforms {
       'a+y+z+SE+BSE+CLK+BCK+SI+BSI' { 01x {'0ns' D/U/N;}}
       'b+SO+BSO' { 01x {'0ns' L/H/X;}}
     }
   }
} // end Timing

// The MacroDefs block represents the protocol or sequence information
// in CTL. The following is an unnamed MacroDefs block; hence, its macros
// are globally available in every test mode. CTL restrictions on
// pattern syntax as defined in 1450.6 forces the Macros to refer to the
// waveforms within it. The syntax and semantics used are defined
// in 1450.0. These macros defined here is needed to support the
// definition of the mechanism used to establish the test modes. The test
// mode is to be established by patterns that call the macros in this
// example.
MacroDefs {
   // Macro setupseq takes in two parameters y and z and assigns them to
   // the signals y and z. The following uses syntax from 1450.0
   setupseq {
     W W1;
     V { y=#; z=#; }
   }
} // end MacroDefs

// This is the primary block of information being written in CTL. The
// ordering rules of 1450.0 require language blocks to follow the rule
// that blocks must be defined before they are referenced (with some
// exceptions around pattern blocks). One named Environment block is
// expected to have information pertaining to the test needs of the
// design after inheritance of environments is resolved.
Environment design {

   // A nameless CTLMode block may be used to represent global
   // information that is common to all named CTLMode blocks.
   // In this example the Timing used in all the test modes is the same.
   CTLMode {

     // The named timing block is brought into the scope of the global
     // or nameless CTLMode block of statements. Hence it is also brought
     // into the scope of every other named CTLMode block of statements.
     // In this example the timing block is brought into the scope of the
     // test modes allowing for its information to be used by other
     // constructs in the test mode. As a result of the following
     // statement the Macro setupseq can use waveform table W1.
     DomainReferences {
       Timing T1;
     }
   } // end nameless CTLMode

   // One of the configurations to be described is the normal operation
   // of the design. This is achieved by setting y=0 and z=0. This is
   // achieved in CTLMode through a Pattern that establishes the test
   // mode. The pattern is defined outside the environment and referred
   // to in the PatternInformation block of this test mode. The pattern
   // is accordingly identified by an EstablishMode keyword. After the
   // test mode is established, the signals y and z are to be held
   // constant to maintain the configuration. This is defined in the
   // Internal block.
   CTLMode myN {

     // Test mode is used to define the functional operation of the
     // design, and the function is a non-test function.
     TestMode Normal;

     // The internal block of statements contains information on the
     // signals of the design looking inward. This block contains
     // information on the characteristics of the signals as they are
     // used in the test mode (myN in this case).
     Internal {
       // Signals y and z both are of type TestMode. That is they are
       // to remain constant after the configuration is established
       // until termination of the test mode. The establish mode
       // pattern would have left these signals at a logic-0 state.
       'y+z'{
         DataType TestMode {
           // Both signals y and z are to be held to a Logic-0
           ActiveState ForceDown;
         }
        } // end 'y+z'
       } // end Internal

       // Container for Pattern Related Information
       PatternInformation {
         // Look for a Pattern P1 outside the Environment. That Pattern
         // is used to Establish this Normal test mode. The Pattern would
         // be written with 1450.6's syntax that restricts patterns to
         // have calls to a sequence (Macro or Procedure) and pass
         // the necessary data to it. If it did not follow the CTL rules
         // for Patterns, it would be identified as a Foreign pattern.
         // Pattern P1 uses Macro setupseq. The complete execution of
         // P1 takes 1 clock period.
         Pattern P1 {
           Purpose EstablishMode;
           Protocol Macro setupseq;
           CycleCount 1;
         }
       } // end PatternInformation
   } // end CTLMode myN

   // The test mode that makes the boundary scan chain of the design
   // configured to be passively involved in the test of logic outside
   // the embedded design. y=0, z=1.
   CTLMode myE {
     // The design is setup to be outward facing.
     TestMode ExternalTest;

     // Information for the ExternalTest test mode on the signals
     Internal {
       // Signal y is expected to remain constant after the test mode
       // is established until termination of the test mode.
       // y is expected to remain at a logic-0. The patterns that
       // establish this configuration would have set y to a logic-0.
       y {
         DataType TestMode {
           ActiveState ForceDown;
         }
       }
       // Signal z is expected to remain constant after the test mode
       // is established and before the termination of the test mode.
       // z is expected to remain at a logic-1. The patterns that
       // establish this configuration would have set z to a logic-1.
       z {
         DataType TestMode {
```

```
        ActiveState ForceUp;
      }
    }
  } // end Internal

  // The container of Pattern information for the test mode.
  PatternInformation {
    // Look for Pattern P2 defined outside this environment. That
    // pattern follows the CTL rules and is labeled
    // as a pattern that is to be used to get the design into the
    // outward-facing test mode. Pattern P2 only uses Macro setupseq.
    // The complete execution of P2 takes 1 clock period.
    Pattern P2 {
      Purpose EstablishMode;
      Protocol Macro setupseq;
      CycleCount 1;
    }
  } // end PatternInformation
} // end CTLMode myE

// The test mode that defines the configuration that is to be used
// to test the design. This test mode would typically also include
// test patterns. However, for this example, we do not require them
// to be defined. This configuration requires y=1 and z=0.
CTLMode myI {
  // the test mode to test the internals of this design.
  TestMode InternalTest;

  // the container for information on signals.
  Internal {

    // y is expected to remain constant logic-1 after the test mode
    // has been established to maintain the configuration.
    y {
      DataType TestMode {
        ActiveState ForceUp;
      }
    }
    // z is expected to remain constant logic-0 after the test mode
    // has been established to maintain the configuration.
    z {
      DataType TestMode {
        ActiveState ForceDown;
      }
    }
  } // end Internal

  // the container for Pattern information
  PatternInformation {

    // Look for Pattern P3 defined outside this environment. That
    // pattern follows the CTL rules and is labeled
    // as a pattern that is to be used to get the design into the
    // outward-facing test mode. Pattern P3 only uses Macro setupseq.
```

```
        // The complete execution of P3 takes 1 clock period.
        Pattern P3 {
          Purpose EstablishMode;
          Protocol Macro setupseq;
          CycleCount 1;
        }
      } // end PatternInformation
   }// end CTLMode myI
} // end Environment design

// Definition of Patterns P1, P2, and P3. Note that the pattern follows
// the definition defined in this document. That is, Patterns must only
// call sequences (Protocols—Macros and Procedures). Patterns
// pass data to the sequences. Patterns must call only one type of
// sequence, perhaps multiple times, as limited by the syntax. The name
// of the sequence is defined by the Protocol statement in the invoking
// PatternBurst or the PatternInformation block of statements. In this
// example there is no invoking PatternBurst; hence, the Protocol
// statement must exist in the PatternInformation. In this example the
// Protocol invoked is a Macro called setupseq for all patterns.

// Note that these patterns are not included in any PatternExec or
// PatternBurst. This is legal CTL. The information did not need the
// PatternExec and PatternBurst constructs. In this example P1, P2, and
// P3 are used to define the patterns that are to be executed to
// establish the test mode.
Pattern P1 {
   P { y=0; z=0;}
}
Pattern P2 {
   P {y=0; z=1;}
}
Pattern P3 {
   P { y=1; z=0;}
}
```

Let us step through the thinking process that led to writing the above CTL. We need to describe test modes for the design. Thus, we need an Environment block of statements (Environment {}) with three CTLMode blocks of statements one for every test mode. Each CTLMode block is labeled with the type of configuration it falls under with a TestMode statement. Each test mode has an initialization sequence that sets *y* and *z* to certain values. This would be written by creating a reference to an initialization pattern in the appropriate CTLMode block and writing the patterns outside. The patterns rely on infrastructure such as Macros, Timing, and Signals. After establishing the test mode, some signals need to remain constant. This is indicated with the DataType-ActiveState statement. To write this statement, the Signals need to be defined in the Signals block of statements. All constructs written outside the CTLMode blocks, if referred by this block, should be in the scope of the CTLMode block of statements. This is done by the DomainReferences statement. Common information across all modes can be collected in the nameless CTLMode block of statements.

Now let us step through the example in the reverse direction and see if what is described says what we expect it to say. Start with the Environment. The Environment has one common CTLMode block and three CTLMode blocks for each test mode defined. Every test mode has some sequence defined to establish its configuration. The test modes are established by executing patterns that are identified in their appropriate

PatternInformation blocks. The configurations are maintained by holding signals *y* and *z* to their appropriate values as specified by the ActiveStates of the TestMode data types.

Thus, in this example, the party that created the CTL and the party that interprets the CTL would have the same information about the design.

4.3 CTL for structural information

We use the same design of the previous example to show how scan chains and a subdesign are represented in CTL.

In this subclause, the two scan chains of the example design are described in CTL. One scan chain is a boundary scan chain, and the other scan chain is an internal scan chain of the design. A subdesign (the shaded portion of the design) as an embedded hierarchy is recognized, and the relationship between scan cells and the hierarchy is maintained. The example describes the fact that the subdesign comes with a single scan chain with two scan cells and its construction as part of the scan chains of the complete design.

Although the scan chains may be available in some modes and not others, in this example, the information is made test mode independent:

```
STIL 1.0 {
   Design 2005;
   CTL 2005;
}

// The information requires the recognition of the
// boundary of the design. This is described in the Signals block outside
// the environment. This is the same as that defined in the previous
// example.
Signals {
   a In { DefaultState Z;}
   b Out;
   y In { DefaultState Z;}
   z In { DefaultState Z;}
   SE In { DefaultState Z;}
   BSE In { DefaultState Z;}
   CLK In { DefaultState D;}
   BCK In { DefaultState D;}
   SI In { DefaultState Z; ScanIn 2; }
   SO Out { ScanOut 2; }
   BSI In { DefaultState Z; ScanIn 2;}
   BSO Out { ScanOut 2;}
} // end Signals

// A hierarchy is defined for the shaded area. This hierarchy is
// recognized to have some inputs and outputs and a scan chain. The
// scan chain is of length 2 with scan cells i[0..1] (this expands
// to be a list of cells i[0], i[1]). The first cell in the list i[0] is
// closer to si and i[1] is closer to so. The syntax for CoreType is
// defined in this document. However, it reuses the syntax for Signals
// and ScanStructures from 1450.0 and 1450.1.
CoreType subDesign {
   Signals {
     si **In**;
     so Out;
     se In;
     clk In;
   }
   ScanStructures {
     ScanChain twoCellChain {
       ScanLength 2;
       ScanIn si;
       ScanOut so;
       ScanCells i[0..1];
       ScanEnable se;
       ScanMasterClock clk;
     }
   }
} // end CoreType subDesign

// A single instance of the design is defined. This instance is called
// shadedRegion (see Figure 4). As a result, the signals
// of shadedRegion are shadedRegion:si, shadedRegion:so, shadedRegion:se
// and shadedRegion:clk. The scan cells of this hierarchy are hence
// shadedRegion:i[0..1]. If a domain name was used for the scan
// structures in the CoreType definition, the name of the scan cells
// would include the domain-name from the scan structures to look like:
// CORE_INSTANCE_NAME:DOMAIN_NAME::SCAN_CELL_NAMES
CoreInstance subDesign {
   shadedRegion;
}

// The scan chains of the design are defined in the ScanStructures. In
// this example, one of the scan chains is constructed out of scan cells
// of the embedded hierarchy. The syntax used here is defined in 1450.0
// and 1450.1.
ScanStructures {
   ScanChain bc1 {
     ScanIn BSI;
     ScanOut BSO;
     ScanLength 2;
     ScanEnable ~BSE;
     ScanCells c[0..1];
     ScanMasterClock BCK;
   }
   ScanChain ic1 {
     ScanIn SI;
     ScanOut SO;
     ScanLength 2;
     ScanCells shadedRegion:i[0..1];
     ScanEnable SE;
     ScanMasterClock CLK;
   }
} // end nameless ScanStructures

// Values in CTL are waveform characters that are defined in the timing
// block as events (D, U, N, L, H, X) associated with timing within
// a clock period. Details of the Timing block should be obtained from
// 1450.0. Since the timing block is a named block it needs to be
// explicitly brought into the scope of the test modes. This would
// be done through a statement in the DomainReferences. The timing
// block is needed to support the definition of the sequences that
// configure the design into the test mode. The design is configured
// into a test mode by a sequence of clock periods as defined by the
// Macros and the values are waveform characters that are supplied by
// the patterns to the Macros.
Timing T1 {
   WaveformTable W2 {
     Period '100ns';
     Waveforms {
       'a+y+z+SE+BSE+CLK+BCK+SI+BSI' { 01x {'0ns' D/U/N;}}
       'CLK+BCK' { P {'0ns' D; '50ns' U; '60ns' D;}}
       'b+SO+BSO' { 01x {'0ns' L/H/X;}}
     }
   }
} // end Timing

// Some protocols take in parameters. These parameters can be provided
// as values on signals, signal groups, or variables. In this example
// the variables used are defined below. Note these variables are defined
// in a nameless Variables block. Thus, the variables are global
// across the information in every test mode. If the Variables had a
// domain name associated with it, then the DomainReferences block of
// statements would have to be used in the CTLMode{}.
Variables {
   SignalVariable bsivals[1..0];
   SignalVariable bsovals[1..0];
   SignalVariable sivals[1..0];
   SignalVariable sovals[1..0];
}

// All protocols that need to be described are defined in the MacroDefs
// blocks. In this example there are four protocols that are written in
// an unnamed MacroDefs block (global to all test modes).
MacroDefs {
   // scanbsi takes in variables and applies it as a shift operation to
   // BSI and BSO. The shift register gets bsivals[1..0] and
   // simultaneously bsovals[1..0] are observed at BSO.
   scanbsi {
     W W2;
     C {BSE=0; BCK=0; bsivals=#; bsovals=#;}
     Shift { V { BSI= \W bsivals[1..0]; BSO=\W bsovals[1..0]; BCK=P;}}
   }
   // scansi is similar to scanbsi but defines the sequence through which
   // values in variables sivals[1..0] are applied to scan cells through
   // SI, and values in variables sovals[1..0] are observed from scan
   // cells through signal SO.
   scansi {
     W W2;
     C {SE=1; CLK=0; sivals=#; sovals=#;}
     Shift { V { SI=\W sivals[1..0]; SO= \W sovals[1..0]; CLK=P;}}
   }
   // MCaptureBCK is a sequence that pulses the BCK clock to capture
   // values through the functional path (BSE=1).
   MCaptureBCK {
     W W2;
     V { BSE=1; BCK=P;}
   }
   // MObserveConnection is a sequence that allows for data to be
   // captured and observed through the BSO signal.
   MObserveConnection {
     W W2;
     Macro MCaptureBCK;
     Macro scanbsi;
   }
} // end nameless MacroDefs

// The block of statements that contains the primary information in CTL.
// If one was reading this example, they should start from here.
Environment design {
   // A nameless CTLMode block of statements carries test mode-
   // independent information.
   CTLMode {

     // The named domain for timing information needs to be brought into
     // the scope of the information in this example. If this statement
     // is missing, no information in the scope of this CTLMode block of
     // statements should refer to waveforms in T1. If this statement
     // is missing in this example, an error would occur when the Macros
     // of this CTLMode {} are to be interpreted.
     DomainReferences {
       Timing T1;
     }

     // A container for information on the signals of the design.
     Internal {

       // Information on signal "a" is to follow. Another block of signal
       // "a" cannot occur in this CTLMode{}. Signal "a" can be part of
       // another named group in this CTLMode{}. The same is true for all
       // other named signals and signal-group names in this CTLMode{}.
       a {

         // Values on signal a are captured in scan cell c[0]. The
         // clock that captures values in the state element c[0] is BCK.
         // Values are captured on the LeadingEdge of BCK, and c[0] takes
         // on a value from the Connection when this event occurs. To
         // observe values through the connection, one would have to
         // execute the Macro MObserveConnection. MObserveConnection
         // should be in the scope of information available in this
         // CTLMode{}.
         IsConnected In {
           StateElement Scan c[0];
           CaptureClock BCK {
             LeadingEdge; StateAfterEvent Connection;
```

```
      }
      TestAccess Observe Macro MObserveConnection;
    }
  }
  b { // information on b.

    // This is similar to the connection described on signal a.
    // In this case signal b is on the receiving end of the
    // connection, and it receives values from scan cell c[1].
    // The clock that affects the values in c[1] is BCK. After
    // the leading edge of BCK, c[1] would go to an Unknown state.
    // Values are put into c[1] by executing the macro scanbsi that
    // is in the scope of the current CTLMode{}.
    IsConnected Out {
      StateElement Scan c[1];
      LaunchClock BCK {
        LeadingEdge; StateAfterEvent ExpectUnknown;
      }
      TestAccess Control Macro scanbsi;
    }
  }
  BSE { // information on BSE
    // BSE is the test control signal. Specifically it is a
    // ScanEnable that enables the scan configurations with a
    // logic-0.
    DataType TestControl ScanEnable { ActiveState ForceDown; }
  }
  SE { // information on SE
    // SE is a test control signal. Specifically it is a ScanEnable
    // that enables the scan configuration with a logic-1.
    DataType TestControl ScanEnable { ActiveState ForceUp; }
  }
  'BCK+CLK' { // information on BCK and CLK
    // both BCK and CLK are test control signals. They are clocks
    // used for Scan Operations and Capture operations. All
    // sequences in this CTLMode {} assume that these clocks are
    // at a logic-0 at the beginning of every protocol.
    DataType TestControl CaptureClock ScanMasterClock
      { AssumedInitialState ForceDown; }
  }
  BSI { // information on BSI
    // BSI is a scan input. Since no explicit connection is defined
    // using the IsConnected statement, the connection from
    // the BSI to the first scan cell can be obtained from the
    // scan chain definition.
    // BSI is connected internally in the design as a scan-in
    // where the connection is defined in the Scan chains in the
    // scope of this CTLMode {}. Looking at scan chain bc1, BSI is
    // connected to scan cell c[0]. The connection is enabled by
    // setting BSE to a logic-0.
    // BSI is of the general type TestData and specifically receives
    // scan data values. BSI is the scan input of the boundary scan
    // chain.
    DataType TestData ScanDataIn { ScanDataType Boundary; }
```

```
        }
        BSO { // information on BSO
          // Similar to BSI, BSO's connection is defined by the scan
          // chain. BSO is connected internally to the design to c[1]
          // where BSO is at the receiving end of the connection. There
          // is no gating logic between c[1] and BSO as no Enabling
          // condition exists for the connection. BSO is of type test
          // data and is a scan-out for a boundary scan chain of the
          // design.
          DataType TestData ScanDataOut { ScanDataType Boundary; }
        }
        // Similar to BSI and BSO, the SI and SO are defined as scan
        // signals that are used for internal scan chains of the design.
        SI { DataType TestData ScanDataIn { ScanDataType Internal; }}
        SO { DataType TestData ScanDataOut { ScanDataType Internal; }}
      } // end Internal
      PatternInformation {

        // If one needs to operate scan chains in this configuration,
        // scanbsi can be used to control and observe values in scan chain
        // bc1 and scansi can be used to control and observe values in
        // scan chain ic1.
        Macro scanbsi {
          Purpose ControlObserve; ScanChain bc1;
        }
        Macro scansi {
          Purpose ControlObserve; ScanChain ic1;
        }
      } // end PatternInformation
    } // end CTLMode
} // end Environment
```

Several things that needed to be described in this example were associated with the scan chains of the design. The information is provided through several statements. Scan chains can be defined only in the scan structures. Connections from the boundary of the design to the scan cells are defined through the IsConnected statement in the environment. Various aspects about the signals and the connections are embedded along with the connection information. From the information there should be no doubt about the relationship between scan cells and the hierarchy called *shadedRegion*.

The reader should perform a similar exercise to that of the previous example and see that the constructs used are the only mechanisms in place for the information that can be described. Reading the CTL should give the reader a crisp view of the information that was described in the paragraph before the example.

4.4 CTL for test pattern information

This example uses the same design as the previous examples.

The intent of this example is to describe test patterns in CTL. The design is to be configured in its internal test mode such that the test patterns can be applied for testing the manufactured product. The details of getting into the internal test configuration are the same as described in the example that described test modes for the design. That is, signals *y* and *z* of the design are to be set to logic-1 and logic-0, respectively. The test patterns are a collection of stimulus and responses that are applied on a cycle-by-cycle basis to the inputs and outputs of the design. The design has 100 stuckat faults when every gate of the design is faulted. Of these faults, 99 are detected by the tests. Just to show a feature of CTL that would not be highlighted otherwise, the information requires the event during which the test patterns capture values through the functional path of the design be highlighted in the information.

In the explanations of this example, statements will be made to highlight certain features of the test patterns relating to the restrictions CTL puts on STIL syntax.

```
STIL 1.0 {
   Design 2005;
   CTL 2005;
}

// The information requires the recognition of the
// boundary of the design. This is described in the Signals block outside
// the environment. This is the same as that defined in the previous
// example.
Signals {
   a In { DefaultState Z;}
   b Out;
   y In { DefaultState Z;}
   z In { DefaultState Z;}
   SE In { DefaultState Z;}
   BSE In { DefaultState Z;}
   CLK In { DefaultState D;}
   BCK In { DefaultState D;}
   SI In { DefaultState Z; ScanIn 2;}
   SO Out { ScanOut 2; }
   BSI In { DefaultState Z; ScanIn 2;}
   BSO Out { ScanOut 2; }
} // end Signals

// Some signal groups are defined for the design for ease of use in
// writing the patterns.
SignalGroups {
   Ins[0..3]= 'BSE+SE+SI+BSI';
   Clocks[0..1] = 'BCK+CLK';
   Enables[0..1]= 'BSE+SE';
   Outs[0..1] = 'SO+BSO';
} // end SignalGroups

// Some variables are defined for the parameters of the Macros that are
// written for the test patterns.
Variables {
   SignalVariable aval; // to carry only a single value for signal a.
   SignalVariable bval; // to carry only a single value for signal b.
   SignalVariable sivals[1..0]; // to carry stimulus for the scan cells
   SignalVariable sovals[1..0]; // to carry response data for scan cells
} // end Variables
```

```
// Some default timing data that are referenced by Macros. This is the
// same as those used in the previous examples. For details refer to
// 1450.0 and 1450.1.
Timing T1 {
   WaveformTable W2 {
     Period '100ns';
     Waveforms {
       Clocks[0..1] {0P{'0ns' D/D; '60ns' D/U; '65ns' D/D;}}
       Ins[0..3] { 01x {'0ns' D/U/N;}}
       Outs[0..1] { 01x {'0ns' L/H/X;}}
     } // end Waveforms
   } // end WaveformTable W2
   WaveformTable W1 {
     Period '100ns';
     Waveforms {
       'a+y+z+SE+BSE+CLK+BCK+SI+BSI' { 01x {'0ns' D/U/N;}}
       'b+SO+BSO' { 01x {'0ns' L/H/X;}}
     } // end Waveforms
   } // end WaveformTable W1
} // end Timing T1

// The Macros that are written to support the patterns. CTL syntax
// disallows waveform information in patterns. CTL requires the
// data portion of the tests to be in the patterns and the
// protocols contain the sequencing information. With this restriction
// the Macros would have to define the Waveform tables used. Two macros
// are written in this example, one to support the test pattern used
// to detect the faults and one to support the pattern (seq) that
// is written to establish the internal test mode of the design
// (setupseq).
MacroDefs forInternalTest {

   // seq is a sequence that takes in the following parameters.
   // aval, sivals[1..0], bval and sovals[1..0]
   // This sequence first sets up the scan configuration and
   // shifts in aval into the boundary scan chain and sivals[1..0]
   // into the internal chain of shadedRegion. Note this information
   // does not require the definition of Scan Cells. Then a labeled
   // statement exists where the clocks are being pulsed. Finally,
   // the scan chains are configured to shift and the values in the
   // wrapper chain and internal chain are observed.
   seq {
     W W2;
     C {aval=#; sivals=#; bval=#; sovals=#;}
     C {Enables[0..1]=01; Clocks[0..1]=00; Ins[2..3]=xx;}
     Shift {V {BSI= \W aval; SI= \W sivals[1..0]; Clocks[0..1]=PP;}}
     mylabel: V { Enables[0..1]=10; Clocks[0..1]=PP;}
     V { Enables[0..1]=01; Clocks[0..1]=00;}
     Shift {V{BSO=\W bval; SO=\W sovals[1..0]; Clocks[0..1]=PP;}}
   } // end seq

   // This macro is used to apply values to y and z.
   setupseq {
     W W1;
     V { y=#; z=#;}
   } // end setupseq
} // end MacroDefs forInternalTest

// The order in which the Patterns will be applied as it relates to the
// patterns that is defined to establish the test mode. This is the
// schedule of the patterns.
PatternBurst pats {
   // The following MacroDefs must be in the DomainReferences of the
   // associated CTLMode block that invokes the pattern burst.
   MacroDefs forInternalTest;
   PatList {
     // If a Protocol statement exists in the PatternInformation for P3
     // and all_pats the information must be consistent.
     P3 {Protocol Macro setupseq;}
     all_pats {Protocol Macro seq;}
   }
} // end PatternBurst pats

// The pattern exec that contains the test patterns for the design.
PatternExec topPat {
   Timing T1;
   PatternBurst pats;
} // end PatternExec topPat

// The primary block of information that is being written for this
// example. A test mode is defined to be of type internal test. The
// patterns that relate to the test mode are referred to in the test mode
// but the patterns are defined outside. Since CTL follows the same
// rules for "define before referencing" as STIL, the pattern-exec
// and Pattern-Burst blocks are written before the Environment block
// of statements. The patterns are labeled as Production patterns,
// and the fault coverage is specified in the environment. The Macro
// is identified to be used to apply non-overlapped test patterns
// in this test mode. The statement with a label is identified to
// be the time when values are Captured into the memory elements
// through the functional paths.
Environment design {
   // Internal test mode of the design that has its patterns and
   // establishes the test mode sequence. Details of y and z remaining
   // constant for the validity of the test mode are not required
   // in the information that needs to be specified. However, if it
   // does need to be specified, the DataType syntax would have to
   // be used from the first example.
   CTLMode myI {
     // this test mode allows for the internals of the design to be
     // tested.
     TestMode InternalTest;

     // the named domains need to be brought into the scope of the
     // information in this test mode.
     DomainReferences {
       Timing T1;
       MacroDefs forInternalTest;
```

```
      }

      // All the patterns to be applied in the test mode are identified
      // in the PatternInformation. There could have been many more
      // patterns written outside, and only some of them are relevant
      // to the test mode.
      PatternInformation {

        // Of all the pattern execs, topPat contains the patterns that
        // are to be used for production testing. Within PatternExec,
        // of all the bursts that may exist, only pats contains all
        // the test patterns. The fault coverage of the tests is
        // 99% single-stuck at coverage of faults.
        PatternExec topPat {
          Purpose Production;
          PatternBurst pats;
          Fault {
            Type StuckAt UnCollapsed;
            FaultCount 100;
            FaultsDetected 99;
          }
        }

        // Macro seq is identified to be used in this test mode to
        // apply test pattern sequences that are not overlapped. mylabel
        // in seq is a Capture event that occurs during the clock period
        // associated with the statement.
        Macro seq {
          // also could have used forInternalTest::seq
          // Since there is no ambiguity in using the shorter name seq,
          // both names resolve to the same Macro. seq is the only macro
          // with that name in all the named MacroDefs blocks active in
          // the current test mode.
          Purpose DoTest;
          Identifiers {
            EventType Capture {
              Label myLabel { Complete; During;}
            }
          }
        }

        // Pattern P3 is used to establish the test mode. It uses macro
        // setupseq. Note that setupseq is unique in the active
        // MacroDefs. If it was not unique, the macro name would have been
        // forInternalTest::setupseq
        Pattern P3 { Purpose EstablishMode; Protocol Macro setupseq;}

        // Pattern all_pats applies scan patterns and uses macro seq.
        Pattern all_pats { Purpose Scan; Protocol Macro seq; }
      } // end PatternInformation
    } // end CTLMode myI
} // end Environment design

// Definition of pattern P3. It supplies data to the Macro setupseq.
```

```
Pattern P3 {
   P {y=1; z=0;}
}

// Definition of pattern all_pats. It supplies data through variables
// in this example. Note patterns only call protocols and supply the
// data to the protocols. CTL syntax does not allow for V statements
// in Patterns. In this example all_pats is defined to use a Protocol
// that is defined as a Macro named seq.
Pattern all_pats {
   P {aval=1; sivals[1..0]=10; bval=0; sovals[1..0]=11;}
   P {aval=1; sivals[1..0]=10; bval=0; sovals[1..0]=11;}
   // many more calls to Macro seq (see Protocol statement for name).
} // end all_pats
```

The example shows test patterns that are provided in CTL. Although no information was provided on Macro setupseq in this example, it could have been referenced in the PatternInformation block as a sequence with a ModeControl purpose and to be used by patterns that establish the test mode.

4.5 Beyond the examples

The examples describe a portion of the constructs in CTL that could be used to convey information to the core user. The keywords in the examples are a limited subset of the keywords available in the language. CTL can be used to describe digital information for many designs and different test methodologies. Combined with the number of use models for the CTL information, it is impossible to describe every scenario. With the examples described, the reader is expected to understand the basic mechanism behind writing CTL. With that understanding and the definitions of the statements allowed, the reader should be able to describe test information for digital ICs.

The reader should be aware that this standard has restricted STIL in certain ways by defining extensions to 1450.0 and 1450.1. As a result, CTL patterns are different from patterns that are written using just the syntax defined in IEEE Std 1450-1999, IEEE Std 1450.1-2005, and IEEE Std 1450.2-2002. The syntax and restrictions on patterns defined in this standard are very important for the reusability of test patterns in SoC flows where test patterns are mapped from the core boundary to the SoC boundary. The mechanism supported by CTL allows for protocols to be changed or replaced without having access to or modifying the bulk of the test pattern data, which lies in the Pattern blocks. The reader should also be sensitive to a basic difference between STIL (1450.0, 1450.1, and 1450.2 syntax) and CTL (1450.6, 1450.0, 1450.1, and 1450.2 syntax) that exists:

— In STIL the Environment block is a user defined block that provides some more information about the patterns. In CTL the Environment is the primary block of information, it utilizes constructs outside its environment to complete its information that is partitioned across test modes.
— CTL restricts the grouping of signals and variables to allow for explicitness and bit-indexing capabilities when partial load and unload operations are performed on the scan chains.
— CTL restricts pattern constructs to separate the data and protocol portions of patterns for reusability of tests.
— CTL understands the concept of hierarchy through core-instances.

During the design of CTL, care has been taken to define a single mechanism to describe any given construct. For example, scan cells can only be defined in ScanStructures. Hierarchy can only be represented in CoreTypes and CoreInstances. However, depending on the information to be described, the scan chains can be written to show or not show the hierarchy. The choices will lie in the user flows being constructed.

Similarly the informational needs may spread the information to be described in single or multiple CTL test modes (CTLMode blocks). However, there is only one way to describe a DataType of a signal.

5. Extensions to IEEE Std 1450-1999 and IEEE Std 1450.1-2005

5.1 STIL name spaces and name resolution

The special characters ":" and "::" as defined in 1450.1 are used to concatenate two user-defined names that follow 6.8 of IEEE Std 1450-1999. With this addition, any entity within a domain can optionally be identified across all constructs in CTL (inside or outside the Environment) as D::E, where D is the domain name and E is the entity name. The prior mechanisms allowed by IEEE Std 1450-1999 for identifying the entities without the "::" as defined in 6.16 of IEEE Std 1450-1999 remain when no ambiguity exists in the identification of the entity. Similarly, with this addition, the ":" operator can be used to concatenate the core instance name to the entity name (E or D::E) to identify entities of the core instance. The names in this case take on the form C:E or C:D::E. In each case, C, D, and E are required to follow the naming conventions defined in 6.8 of IEEE Std 1450-1999. The fully expanded name of an entity is required to be unique across all entities of the same type.

Thus, this extension allows for a scan cell name c1 that is defined in CoreInstance i1 to be used as part of the scan chain definition of a ScanStructures outside the Core. This cell would be identified as i1:c1 when used in a ScanStructure as defined by IEEE Std 1450-1999 or IEEE Std 1450.1-2005. This extension allows any signal groupname used in a pattern to be referenced along with its domain name with the "::" separator.

5.2 Optional statements of IEEE Std 1450-1999

IEEE Std 1450-1999 is not specific about the optional and required blocks. CTL's usage of the constructs requires that all constructs be optional. The statements as identified in Table 7 and Table 8 of IEEE Std 1450-1999 are all optional in CTL. The statements are used when required to support the informational needs of the CTL being written. When the statements are used the define before use ordering as required by STIL still remains for these statements (also see 9.3 of IEEE Std 1450-1999).

Although all statements in CTL are optional, it is very difficult to write a typical CTL file without a Signals block and the Environment block. Furthermore, test patterns that are targeted for execution require the existance of the PatternExec statement and a PatternBurst statement. Thus, there is a requirement that if a Pattern statement exists outside of the EstablishMode or TerminateMode patterns in the CTL mode being described, then a PatternExec and PatternBurst must exist that invoke that Pattern.

5.3 Restricting the usage of SignalGroup and variable names

Values that are passed into Macros and Procedures could be SignalGroups or Variables. When the non-bussed representation (without []) is used to provide scan data, the data are assumed to represent a complete shift operation (full operation of the scan chain). When partial scan operations are performed, the syntax for the parameters (signal-group or variable) is required to use the bussed naming convention in the protocols (Macros and Procedures) to explicitly define the number of shift operations in the load–unload of the scan chain.

5.4 Additional reserved words

Table 1 lists all STIL reserved words defined by this standard and not defined in IEEE Std 1450-1999 or IEEE Std 1450.1-2005. Subsequent clauses in this standard identify the use and context of each of these additional reserved words.

Table 1—Additions to STIL reserved words

CTL, CoreType, CoreInstance
Protocol
Setup

5.5 STIL statement—extensions to IEEE Std 1450-1999, Clause 8

5.5.1 General

The STIL statement as defined in IEEE Std 1450-1999 identifies the primary version of IEEE Std 1450-1999 information contained in a STIL file and the presence of one or more standard Extension constructs.

All other constructs and restrictions for IEEE Std 1450-1999, Clause 8 are in effect here.

5.5.2 STIL syntax

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

5.5.3 STIL syntax description

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

IEEE_1450_0_IDENTIFIER: Defined to be 1.0. The primary version of STIL, as identified by IEEE Std 1450-1999.

(2) EXT_NAME: The specific name of the Extension. This syntax of this standard is identified by the name **CTL**. Although the CTL concept and mechanisms include 1450.1, the associated ext_name for 1450.1 (namely Design) should be used to indicate use of syntax defined by 1450.1. Refer to IEEE Std 1450.1-2005 for details.

EXT_VERSION: The primary version of an EXT_NAME. This version of the standard is identified by the value **2005**.

5.5.4 STIL syntax example

```
STIL 1.0 {
   Design 2005;
   CTL 2005;
}
```

5.6 Extensions to IEEE Std 1450-1999, 17.1 and 23.1

5.6.1 General

The pattern-statements allowed within the Pattern block are limited to procedure calls and macro calls (defined in Procedures and MacroDefs) and a subset of STIL statements that do not impact the cycle-count, the predictability of when the protocols are invoked by the pattern, and the ability to change the way the patterns are applied without changing the data portion of the patterns. Thus, the following statements are allowed from IEEE Std 1450-1999:

a) Call and Macro statements
b) Loop statement
c) Goto statement
d) BreakPoint statement

The following statement is allowed from IEEE Std 1450.1-2005:

e) Xref statement

5.6.2 Pattern block syntax

To support the syntax in the patterns, a supporting Protocol statement is added outside the Pattern, in the PatternBurst, and the corresponding location in the PatternInformation in cases where PatternBursts do not exist. The following syntax is relevant to 1450.6. In the *PatternBurst* syntax, two new statements are added to the existing syntax defined by 1450.0 and 1450.1. The *Pattern* syntax from 1450.0 or 1450.1 that does not appear below is not allowed in CTL as it does not support the data-protocol separation mechanism:

```
// The following syntax of PatternBurst defines incremental statements
// to the existing statements defined by 1450.0 and 1450.1
PatternBurst PAT_BURST_NAME {
   (Protocol <Macro MACRONAME (SETUP_MACRONAME)                          (3)
             | Procedure PROCNAME (SETUP_PROCNAME)>;)
   (<PatList | PatSet |
     ParallelPatList (SyncStart | Independent | LockStep)> {
        (PAT_NAME_OR_BURST_NAME {
          (Protocol <Macro MACRONAME (SETUP_MACRONAME)                   (4)
                    | Procedure PROCNAME (SETUP_PROCNAME)>;)
        })*
   })+
}
```

The following syntax reflects the only syntax allowed in Patterns. Patterns are allowed to be defined either with the P-syntax or with Macros and Procedures. A single Pattern cannot use both mechanisms:

```
Pattern PATTERN_NAME {
   ( Setup { (sigref_expr = value_variable_expr;)* })                    (5)
   <((LABEL:) P;)* |                                                     (6)
   ((LABEL:) P { (sigref_expr = value_variable_expr;)*})*>               (7)
   // Pattern-statements that are allowed from the ones defined in
   // 1450.0 and 1450.1
   ((LABEL:) Loop LOOPCNT { (P-statements)* })*
   ((LABEL:) Goto LABELNAME;)*
   ((LABEL:) BreakPoint;)*
   ((LABEL:) X TAG;)*
}

// OR

Pattern PATTERN_NAME {
   // Pattern-statements that are allowed from the ones defined in
   // 1450.0 and 1450.1. For definitions of the statements,
   // refer to the approprite standard where they are defined.
   ((LABEL:) Macro MACRONAME;)*                                             (8)
   ((LABEL:) Macro MACRONAME { (sigref_expr = value_variable_expr;)*})*     (9)
   ((LABEL:) Call PROCNAME;)*                                               (10)
   ((LABEL:) Call PROCNAME { (sigref_expr = value_variable_expr;)*})*       (11)
   ((LABEL:) Loop LOOPCNT { (Macro-or-Procedure-calls)* })*                 (12)
   ((LABEL:) Goto LABELNAME;)*                                              (13)
   ((LABEL:) BreakPoint;)*                                                  (14)
   ((LABEL:) X TAG;)*                                                       (15)
}
```

5.6.3 Pattern block—syntax descriptions

Two types of Patterns are allowed, as follows:

a) Patterns using the P-statement require the use of the Protocol statement to define the Macro or Procedure being invoked. Patterns limited to using the P-statement allow for maximum test pattern reusability. Patterns provided in this format can be synchronized with other patterns even when there are dependencies between patterns. The protocols used by the patterns are retargetable and usable with the ParallelPatList-LockStep construct. The data are to be specified by the patterns, and the protocol is to be specified by the Macros and Procedures. Any sequencing within a single test pattern unit (for example, an ATPG test pattern for a stuck-at fault) is to be in a single protocol. The protocol can be subdivided into smaller protocols, with Macros or Procedures calling other Macros and Procedures. As limited by the syntax, Patterns are allowed to call only a single protocol multiple times and supply different values for the parameters that represent the data portion of the patterns. This restriction prevents sequencing of the protocols in the patterns to occur, which assists in CTL's need to support pattern reuse without modifying the patterns or looking at details inside the patterns. That is, outside of the setup sequence, a pattern cannot call a protocol that executes a sequence and call another protocol to execute another sequence. *For examples of this syntax,* refer to 4.4.
b) Patterns are limited to calling Macros and Procedures. However, no restrictions are created for the partitioning of the sequencing information across the Macros and Procedures. Pattern reuse relies on changing the protocol invoked by the Patterns. Restricting patterns to call Macros and Procedures allows for this capability. ParallelPatList-LockStep requires synchronizing patterns that cannot be performed if the patterns do not follow the restrictions of Patterns provided with the P-statement. As this method of representing patterns separates data and protocols, albeit limited, pattern reuse can still be performed by replacing the protocols. In this format, without the ability to rewrite the complete pattern data, all test pattern reuse tasks are limited to changing the sequencing activities within each protocol. As this syntax represents restricted STIL, the *example explanations* in IEEE Std 1450-1999 and IEEE Std 1450.1-2005 should be sufficient.

Test patterns represented with P-statements is the preferred method for maximum flexibility for test pattern reuse.

Protocols in CTL are written as Macros and Procedures. Patterns using statements (5), (6), and (7) are restricted to calling only a single protocol outside of the first invocation, which is the setup protocol.

Because of this restriction, the calls to Procedures and Macros in Patterns can be prespecified in the PatternBurst syntax. Statement (3) and (4) identify the protocol used by the patterns that are being invoked. The protocol could be either a Macro or a Procedure, and the appropriate name follows in the MACRONAME or PROCNAME. The syntax [statements (3) and (4)] shown highlights the incremental constructs allowed in CTL for the pattern bursts in addition to the ones defined in IEEE Std 1450-1999 and IEEE Std 1450.1-2005. The Protocol statement is not allowed to be used for Patterns that do not have the P-statement. An optional Setup statement is allowed in a Pattern. If this statement occurs, it is expected to be the first statement within the Pattern. The setup statement is required to call the same type of protocols (Macro or Procedure) as that used by the P-statements that follow. The Setup pattern calls a macro or procedure with a name SETUP_MACRONAME or SETUP_PROCENAME as defined by the Protocol statement for the Pattern. There cannot be a Setup statement in a Pattern with no P statements to follow.

(3)–(4) **Protocol**: This keyword begins the statement that identifies the Macro or Procedure to be applied from P-statements in the Patterns. There are two forms of the Protocol statement, one for Macros and one for Procedures. For any given Pattern or PatternBurst, this statement shall be consistent with the Protocol statement in the PatternInformation block of statements for the associated pattern or burst. This statement shall be present only when the Patterns contain P-statements. A PatternBurst that invokes Patterns with the P-statement is required to have the Protocol statement. It is an error to use this statement for Patterns that do not require it.

If an associated MacroDefs or Procedures statement is used in the PatternBurst, then the Protocol statement is expected to appear after the statement that identifies the block where the Macro or Procedure is defined. Statement (3) is used when all Patterns to follow (in the PatList/PatSet/ParallelPatList) invoke the same protocol. Statement (4) is used on a pattern-by-pattern statement to identify the name of the protocol used by the patterns.

Macro MACRONAME (SETUP_MACRONAME): This part of the Protocol statement is used to identify the macro name that is invoked by the patterns. The MACRONAME is expected to be a valid name of a macro within the scope of the test mode that uses the PatternBurst with this statement. The optional SETUP_MACRONAME identifies the name of the macro that is invoked by the optional Setup statement in the Pattern. If a SETUP_MACRONAME is defined in this statement, there shall be a corresponding Setup statement in the Pattern.

Procedure PROCNAME (SETUP_PROCNAME): This part of the Protocol statement is used to identify the procedure name that is invoked by the patterns. The PROCNAME is expected to be a valid name of a procedure within the scope of the test mode that uses the PatternBurst with this statement. The optional SETUP_PROCNAME identifies the name of the procedure that is invoked by the optional Setup statement in the Pattern. If a SETUP_PROCNAME is defined in this statement, there shall be a corresponding Setup statement in the Pattern.

The Pattern syntax in one of its forms is primarily limited to multiple occurrences of the P-statement and an optional Setup statement. This Pattern format requires the Protocol statement in the PatternBurst that invokes the Pattern or the Protocol statement in the PatternInformation block. The statements are allowed to be labeled with a LABEL, which is the same as that defined in IEEE Std 1450-1999. In addition, the Loop, Goto, Breakpoint, and X statements are allowed.

(5) **Setup { }**: This statement can occur only as the first statement in a Pattern block. This invokes a Protocol (Procedure/Macro) as defined by the Protocol statement in the associated PatternBurst or the Protocol statement on the PatternInformation of the associated Pattern. The Protocol statement shall exist in at least one of the two locations. If a PatternBurst exists that invokes Patterns with the Setup statement, the Protocol statement is required in the PatternBurst. If it appears in the PatternInformation and the PatternBurst invoking this pattern, then the information shall be consistent or an error condition exists. The Setup statement cannot be embedded within a loop in the Pattern such that it is executed twice.

(6) **P**: This invokes a Protocol (Procedure/Macro) once with no parameters. The procedure or macro name is defined by the Protocol statement in the associated PatternBurst or the Protocol statement on the PatternInformation of the associated Pattern. The Protocol statement shall exist in at least one of the two locations. If a PatternBurst exists that invokes Patterns with the P-statement, the Protocol statement is required in the PatternBurst. If the Protocol statement appears in the PatternInformation and the PatternBurst invoking this pattern, then the information shall be consistent or an error condition exists.

(7) **P { }**: This invokes a Protocol (Procedure/Macro) once with parameters. The procedure or macro name is defined by the Protocol statement in the associated PatternBurst or the Protocol statement on the PatternInformation of the associated Pattern. The Protocol statement shall exist in at least one of the two locations. If a PatternBurst exists that invokes Patterns with the P-statement, the Protocol statement is required in the PatternBurst. If the Protocol statement appears in the PatternInformation and the PatternBurst invoking this pattern, then the information shall be consistent or an error condition exists.

In the second format, the Pattern syntax is primarily limited to Macro and Procedure calls. This format of the Pattern syntax cannot be used in conjunction with the Protocol statement. The Macro and Procedure calls are used to separate the data from the protocol information such that the protocol can be modified without having access or having to modify the data in the Patterns. The data are provided to the Macros and Procedures as parameters. The definition for these statements should be taken from the associated construct in IEEE Std 1450-1999 and IEEE Std 1450.1-2005.

(8), (9), (10), (11) See IEEE Std 1450-1999 and IEEE Std 1450.1-2005.

In addition to the statements defined above, the following statements are allowed in the Patterns.

(12) The **Loop** statement is defined by IEEE Std 1450-1999. This statement is restricted by CTL to loop on multiple Macros, Procedures, or multiple P-statements as allowed by the type of Pattern syntax. The LOOPCNT is not allowed to impact the length of a shift-like operation (defined as a Shift or serialized version of it) within the protocols invoked within the Loop. That is, there should be no relationship between the LOOPCNT and the length of scan data.

(13) The **Goto** statement is defined by IEEE Std 1450-1999. This statement is not allowed to be used in situations that would cause infinite loops or unpredictable cycle-counts.

(14) The **BreakPoint** statement is defined by IEEE Std 1450-1999.

(15) The **X** statement is defined in IEEE Std 1450.1-2005. This statement is used to tag statements similar to the mechanism of the label allowed with every statement, with the difference that the tag need not be unique. For details, refer to the associated standard document.

5.7 Extensions associated with the LockStep construct of Clause 13 of IEEE Std 1450.1-2005

5.7.1 General

When signal resources are shared between cores across multiple patterns, the patterns need a synchronization mechanism that is explicitly defined by a common protocol (Macro or Procedure). The rules of LockStep defined in 1450.1 provide the generic framework for this synchronization and remain consistent with the refinements defined in this standard. The refinements of LockStep are defined here. The following are the additional requirements for the LockStep construct:

a) Only Patterns that use the P-statement syntax are allowed to be used with the LockStep construct.
b) All Macros or Procedures that are invoked by patterns under LockStep shall be active within the scope of a single test mode.

c) All protocols of the patterns that are under LockStep are required to resolve to the same macro/procedure that is identified as a LockStep macro/procedure. This may occur through an indirection of Macro or Procedure calls.
d) When a pattern or set of patterns under LockStep has a different number of calls to the common Macro or Procedure than the other pattern or set of patterns it is synchronized with, all (set of) Patterns with fewer calls to the protocol are assumed to have automatic padding at the end that is defined as calls to the common protocol without any values provided for the parameters.

5.7.2 Associated syntax constructs (extensions to 24.1 and 24.3 of IEEE Std 1450-1999)

Common Macros and Procedures that are used by patterns under LockStep are defined along with the other protocols. These special common Macros and Procedures are identified by a LockStep keyword. The names of the parameters are required to use their fully qualified name, which includes the domain name. The syntax is as follows:

```
MacroDefs (MACRO_DOMAIN_NAME) {
    ( MACRO_NAME (LockStep) {
        (PATTERN_STATEMENT)*
    })*
}
Procedures (PROCEDURE_DOMAIN_NAME) {
    ( PROCEDURE_NAME (LockStep) {
        (PATTERN_STATEMENT)*
    })*
}
```

The MACRO_NAME and PROCEDURE_NAME defined with the LockStep keyword are in the same name space as their parallel constructs without the keyword defined in MacroDefs and Procedures, respectively. The PATTERN_STATEMENTS allowed and requirements are the same as those allowed in Macros and Procedures as defined by IEEE Std 1450-1999, IEEE Std 1450.1-2005, and this standard. The following requirements are to be used for Macros and Procedures defined with the LockStep keyword:

a) All names of entities in domained constructs such as variables, signal groups, timing, and nested macros or procedures are used with the naming convention DOMAIN_NAME::ENTITY_NAME or ENTITY_NAME when there is no DOMAIN_NAME. For example, the signal group “allsignals” defined in the named SignalGroups block “mygroups” would be refered to as mygroups::allsignals. The DOMAIN_NAME is *mygroups*, and the ENTITY_NAME is *allsignals*.
b) Calls to protocols (Macro or Procedure) within these special LockStep protocols are limited to only protocols that are identified with the LockStep keyword.

LockStep Macros and Procedures can be invoked by Macros, Procedures, or Pattern constructs in the same way the Macro in the MacroDefs blocks is called and the Procedure in the Procedures blocks is called.

5.7.3 Description of the LockStep construct

CTL has been designed for the manipulation of protocols without modifying the test patterns or even having the test patterns available. As a result, all synchronization information across multiple patterns needs to be explicit in the protocol portion of the patterns. Explicitness is achieved through a common protocol.

5.7.4 Example of the LockStep construct

Lockstep is created to support the reusability of test patterns of the cores. In test pattern reuse, cores come with their own patterns that are written to the boundary of the core. When the cores are embedded in an SoC, these patterns are invalid in the existing form as they are specified to internal points of the SoC. System integration tasks reuse the test patterns at the embedded core boundary with a rewrite of the protocols but not the bulk pattern data. Test patterns in CTL have the Pattern (data portion of the test) calling the Macro or Procedure (sequence portion of the test). When the core is embedded in an SoC without any dependencies on other cores, the retargeted patterns do not require any synchronization with other patterns and the LockStep construct is not useful. However, if patterns from different cores have dependencies on each other, then synchronization is required.

The intent of this example is to create a design that takes two existing cores with one scan chain each and to integrate them at the chip level by creating a single scan chain (Figure 5). The CTL describes the patterns of the resulting chip.

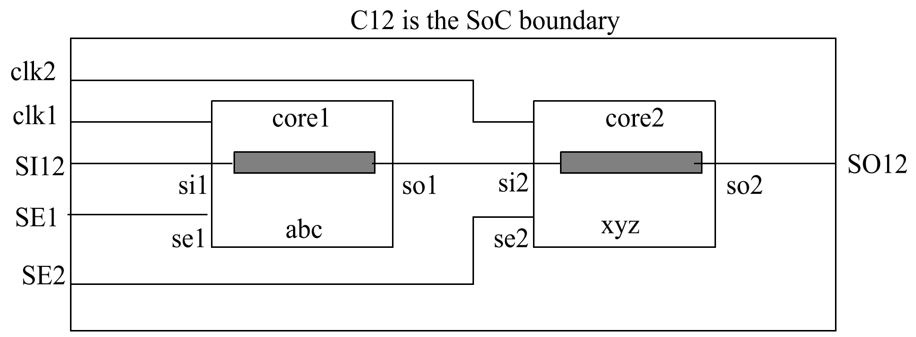


Figure 5—Example design in which two cores are integrated by connecting their scan chains to form a single larger scan chain

The two cores came with their own patterns that had no dependencies between them as they were written to the context of an isolated design. However, the integration process has created dependencies between these patterns. In this case, the dependency is of the sharing of the common scan-in and scan-out by daisy chaining the scan chains of the cores. The integrated patterns would use the LockStep construct to resolve these dependencies.

In the example that follows, core *core1* came with patterns P1, which would have called a Macro that was written to the boundary of *core1*. Similarly, core *core2* came with patterns P2, which would have called a Macro that was written to the boundary of *core2*. In the SoC design (named C12), the system integrator has connected the scan chains of the cores as shown Figure 5. The remapped patterns call the same Macro (namely M12), and the resulting patterns that operate on the SoC boundary would be as follows:

```
STIL 1.0 {
   Design 2005;
   CTL 2005;
}
Signals {
   SI12 In; SO12 Out;
   SE1 In; SE2 In; clk1 In; clk2 In;
}
// Timing is not the focus of the example; details not shown.
Timing { WaveformTable WBoth {
   // timing details not shown
}}
Variables core1 {
   SignalVariable scaninvals[5..0];
   SignalVariable scanoutvals[5..0];
}
Variables core2 {
   SignalVariable scaninvals[3..0];
   SignalVariable scanoutvals[3..0];
}
MacroDefs {
 M12 LockStep {
    W WBoth;
       // Refer 1450.0 and 1450.1 for explanations for
       // # and the C statement.
       C { core1::scaninvals=#; core1::scanoutvals=#;
           core2::scaninvals=#; core2::scanoutvals=#;
           SE1 = 1; SE2 = 1;}
    Shift { V{ SI12 = \W core2::scaninvals[3..0]
                     \W core1::scaninvals[5..0];
                  SO12 = \W core2::scanoutvals[3..0]
                     \W core1::scanoutvals[5..0];
                  clk1=P; clk2=P}}
 }
}

PatternBurst B1 {
    ParallelPatList LockStep {
         // P1 and P2 are redirected to call M12 instead of their
         // respective macros that were written to the core boundary.
         P1 {Variables core1; Protocol Macro M12;}
         P2 {Variables core2; Protocol Macro M12;}
    }
}
PatternExec e1 {
   PatternBurst B1;
}
Environment {
 CTLMode coretests12 {
     TestMode InternalTest;
       DomainReferences {
         Variables core1 core2;
     }
       PatternInformation {
         PatternExec e1 { Purpose Production; PatternBurst B1;}
         PatternBurst B1 { Purpose ChainContinuity; }
       }
   }
}
// Original patterns that came with the cores. They remain untouched
// here. However, in this example, they are valid from the boundary
// of the SoC.
Pattern P1 {
   P{ scaninvals[5..0]=101010; scanoutvals[5..0]=LLHHLL;}
}
Pattern P2 {
   P { scaninvals[3..0]=1111; scanoutvals[3..0]=LHLH;}
}
```

6. Design hierarchy—cores

6.1 CoreType block and CoreInstance statement

The CoreType block refers to an internal level of hierarchy or a subdesign that is created by an imaginary boundary in the design, which is refered to as a core. This construct shall be used in situations dealing with hierarchical cores. This entity internal to the design has CTL on its boundary, and the Environment of the CTL that is being integrated into the design is pointed to by the keyword CoreEnvironment. There may be multiple instances of a subdesign; in which case, multiple core instance names are specified. There shall be only one CoreType block for each core type used in a design.

```
(CoreType CORE_TYPE_NAME {                                    (1)
   ( CoreEnvironment CORE_ENV_NAME; )                         (2)
   ( CoreInstance CORE_TYPE_NAME {                            (3)
      (CORE_INSTANCE_NAME;)+
   } )* // end Inherited CoreType- Instance definition
   ( Signals {                                                (4)
      core_signal_definitions
   } ) // end Signals
   ( SignalGroups (DOMAIN_NAME) {                             (5)
      core_signal_group_definitions
   } )* // end SignalGroups
   ( ScanStructures (SCAN_STRUCT_NAME) {                      (6)
      core_scan_structure_definitions
   } )* // end ScanStructures

})*// end CoreType

( CoreInstance CORE_TYPE_NAME {                               (7)
   (CORE_INSTANCE_NAME;)+
})*
```

6.2 CoreType block syntax descriptions

(1) **CoreType**: This statement begins the definition of the block that defines a level of hierarchy within the design or a subdesign that is created by an imaginary boundary in the design. There can be multiple instantiations of the named core type in the design as specified by the *CoreInstance* statement (7). This is the only mechanism that is to be used to define a hierarchy in the design.

CORE_TYPE_NAME: It identifies the core type definition with a name. This name shall be unique across all core types of the design. The core type name shall follow the naming conventions for user-defined names as defined by IEEE Std 1450-1999.

(2) **CoreEnvironment** CORE_ENV_NAME: Every reusable design entity is expected to be represented in CTL. Through this statement, the history of the *Environment* that was used to integrate the core into the design is maintained. CORE_ENV_NAME is the name of the *Environment* block in the CTL file that came with the reused design that is represented by the current *CoreType* (1).

(3) **CoreInstance** CORE_TYPE_NAME: This statement is used to define a core that is embedded inside another core. This operation defines instances of the inherited core (CORE_TYPE_NAME) in the current core type as identified by the embedded CORE_INSTANCE_NAMEs. The CORE_TYPE_NAME is required to be a valid *CoreType* within the current CTL.

> CORE_INSTANCE_NAME: The instances (CORE_INSTANCE_NAME) are defined for the embedded *CoreInstances* to define several instances of the CORE_TYPE_NAME. The contents of these instances are accessed by prefixing the names of the inherited information with CORE_INSTANCE_NAME followed by a ":". The core_instance_name should be a valid user defined name that follows the naming convention defined in IEEE Std 1450-1999.

(4) **Signals**: This block (analogous to the global Signals block of the top-level design) defines the signals of this core type. The signals are available to the top level by referencing as core_inst_name:signal_name. All statements defined in IEEE Std 1450-1999 in the Signals block are allowed to be used on the CoreType definition of the Signals. These signals define the hierarchical boundary of the embedded design.

(5) **SignalGroups**: Analogous to the SignalGroups for the top-level design, the signals of the CoreType being described can be grouped. The signal-groups are available to the top level by referencing as core_inst_name:signal_group_name. If multiple signal-group domains exist, the names are specified as: core_inst_name:domain_name::signal_group_name.

(6) **ScanStructures**: This block defines the scan structures that are contained within this core type. The scan structures definition is the same as that of the Scan Structures defined by IEEE Std 1450-1999 and IEEE Std 1450.1-2005. The signal names used to define the scan chains are required to be part of the Signals block defined within the CoreType block. The scan cells are available to the top level by referencing as: CORE_INSTANCE_NAME:(DOMAIN_NAME::)CHAINNAME or CORE_INSTANCE_NAME:CELLNAME.

(7) **CoreInstance** CORE_TYPE_NAME: This statement is a top-level statement that defines the instance names of all cores of a given type. The CORE_TYPE_NAME is required to be a valid *CoreType* within the current CTL.

> CORE_INSTANCE_NAME: A unique core instance name across all *CoreInstances*-CORE_TYPE_NAME blocks that follows the naming conventions defined by IEEE Std 1450-1999. This name represents a hierarchy at the top level of the design for which the CTL is being written. As a result of this definition, all entities defined in the associated *CoreType* (CORE_TYPE_NAME) of this block of statements are prefixed with the CORE_INSTANCE_NAME followed by a ":". Subcores of this core instance are uniquely identified by the concatenation of the core instance names separated by ":". The names are concatenated in the order beginning with the highest hierarchical entity to the embedded entity.

6.3 CoreType block code example

This is a CoreType code example for an integrated chip that has two instances of core *egcore* embedded. The two instances are named *A1* and *A2*.

```
CoreType egcore {
   Signals { s[1..10] In; si_core In; }
   SignalGroups { g1[0..4] = 's[1..5]'; g2[0..4] = 's[6..10]'; }
   ScanStructures {
     chain1 {
       ScanCells c[0..5];
     }
   }
}
CoreInstance egcore { A1; A2; }
```

```
// The names of the signals on A1 are A1:s[1..10] and A1:si_core.
// The names of the signals on A2 are A2:s[1..10] and A2:si_core.

Environment { CTLMode {
   CoreInternal {
     A1:s[1] { /* information on the signal */ }
     A1:g1[1..4] { /* information on the signal */}
     A1:si_core { /* information on the signal */}
     A2:s[1..10] { /* information on the signal */ }
     A2:si_core { /* information on the signal */}
   }
   ScanInternal {
     A1:c[0..5] { /* information on the scan cells of A1 */ }
   }
}}
```

NOTE—No domain names exist in these definitions. If domain names existed, they would be optionally used to create unique names that could take the form CoreInstanceName:DomainName::entityname

7. Cell expression (cellref_expr)

A cellref_expr is defined to be a similar construct to the *sigref_expr* as defined by 6.14 of IEEE Std 1450-1999. Instead of signals and signal-group names the cellref_expr allows cells and cell-groups to be used. The cell expressions define an ordered list of cells; they are either a single token, or an expression enclosed in single quotes. Cell expression operators are plus (+), minus (–), ellipsis (..), and parenthesis. These operators are not extendable. Expressions are evaluated left to right, with parenthesis used to override this order. Cells referenced in cell expressions may occur only once in the subexpressions generated during evaluation of the expression. The content of cell groups are to be evaluated assuming an implicit plus (+) operator between the list of cells it represents.

Just like its counterpart, the cell expression shall not "remove" a cell (using a minus operator) that is not part of an expression as currently defined, and an expression shall not "add" a signal (using the plus operator) that is already part of an expression as currently defined.

The above represents a parallel construct of the *sigref_expr* with an assumed plus operator between cell names that are part of cell groups. Refer to examples of *sigref_expr* in IEEE Std 1450-1999 for more explicit use of the operators.

Example Syntax of cellref_expr:

```
ScanStructures {
   ScanChain grp1 {
     ScanLength 10; ScanCells {aa[0..9];}
   }
   ScanChain grp2 {
     ScanLength 10; ScanCells {bb[0..9];}
   }
   ScanChain grp3 {
     ScanLength 30; ScanCells { grp1; grp2; cc[0..9];}
   }
}
Environment { CTLMode foo { ScanInternal {
   grp3 { // information on grp3 }
   'grp3-cc[1,3,5,7,9]' { // information on individual scan cells
               // after removing cc[1], cc[3], cc[5], cc[7], cc[9] }
}}}
```

8. Environment block—extensions to IEEE Std 1450.1-2005, Clause 17

8.1 General

The Environment block is a block in STIL for the purpose of defining data about the design. This data may include raw attributes as defined in this standard, sequence information about the design configuration, or pattern data.

In CTL, the Environment block is the top-level block that contains information regarding the design. When information in the Environment requires STIL constructs, a reference is created in the Environment to the STIL construct and the construct is defined outside of the Environment. For example, design entities are named by reference in *sigref_expr*'s and *cellref_expr*'s. Pattern constructs are refered to by name in the Environment. As such, the positioning of the Environment block will be after all of the referenced blocks to satisfy the define before use requirement of STIL. It typically will be immediately before the Pattern blocks.

Although multiple Environment blocks can exist for a design, only one named environment block is assumed to contain all necessary CTL information. In case no named environment block exists, then the global (nameless) Environment block is to be used. Thus, multiple orthogonal-named Environment blocks can be created for a design to describe completely different test methodologies with the understanding that only one is to be used.

8.2 Definition of FileReference keywords

The FileReference statement as defined in 1450.1 is reproduced below for reference. In 1450.1, the Type and Format statements are defined, but the definition of the keyword parameters are left to the individual environment to define. Below are the keywords as defined for use in this CTL standard:

(**FileReference** "FILE_PATH_NAME"**;**)*
(**FileReference** "FILE_PATH_NAME" {
  Type *file_type* **;**
  Format <*pattern_file_format* | *design_file_format* | *layout_file_format* | *fault_list_file_format* |
    script_file_format | *doc_file_format*>**;**
  Version "VERSION_NUMBER" **;**
})* *// end FileReference*

file_type =
  < **Pattern**
  | **Design**
  | **Layout**
  | **FaultList**
  | **Script**
  | **Documentation**

| **User** USER_DEFINED >

pattern_file_format =
< **STIL**
| **WGL**
| **Verilog**
| **VHDL**
| **VCD**
| **User** USER_DEFINED >

design_file_format =
< **CTL**
| **EDIF**
| **Verilog**
| **VHDL**
| **User** USER_DEFINED >

layout_file_format =
< **GDSII**
| **DEF**
| **LEF**
| **Oasis**
| **User** USER_DEFINED >

fault_list_file_format =
< **DTIF**
| **User** USER_DEFINED >

script_file_format =
< **AWK**
| **Perl**
| **SED**
| **Python**
| **Tcl**
| **User** USER_DEFINED >

doc_file_format =
< **HTML**
| **PDF**
| **Postscript**
| **RTF**
| **Text**
| **User** USER_DEFINED >

FileReference "FILE_PATH_NAME"**:** The FileReference statement is used within an Environment block to specify various other files associated information that is not already in the scope of the CTL through the include statement. The content and application of the referenced files are specified by this statement. All constructs referenced by the ForeignPatterns statement in CTL shall have a FileReference statement. Refer to 1450.1 for details on the FILE_PATH_NAME.

Type *file_type*: Specifies the type of this file. The file type shall be one of the specified types, or else User followed by the user type name.

Pattern: Any of the pattern file formats as enumerated below in the pattern file formats. Pattern files contain the information that defines the input logic values and expected logic values of the device under test. Normally a detailed timing for these logic levels is included or a timing file is referenced. Likewise, logic levels, that is, the actual physical values to be forced and sensed, are included or included by reference to another file.

Design: Design files contain the behavioral, logical, and structural description of the device. A design file shall define the behavior and/or logic of the device, or detail the structure of the device. Various forms of the design are enumerated below in the design_file_format.

Layout: Any of the formats enumerated in the layout_file_format that describes the physical form of the device as is implemented to conform to the behavior specified in the design file. This file, as well as the design file, is needed to fabricate the final device. The layout is valuable in predicting the actual timing behavior as well as the logical behavior of the device.

FaultList: A file documenting the faults by various fault criteria that are associated with the design. Usually some kind of coverage metric is included that indicates what level of fault detection is achieved by an associated test program.

Script: A file that is referenced in the CTL because it is useful for performing some automated process in test generation or design. These are typically executable files that operate on one or more of the other files in this list.

Documentation: Any file that contains information useful in explaining what the design does, how to use the design, what was done to realize the physical implementation, or what needs to be done. Formats for these files are documented below in doc_file_format.

User USER_DEFINED**:** Any file that the user wishes to reference. (Normally such a file would not be one of the standard formats and would not be processed by standard tools.)

Format *file_format*: Specifies the format of this file type. The file format shall be one of the specified formats for the associated type, or else User followed by the user type name.

pattern_file_format:

STIL: Standard Test Interface Language. This term generally refers to IEEE Std 1450-1999. This is the first version of the language. STIL allows a way to completely specify logic levels, timings, parametric tests, pattern order, and so on for a test program. Patterns in this format do not follow the pattern restrictions specified in this standard.

WGL: An ascii test pattern description language placed into the public domain by TSSI (currently under IMS, a wholly owned subsidiary of Credence). WGL is a text representation of the binary database WDB.

Verilog: Although Verilog is a Hardware Description Language, there are ways to describe event lists in Verilog. This is a way of specifying the purely logical and timing part of a test program.

VHDL: As with Verilog, event lists can be created in VHDL, which is an alternative, somewhat more formal HDL.

VCD: Value Change Dump. This is a widely used file format to record simulation results and event lists. It is also frequently used to generate the logical and timing parts of test programs. This file format is produced by most Verilog simulators and is documented in the standard.

design_file_format:

CTL: The common name for the language that encompasses syntax from this standard, IEEE Std 1450-1999, IEEE Std 1450.1-2005, and IEEE Std 1450.2-2002. This is a test-mode-oriented language to describe all test-related information about a design.

EDIF: Electronic Data Interchange Format. A standard language used to describe hardware in the form of Net lists. Other parts of the format describe physical layout. EDIF is widely used for Printed Circuit Board design, and it is useful in device and semiconductor engineering.

Verilog: A hardware description language. (HDL) A language that describes hardware logical behavior and the related timing. The language specifications are in IEEE Std 1364-2001.

VHDL: A hardware description language. Evolved as a second-generation language and has greater formality and consistency than some of the earlier languages like Verilog. Language specification is in IEEE Std 1076-2000.

layout_file_format:

GDSII: The file from which photolithography masks are made for creating semiconductors. This is the physical specification for the various layers to be created by photolithography.

DEF: Namely the Design Exchange Format in which the elements of the IC design relevant to physical layout, including the netlist and design constraints.[5]

LEF: Namely the Library Exchange Format in which an IC process technology and associated cell models are represented.[6]

Oasis: It is a 64 bit compact layout format replacing GDSII.[7]

fault_list_file_format:

DTIF: Digital Testing Interchange Format. The information content and the data formats for the interchange of digital test program data between digital automated test program generators (DATPGs) and automatic test equipment (ATE) for board-level printed circuit assemblies or semiconductor chips are defined. This information can be broadly grouped into data that define the following: UUT Model, Stimulus and Response, Fault Dictionary, and Probe. The specification is IEEE Std 1445-1998**.**

script_file_format:

AWK: A popular scripting tool available in many operating systems. Awk allows one to manipulate text files by way of cryptic commands. It is very useful for formatting columns of data.

SED: Another tool used in available in many operating systems. The name "SED" comes from Stream Editor. This is a scripted editor for text files. Very useful for doing substitutions and changes.

Perl: A rational attempt to eliminate all separate scripting languages such as SED and AWK, in which each has a unique syntax. Perl closely follows C language syntax and has many of the capabilities of C. Implementations are available in the public domain.

[5]DEF is available from OpenEDA.org.
[6]LEF is available from OpenEDA.org.
[7]Standardized by, and available from, SEMI.org.

Python: An interpreted, interactive, object-oriented programming language. It is often compared with Tcl, Perl, or Java. Python supports modules, classes, exceptions, very high level dynamic data types, and dynamic typing. There are interfaces to system calls, libraries, and windowing systems. It is usable as an extension language for applications that need a programmable interface.

Tcl: Tcl is a scripting language something like Perl but extensible and somewhat cleaner. Many extensions are available. Tk is an extension developed by the creator of Tcl and is used for creating scripts that interact with users through windows. Tcl was meant to be portable, and TcVTk has been ported to many different operating systems.

doc_file_format:

HTML: Hyper Text Markup Language. The current language of the World Wide Web. A standardized graphics language interpreted by Internet browsers. HTML is useful as a way of communicating documents because almost every kind of host computer has an Internet browser available in the public domain. It is very close to a universal documentation and graphics tool.

PDF: Portable Document File. A public domain PDF format reader is universally available for almost any kind of host computer. It is much more complicated and more powerful than HTML and is oriented toward the creation of document rather than Web page display as is the case for HTML.

Postscript: A graphics language that is used to drive many kinds of printers, and it is a display language for some kinds of computer systems. There are public domain readers to display files on many computer operating systems.

RTF: Rich Text Format. Another way of representing text documents that contain formatting commands and graphics. It is produced and consumed by some word processing programs. It is also produced and consumed by some page layout programs.

Text: Pure ASCII character files. There are minimal formatting features. Most files are characters with only new line, new page, and indention characters. Normally, there is no provision for graphics. These files are nearly universally readable and one can easily create code to parse text.

Version “VERSION_NUMBER”;: A quoted string identifying the version of this file. The format and information of the VERSION_NUMBER is dependent on the file type and format and not defined here.

8.3 Example of Environment block FileReference syntax

```
Environment core_example {
   FileReference "$xyz/pats/patfile1.wgl" {
     Type Pattern;
     Format WGL;
     Version "4.3";
   }
}
```

8.4 Extension to NameMaps

(**NameMaps** (MAP_NAME) {
    (**CoreType** CORE_TYPE_NAME {
        (**CoreInstance** {(CORE_INSTANCE_NAME “map_string”;)+})
        (**Signals** { (SIGNALNAME “map_string”;)+})
        (**ScanCells** { (CELLNAME “map_string”;)+ })
    })*

(**CoreInstance** { (CORE_INSTANCE_NAME "map_string";)+ })
})

Similar to the other namemaps constructs defined in 1450.1 the names of core-instances and entities within it can be mapped to another name. The mapping definition is exactly the same as that defined in 1450.1 for the other constructs. See the definition of map_string in 1450.1. The mapping of the separators as defined in 1450.1 remains valid.

8.5 Extension to the inheritance of environment statements

Only a single InheritEnvironment statement is allowed. InheritEnvironment rules also apply to all CTLMode blocks defined in the Environment.

The nameless Environment block is automatically inherited by all other named Environment blocks. Inheritance in CTL creates a chain of information flow between the unnamed blocks and the named inherited blocks, with the unnamed block at the top of the chain. Inherited information is overridden in blocks by redefinition of the same construct. Thus, a named CTLMode block of statements will override any inherited CTLMode block of statements with the same name. A unnamed CTL block will redefine the unnamed CTL block that it has obtained through inheritance. Resolution of information for any Environment occurs in the following order:

a) Resolution of information inherited across Environments
b) Resolution of inherted information within an Environment

These two steps require that the first step only be performed when inherited information is already resolved. Thus, in a chain of Environments that inherit information from each other where inherited information flows down the chain, the information shall be resolved for the Environment at the top of the chain and be successively evaluated for Environments that inherit information from it down the chain.

Information sharing or inheritance between CTLMode blocks is defined by the Inheritance rules defined with the InheritCTLMode statement (see 9.3).

Inheritance across environments at the high level works as follows. Details of the inherited information is described in the previous example:

```
Environment {
   CTLMode { /* information segment I */ }
   CTLMode mode1 { /* information segment II */ }
   CTLMode mode2 { /* information segment III */ }
}
Environment myEnv {
   CTLMode { /* information segment IV */ }
   CTLMode mode1 { /* information segment V */ }
}
Environment anotherEnv {
   CTLMode mode2 { /* information segment VI */ }
}
```

The two named Environenements inherit information from the nameless Environment. Thus, two chains of inherited information exist, as follows:

— Environment {} - Environment myEnv {}
— Environment {} - Environment anotherEnv {}

The resolution of information begins with the nameless Environment in both situations. The nameless Environment does not inherit from any other Environment. The mode1 and mode2 of the nameless Environment inherit information from the nameless CTLMode block within the Environment. Then the information is resolved for myEnv and anotherEnv. Both named Environments first inherit the blocks of information and then resolve the blocks within their respective Environments.

Thus, the nameless environment-mode1's information is constructed out of segments I and II. Information of the nameless environment-mode2 is constructed out of segments I and III. Information of myEnv-nameless_CTLMode_block overides the nameless CTLMode block being inherited. Information of myEnv-mode1 is constructed with segments IV and V. myEnv has a mode2 that is constructed out of segments I and III. Finally, anotherEnv has two named modes. anotherEnv-mode1 is constructed out of segments I and II. anotherEnv-mode2 is constructed out of segments I and VI. Although the information is split across three environment blocks, when named environments exist, the nameless environment block is not to be used explicitly other than to determine the information in the other environment blocks. Each of the two named environments is considered to be complete representations of the information. That is, information from myEnv and anotherEnv should not be combined (refer to 9.3).

9. CTLMode block

9.1 General

The CTLMode block typically defines a configuration of the design, which is also called a test mode. If the block is unnamed, then it is the global block containing information common to all CTLMode blocks within the Environment. Typically, a file will have a global block and then one named block for each test mode, which would reflect different configurations of the design such as internal-test and external-test.

9.2 CTLMode syntax

test_mode_enum =
    < **Bypass**
    | **Controller**
    | **Debug**
    | **ExternalTest**
    | **ForInheritOnly**
    | **InternalTest**
    | **Isolate**
    | Normal
    | **PowerOff**
    | **PowerOn**
    | **PreLoad**
    | **Quiet**
    | **Repair**
    | **Sample**
    | **Transparent**
    | **User** USER_DEFINED >
exec_enum =
    < **Characterization**
    | Diagnostic
    | **Production**
    | **Verification**
    | **User** USER_DEFINED >
pattern_or_burst_enum =

```
    < AtSpeed
    | ChainContinuity
    | CompatibilityInformation
    | Endurance
    | EstablishMode
    | IDDQ
    | LogicBIST
    | MemoryBIST
    | Padding
    | Parametric
    | Retention
    | Scan
    | TerminateMode
    | User USER_DEFINED >

Environment (ENV_NAME) {                                                        (1)
    ( CTLMode (CTLMODE_NAME) {                                                  (2)
        ( CoreInternal {} )                                                     (3)
        ( DomainReferences ( CORE_INSTANCE_NAME)* {                             (4)
            ( Category (CATEGORY_NAME)+; )
            ( DCLevels (DC_LEVELS_NAME)+;)
            ( DCSets (DC_SETS_NAMES)+;)
            ( MacroDefs ( MACRO_DEF_NAME )+ ; )
            ( Procedures ( PROCEDURE_NAME )+ ; )
            ( Selector ( SELECTOR_NAME)+;)
            ( SignalGroups ( SIGNALGROUPS_NAME )+ ; )
            ( ScanStructures ( SCAN_STRUCT_NAME )+ ; )
            ( Timing ( TIMING_NAME )+ ; )
            ( Variables ( VARIABLES_NAME)+; )
        } )* // end DomainReferences
        ( External { } )                                                        (5)
        ( Family (NAME)+ ; )                                                    (6)
        ( Focus ( Top) (CoreInstance (CORE_INSTANCE_NAME)+ ) {                  (7)
            ( PatternTypes (pattern_or_burst_enum)+ ; )                         (8)
            ( CTLMode CORE_INSTANCE_NAME CTLMODE_NAME; )*                       (9)
            ( TestMode (test_mode_enum)+ ; )                                    (10)
            ( Usage (exec_enum)+ ; )                                            (11)
        } )* // end Focus
        ( InheritCTLMode CTLMODE_NAME; )                                        (12)
        ( Internal { } )                                                        (13)
        ( PatternInformation { } )                                              (14)
        ( Relation { } )                                                        (15)
        ( ScanInternal { } )                                                    (16)
        ( ScanRelation { } )                                                    (17)
        ( TestMode (test_mode_enum)+ ; )                                        (18)
        ( TestMode (test_mode_enum)+ {
            AlternateTestMode (CTLMODE_NAME)+ ;                                 (19)
        }) // end of TestMode
        ( TestModeForWrapper WRAPPER_TEST_MODE TEST_MODE_CODE; )                (20)
        ( Vendor (NAME)+ ; )                                                    (21)
        ( Compliancy < IEEE1500 EXT_VERSION <Wrapped | Unwrapped>               (22)
                        | None | User USER_DEFINED > ; )
    } )* // end CTLMode
} // end Environment
```

9.3 CTLMode block—syntax descriptions

(1) **Environment** ENV_NAME: This is a STIL statement within which CTL statements are embedded. CTLMode is a construct (within the Environment block defined by 1450.1) that builds on existing STIL structures. All CTLMode blocks within an environment are applicable to a design being described. For details on the environment block, refer to Clause 8 or IEEE Std 1450.1-2005. The CTLMode block defined in this clause coexists with the statements defined in 1450.1 for the environment. Environment blocks may be named (ENV_NAME) according to its definition in 1450.1. Many environments containing CTL information can be defined with the understanding that only one of these named environments contains the complete information. In the absence of a named environment that contain CTLMode blocks, the unnamed environment is to be considered.

(2) **CTLMode** CTLMODE_NAME: This statement begins a block that contains all of the information needed to describe a configuration of the design for the purpose of testing. Each CTLMode block can be assigned a user-defined name (CTLMODE_NAME) to identify the block. CTLMODE_NAME is required to follow the naming rules for user-defined names as defined by IEEE Std 1450-1999. There can be only one CTLMode block without a name in an Environment.

The nameless CTLMode block, called the global CTLMode block, contains information that is common to all named CTLMode blocks. If a global CTLMode block is created, all of the information other than the PatternInformation block is automatically inherited into all other named CTLMode blocks. The information is overridden by local definitions of the same constructs in the named CTLMode blocks receiving this information. The inherited information on signals, signalgroups, cellnames, or cellgroups is overridden only when the same statement is redefined on the associated entity. If a chain of inherited CTLMode blocks exist, the global CTLMode block is available only once to any CTLMode block of the chain as the block of information at the head of the inheritance chain. Although pattern information is not inherited, it can be used to define information that spans test modes. For example, the fault coverage of patterns that encompasses the tests of different modes can be given in the pattern information of the global CTLMode block. More details are available in the InheritCTLMode statement definition.

The following rules apply across all statements used to describe information for a test mode (which essentially means the complete CTL language):

a) *Design structures*: All design structures defined in CTL should have a one-to-one correspondence with the associated entities in the design being described. Terminals of the design are defined in Signals, scan cells of the design are defined in ScanStructures, and hierarchies of the design are defined with CoreInstance-CoreTypes. These design entities are referenced in *sigref_expr* and *cellref_expr's* in the Environment.
b) *Conflicting definitions*: Local definitions of terms override globals or inherited terms in case of conflicts of the same keyword.
c) *Multiple use of the same keyword with associated parameters*: Unless specified otherwise, this defines a logical OR relationship between the types defined for the keyword.
d) *Missing information*: Unless specified in the definitions, if a keyword or block is missing, it should be assumed that the information was not required or the function that the keyword is describing does not exist.
e) *Symbolic names*: This represents a unique name across all CTLs. The symbol name follows the naming conventions for user-defined names (IEEE Std 1450-1999). That is, if multiple references exist to a symbolic name in CTL, the references are to the same entity. Symbolic names are used in locations of the syntax that are identified as SYMBOLNAME. Non-scan cells and connections to common entities use SYMBOLNAME's.
f) *Names in domains*: Names in two different domains could have a conflict. That is, the entity name in a domain could be defined differently than the same entity name in a different domain. When such situations arise, the domain name should be used in CTL to resolve the conflict. All names in domains shall be used as concatenation of the domain name and the name of the entity separated by

a double colon (::). The domain or the entity name can be double quoted or unquoted. Thus, the composite name shall be of the form “D”::”E”, “D”::E, D::”E”, D::E . In the case of global (nameless domain) blocks in CTL, the entities are identified with the entity name without any concatenation of the domain name with the entity name. When statements are sequential in nature (pattern data), the extended name can be avoided by defining the scope of the domain first before the entities are named. Similarly, unique entity names across all domains of its type can be optionally used without being qualified by the domain name it is defined within.

g) *Names in core instances*: All names in core instances shall be used as concatenation of the core instance name and the name of the entity separated by a colon (:). The core instance name or the entity name can be double quoted or unquoted. Thus, the composite name shall be of the form “I”:”E”, “I”:E, I:”E”, I:E. Core instance names can be hierarchical names. The name of a core instance of a core in the hierarchy would be referred to with a hierarchical concatenation of the names of the instances it is embedded in beginning with the highest level in the hierarchy. A colon (:) is used to separate the names in the hierarchy. Several entities defined in a core are Signals, SignalGroups, and ScanStructures. All of the entities follow the naming conventions globally across the CTL. When domain names are also used for entities associated with cores, then the complete name would take on the form I:D::E, where I refers to the core-instance name, D refers to the domain name, and E refers to the entity name.

h) *Order*: CTL information is to be formatted with the same rules specified by STIL. Forward references are allowed to patterns and those as a result of the AlternateTestMode statement.

i) USER_DEFINED: CTL allows for extendability through user-defined names. The user-defined name cannot conflict with the existing keywords used in CTL and should be consistent with the conventions set by IEEE Std 1450-1999. In almost all locations of the syntax that allows user-defined names, the name appears after a keyword namely User.

CTLMODE_NAME: A user-defined name that is consistent with the naming conventions of IEEE Std 1450-1999. This name is assumed to be unique across all names of CTLMode blocks defined in an environment it is specified in.

(3) **CoreInternal**: This statement identifies the beginning of the block in which the description of test information between the boundaries of embedded cores of the design is represented. Within this block, signals and signal-groups of cores are assigned keywords to describe test related information. Refer to “CoreInternal Syntax” for details.

(4) **DomainReferences**: This block identifies the domains that define the scope of information used in the current CTLMode block. Unnamed domains are always included for the CTLMode blocks. The identified block contents are defined in 1450.0, 1450.1, 1450.2, and this standard. The various domains that can be included in a CTLMode block are as follows:

— Category
— DCLevels
— DCSets
— MacroDefs
— Procedures
— Selector
— SignalGroups
— ScanStructures
— Timing
— Variables

Without domain references, any information in a named domain cannot be considered as part of the information of the current CTLMode block being defined. This rule holds for referenced entities in the scope of the current CTLMode block. For example, a Pattern referenced in the PatternInformation cannot use a Macro, or the Macro it invokes should not use a Waveform that is not in the scope of the CTLMode block.

Domains brought into the scope of the test mode can have entities with the same name. These entities are uniquely referenced in the test mode information by using the complete name that includes the domain name and the entity name [refer to item f) in this subclause]. When names do not clash, the domain name can be omitted. Information in patterns is sequential in nature, and the domain name can be omitted when the domain being used is identified before the entity name.

(5) **External**: This statement identifies the beginning of the block in which the description of the recommended test environment outside the design is specified. Within the External block, the signals and signal groups are assigned keywords to describe test-related information that the system integrator would consume for embedding the design into a larger design. Refer to Clause 15 for details.

(6) **Family**: This keyword associates the CTLMode block of information to a library (or set) of similar designs and their configurations.

NAME: It is a user-definable name that is consistent with the naming conventions of 1450.0.

(7) **Focus**: This keyword indicates information about the current configuration (mode) that is reflected in the details within the current CTLMode block. That is, there shall exist at least one instance of the information described in the focus statement in the remainder of the information in the scope of the current test mode. The Focus block is primarily used to indicate the configurations of subentities of the design that is being described. The TestMode statement indicates which cores are configured in internal test mode, external test mode, and so on. The PatternType statement indicates what type of patterns of the core are applied in the current mode. If no statement is used in the Focus block, then no focus can be interpreted for the associated CTL test mode (CTLMode block).

The parts of the design being accessed or stressed in the current configuration (i.e., the “focus”) are specified by parameters on the Focus statement. Parts of the design not being stressed can be identified as to their inactive state (i.e., Isolate and Bypass):

a) **Top**: This identifies the user-defined logic of the design outside of any cores that are integrated in the design.
b) **CoreInstance** CORE_INSTANCE_NAME: It identifies the instances of the cores that are incorporated in the design. This name shall be a valid core instance of the design.

(8) **PatternTypes** PATTERN_OR_BURST_ENUM: This statement is used within the Focus block to define the types of patterns that are applied in the current configuration for the corresponding sub-block of the design (Top or CORE_INSTANCE_NAME). The PATTERN_OR_BURST_ENUM is defined when pattern information is defined. The enumerated type that appears in this statement shall be consistant with information within the current mode. There shall exist at least one pattern or pattern burst of the identified type in the test mode as identified by the Purpose statement.

(9) **CTLMode** CORE_INSTANCE_NAME CTLMODE_NAME: This statement is used within the Focus block to name the CTLMode block in the original, non-integrated core’s CTL definition that establishes the mode. The statement identifies the core_instance and the name of the CTLMode block of the core that is defined. The CTLMODE_NAME shall be a valid name in the environment identified for the core instance. The test mode in this block shall be consistent with the test mode in this statement (statement 10). The use of this link is primarily for debug of this current test mode and its setup. It provides an ability to go back to the original test information that comes with the core.

(10) **TestMode** *test_mode_enum:* This statement is used within the Focus block to specify the test mode that applies to the identified entity (Top or CORE_INSTANCE_NAME). This statement is primarily used for cores whose configuration does not change during the test mode. If a core switches modes such that one of its modes is an InternalTest test mode, then the TestMode InternalTest shall be used.

(11) **Usage** *exec_enum*: This statement is used within the Focus block to define the application of the patterns within the current CTLMode block. There shall exist at least one PatternExec in the current test mode identified with the same *exec_enum* as appears in this statement:

a) **Production**: A configuration of the design that is created for production test.
b) **Diagnostic**: A configuration of the design that is created for diagnostics of a subset of the design.
c) **Characterization**: A configuration of the design that is created for characterization of the design.
d) **Verification**: A configuration of the design that is created for Verification of the design.
e) **User** USER_DEFINED: This represents a user-defined name that follows the guidelines specified by IEEE Std 1450-1999.

(12) **InheritCTLMode** CTLMODE_NAME: Through this construct, a CTLMode block can reuse information defined in another CTLMode block. Inheritance is the direct way of specifying such reuse. By default, the information in the unnamed CTLMode block is inherited by a named CTLMode block. The information redefined locally in the named CTLMode block overrides the information that is available through the unnamed CTLMode block. Through this statement, a chain of CTLMode blocks is constructed with the lowest priority information coming from the global block (unnamed CTLMode block), which is at the top of the chain. PatternInformation is never inherited. The CTLMODE_NAME shall be a valid name for a CTLMode block of statements in the scope of the current Environment.

There are two types of blocks with statements in CTLMode: blocks not associated with information on design entities (statements not on *sigref_expr* or *cellref_expr*) and blocks that are associated with information on design entities.

Some blocks like the Focus block or the DomainReferences block have statements that are not defined on *sigref_expr*'s or *cellref_expr*'s. These blocks are inherited in one piece. Any local definition of the complete block overrides all inherited information in the same block that was inherited. Inherited information is not overridden for blocks that are not redefined locally. PatternInformation is the only block of this type where the information cannot be inherited.

The Internal block, ScanInternal block and CoreInternal block have statements that apply attributes to *sigref_expr*'s and **c***ellref_expr*'s. Inheritance occurs on a statement-by-statement basis for information on any particular type of named entity. For example, if a DataType statement and an IsConnected statement are defined in a CTLMode block that is inherited, a local definition of the DataType on the same signal/signalgroup/cellname/cellgroup/core-signal/core-signalgroup as that of the inherited information will override the inherited information. The other inherited statements on the signal that are not locally defined continue to be active and are not overridden. Thus, in this example, the IsConnected statement is not overridden locally. Resolution occurs to the named entity where attributes propagate down to the individual references in that group. That is, if a signal is part of a named signal group, then the named signal group's attributes are relevant to the situation when the named signal group is used. The individual signals get a union of the attributes from information on the signal, and all information on the signal groups that the signal is part of. Inheritance rules treat the named signal groups as separate entities than the signals that form the group. The same is the true for other named groups (cells and core-signals). Groups created on the fly through the use of the "+" or "–" operator in the blocks of the environment are not considered named groups. Furthermore, any indexing to break a signal group into its subentities makes the entire set of entities in the expression resolve to the individual signals that make up the expression. Refer to the examples in 9.4.

(13) **Internal**: This statement identifies the beginning of a block in which the internal characteristics of the design, between the boundary of the design to which CTL is being written and the internal points where the connection is not blocked by scan cells or core-instances, are specified. Within an Internal block, every signal or signal group can be assigned keywords to describe the test characteristics of the design. Refer to Clause 10 for details.

(14) **PatternInformation**: This statement identifies the beginning of the block that defines information related to patterns (data and protocols). Within the PatternInformation block pattern, constructs such as

PatternExec, PatternBurst, Pattern, Procedures, and Macros are assigned keywords to specify the function they play in the test mode being described. Refer to Clause 16 for details.

(15) **Relation**: This statement begins the block in which relationships between signals and groups of signals are defined. Refer to Clause 13 for details.

(16) **ScanInternal**: This statement identifies the beginning of a block in which the internal characteristics of the design, bounded by internal scan cells of the design, are specified. Within a ScanInternal block, every scan signal associated with a cell on the scan chain can be assigned keywords to describe the test characteristics of the design. Refer to Clause 11 for details.

(17) **ScanRelation**: This statement identifies the beginning of a block in which the relationships between scan-cells or scan-cell-groups are defined. Refer to 14.2 for details.

(18) **TestMode**: A CTLMode block of information can be assigned a test mode that identifies the type of configuration of the design. The TestMode statement shall always be used except in the case of a global CTLMode block that is to be inherited only. The modes available are as follows:

a) **Normal**: The normal test mode type is used to define the configuration of the design in which it performs its functional behavior. For this configuration, CTL supports only the test relevant information. In most cases, the function of the design has very little to do with test, and not much test information can be provided for the configuration. Sometimes the function of the design is to assist in test of other design entities, for example, a test controller design. If the functional test mode of a design is test, the test mode cannot be classified as a Normal mode.
b) **Controller**: Some designs functions are test. Controller designs provide test stimulus to other designs and/or capturing response of other designs as their function. Designs that are only linear feedback shift registers are typically test-only functions. The test mode that defines the aspect of the design whose function is to actively assist in testing other designs external to it is identified by the Controller keyword.
c) **Transparent**: Describes configurations of the design that allow for a subset of the functional inputs of the design to be linked with a subset of functional outputs of the design such that a one-to-one relationship (combinational or sequential path) between the two subsets is created. This configuration allows for test paths to be created through the design to allow for testing logic external to the design.
d) **Bypass**: Describes configurations of the design that allow for a subset of the test inputs of the design to be linked with a subset of test outputs of the design such that a one-to-one relationship (combinational or sequential path) between the two subsets is created. This configuration allows for efficient methods of testing logic outside the design by shortening the lengths of the scan chains through the design.
e) **Quiet**: A configuration of the design that puts it in a state that minimizes the leakage current of the design. Thus, this test mode allows for current testing of logic outside the design in an embedded environment.
f) **PowerOn**: A configuration of the design that is achieved when the device is turned ON.
g) **PowerOff**: A configuration of the design that puts it in a state that minimizes power consumption.
h) **Sample**: A configuration of the design that allows the values of a subset of the inputs and/or outputs of the design to be sampled.
i) **InternalTest**: A configuration of the design that allows for internal logic of the design to be tested. Test patterns for the design would only exist in this mode.
j) **ExternalTest**: A configuration of the design that enables the testing of the logic external to the design. This configuration of the design allows for capture and launch capabilities for values outside the design to allow for shadow logic testing of the embedded design. The configuration is an outward-facing view of the design. This configuration assists the testing of entities external to the design with a passive involvement as opposed to the Control test mode, which has an active involvement.

k) **Isolate**: A configuration of the design that sets the inputs and the outputs of the design in a test mode that isolates the design from activity outside the design.
l) **PreLoad**: A configuration of the design that allows for the initialization of internal states or memories within the design.
m) **ForInheritOnly**: Information in CTLMode that does not constitute a complete test mode, but it is made available through the inheritance mechanism defined in CTL for the construction of other CTLMode blocks.
n) **User** USER_DEFINED: This represents a user-defined name that follows the guidelines specified by IEEE Std 1450-1999. This name is a test mode type that could augment the information available through the existing test mode types by specifying multiple enumeration types.

(19) **AlternateTestMode** CTLMODE_NAME: This statement identifies other CTLMode blocks of statements within the scope of the current environment that can be used as an alternative (in place of) for the current test mode. The CTLMODE_NAME shall be a valid name of a CTLMode block of statements in the scope of the current environment. This statement allows for the description of IEEE 1500 test modes that perform the same function through different interfaces (serial/parallel). This reference between modes is exempt from the ordering restriction of blocks. That is, CTLModes referenced by this statement need not be defined before the CTLMode in which this statement is used.

(20) **TestModeForWrapper**: Whereas the TestMode statement establishes the generic test mode, which applies to either a bare core or a wrapped core, this statement defines the specific identifier that is used to put the wrapped core into its test mode. The TestModeForWrapper statement shall be consistent with the Wrapper statements within Internal and External blocks and the TestMode statement. The reason that there are two separate statements is that the Wrapper internal and external connections will typically be placed in the global CTLMode block, and the TestModeForWrapper will typically be in a named CTLMode block because it is test mode specific.

WRAPPER_TEST_MODE: Whereas the TEST_MODE for a given CTLMode block defines the generic test mode (InternalTest, ExternalTest), the WRAPPER_TEST_MODE defines the specific identifier as defined by the wrapper standard document. This 1500 wrapper is defined in IEEE Std 1500-2005.

TEST_MODE_CODE: is a string of 1,0,X characters (with no spaces) that defines the code that must be written to the control register of the wrapper (e.g., the WIR in IEEE Std 1500-2005) to establish the mode. 1 is to denote a logic-1 or a ForceUp condition, 0 is to denote a logic-0 or a ForceDown condition, and X is to denote a “don’t care” condition in which the value does not matter. The values correspond to the cells of the corresponding instruction register such that the first value applies to the cell closest to the scan-out and the last value corresponds to the cell closest to the scan-in.

(21) **Vendor**: This keyword associates the CTLMode block of information with a list of vendors.

NAME: It is a user-definable name that is consistent with the naming conventions of IEEE Std 1450-1999.

(22) **Compliancy**: This statement indicates that the hardware structures on the periphery of the design defined by standards or user-implementations have been used. This keyword is a global statement that a wrapper exists in the design being described. The Internal block’s Wrapper statements on individual signals describes the exact use of the wrapper referred to here.

a) **IEEE1500** EXT_VERSION <**Wrapped** | **Unwrapped**>: It is used to signify compliance of the *sigref_expr* to the hardware structure defined by IEEE Std 1500-2005. The compliancy rules are defined in the associated standard document. The EXT_VERSION is the version of the standard as identified by IEEE Std 1500-2005. This number could contain a major and an optional minor revision number separated by a period. For example, 1.0 would be a valid EXT_VERSION. IEEE Std 1500-2005 has two levels of compliancy. The Unwrapped compliancy defines the core providers deliverable in CTL with minimal requirements on the logic design of the core. The Wrapped compliancy defines the core providers deliverable, which include a wrapper design and information provided in CTL. The details of the requirements are to be obtained from IEEE Std 1500-2005.

b) **None**: It is used to signify that no wrapper technology is associated with the design being described.

c) **User** USER_DEFINED: It is used to identify the use of a wrapper technology that is defined by another standard or a nonstandardized wrapper. Partial wrappers or IEEE 1500 variant wrappers that cannot be classified as Wrapped or Unwrapped would fall in this category.

9.4 CTLMode block syntax example

Depending on the information that needs to be described, different CTLMode statements will be used. It is impossible to describe all possibilities in examples. In this subclause, a high-level example is constructed that needs three configurations of the design to be described. These configurations are called mode1, mode2, and mode3. Thus, three CTLMode blocks of information are created. Assume that these are configurations of the design for internal testing, external testing, and a safe configuration, respectively. The TestMode statement is used to specify this information. In mode1, it is also assumed that some information is to be described on the signals of the design (Internal {}), some information on the patterns that are to be executed (PatternInformation {}), and some connectivity between scan chains of the design (ScanInternal {}) and some information on internal cores (CoreInternal {}) of the design are described. Similarly, mode2 and mode3 have different information that needs to be described, and the associated blocks are constructed.

```
Environment {
   CTLMode mode1 {
     TestMode InternalTest;
     Focus {
       PatternTypes Scan;
       Usage Production;
     }
     Internal {}
     CoreInternal {}
     ScanInternal {}
     PatternInformation {}
   }
   CTLMode mode2 {
     TestMode ExternalTest;
     Internal {}
     External {}
     PatternInformation {}
   }
   CTLMode mode3 {
     TestMode Isolate;
     Internal {}
     PatternInformation {}
   }
}
```

The example also shows the use of the Focus statement. During the internal testing of the design being described in CTL, which is done in the configuration mode1, only scan patterns are applied and the patterns are to be used for production testing of the implemented design.

Inheritance in CTL allows for reduction of repetition of information, which is the same in different test modes. Through the InheritCTLMode, statement information in one CTLMode block is made available in the other CTLMode block. The inherited information can be overridden in the local block. The nameless CTLMode block has information that is common to all test modes.

```
Signals { a In; b In; c In; }
SignalGroups { allsig[0..2] = ‘a+b+c’;}
Environment {
   // nameless block contains information common to all test modes.
   CTLMode {
     Relation {}
     Internal {
       // In this common test mode, signal “a” has a DataType that is the
       // union of the DataTypes defined in a{} and allsig[0..2]{}.
       // “a” also has the IsConnected statements as defined in a{}
       // associated with it. allsig[0..2] or allsig[0] only has the
       // datatypes as defined in the allsig[0..2]{}. The same mechanism
       // works for the information on “b” and “c”
       a { /* DataType and IsConnected statements.*/ }
       b { /* DataType and InputProperty statements. */ }
       c { /* DataType statement.*/ }
       allsig[0..2] { /* DataType statement a, b, c associated
                        when allsig[0..2] is referred to. */ }
     }
     CoreInternal {}
     ScanInternal {}
     PatternInformation{} //never inherited, test-mode independent info.
   }
   // mode2 gets the Relation statements from the nameless block.
   // PatternInformation is defined locally.
   // Internal and External block inheritance is on a named entity basis.
   // Internal block has the following information:
   //    a: DataType(from mode2), IsConnected(from CTLMode{})
   //       DataType information is union of a{} and allsig
   //    b: DataType(from CTLMode{}), InputProperty(from CTLMode{})
   //       DataType information is union of b{} and allsig
   //    c: DataType(from CTLMode{})
   //       DataType information is a union of c{} and allsig.
   //    allsig[0..2]: DataType(from mode2)
   // All CoreInternal and ScanInternal of CTLMode{} are inherited.
   // External block information is defined locally, nothing inherited.
   // PatternInformation is defined locally and is never inherited.
   CTLMode mode2 {
     TestMode ExternalTest;
     Internal {
       a { /* DataType statement */ }
       allsig { /* DataType statement */ }
     }
     External {}
     PatternInformation {}
   }
   // A chain of inherited information exists with the nameless block in
   // the beginning of the chain and mode3 at the end of the chain.
   // Information flows down the chain beginning with the nameless block.
   // Information in a test mode later in the chain overrides information
   // being inherited down the chain. The following information exists
   // in mode3.
   // Relation (from CTLMode{})
   // PatternInformation (from mode3, never inherited)
   // CoreInternal, ScanInternal (from CTLMode{})
   // External (all information from mode2)
   // Internal:
   //    a: DataType(from mode2), IsConnected(from CTLMode{})
   //       DataType information is the union of a{} and
   //       mode3-allsig[0..2]{}
   //    b: DataType(from CTLMode{}), InputProperty(from mode3)
   //       DataType information is the union of b{} and
   //       mode3-allsig[0..2]{}
   //    c: DataType(from CTLMode{}), InputProperty(from mode3)
   //       DataType information is a union of c{} and
   //       mode3-allsig[0..2]{}
   //    allsig[0..2]: DataType(from mode3)
   CTLMode mode3 {
     InheritCTLMode mode2;
     TestMode Isolate;
     Internal {
       allsig[0..2] { /* DataType statement */ }
       'b+c' { /* InputProperty statement */ }
     }
     PatternInformation {}
   }
}
```

The resolution of information on information on *sigref_expr*'s and *cellref_expr*'s can be seen in the example given earlier on inheritance of statements across modes. How a *sigref_expr*/*cellref_expr* is to be interpreted is described as follows:

```
Signals {a In; b In; c In; }
SignalGroups { allsig[0..2] = 'a+b+c';}
Environment { CTLMode { Internal {
   a { /* Info I */}
   b { /* Info II */ }
   c { /* Info III */ }
   allsig { /* Info IV */}
}}}
// Result: a (I, IV); b (II, IV); c (III, IV); allsig (IV)

Environment { CTLMode { Internal {
   a { /* Info I */ }
   'allsig - a' { /* Info V */}
}}}
// Result: a ( I ); b ( V ); c ( V ); allsig ( )
// 'allsig - a' is the same as 'b + c' is the same as 'allsig[1..2]'
```

10. CTLMode—Internal block

10.1 General

The Internal block defines attributes of all input/output signals of the core and the connections of the signals to internal parts of the core (scan cells, other cores, other signals). The intent of this information is to provide all information needed to incorporate the core into a larger design and to provide the necessary test application interface to the core such that a set of CTL patterns that are provided with the core (and which apply and observe values at the core primary input/output signals) can be applied.

10.2 Internal syntax

data_type_enum =
    < **Asynchronous**
    | **Synchronous**
    | **In**
    | **Out**
    | **InOut**
    | **Constant**
    | **TestMode**
    | **Unused**
    | **UnusedDuringTest**
    | **Functional**
    | **TestControl** (*testcontrol_subtype_enum*)*
    | **TestData** (*testdata_subtype_enum*)*
    | **User** USER_DEFINED >

testcontrol_subtype_enum =
        < **CaptureClock**
        | **CoreSelect**
        | **ClockEnable**
        | **InOutControl**
        | **Oscillator**
        | **OutDisable**
        | **OutEnable**
        | **MasterClock**
        | **MemoryRead**
        | **MemoryWrite**
        | **SlaveClock**
        | **Reset**
        | **ScanEnable**
        | **ScanMasterClock**
        | **ScanSlaveClock**
        | **TestAlgorithm**
        | **TestInterrupt**
        | **TestPortSelect**
        | **TestRun**
        | **TestWrapperControl** >

testdata_subtype_enum =
        < **MemoryAddress**

```
        ( Row
        | Column)*
    | MemoryData
    | Indicator
        ( TestDone
        | TestFail
        | TestInvalid)*
    | Regular
    | ScanDataIn
    | ScanDataOut>


cell_enum =
<<(WC_S | WH_S) <D|F># | WH_C |WH_CI> (_C<<I|O|B><I|O|U>|N>)(_U<D|F>)(_O)(<_G0|_G1>)
    | User USER_DEFINED >


ScanDataType_enum =
    < AddressGenerator
    | Boundary
    | Bypass
    | Counter
    | DataGenerator
    | Identification
    | Instruction
    | Internal
    | ResponseCompactor
    | User USER_DEFINED >

Environment { CTLMode (CTLMODE_NAME) {
    Internal {                                                          (1)

        ( sigref_expr {                                                 (2)

            ( DataType ( data_type_enum )+ ; )                          (3)
            ( DataType ( data_type_enum )+ {
                ( ActiveState                                           (4)
                    < ForceDown
                    | ForceUp
                    | ForceOff
                    | ForceValid
                    | ExpectLow
                    | ExpectHigh
                    | ExpectOff
                    | ExpectValid > (Weak); )
                ( DataRateForProtocol <Average|Maximum> INTEGER;)       (5)
                ( AssumedInitialState                                   (6)
                    < ForceDown
                    | ForceUp
                    | ForceOff
                    | ForceValid
                    | ExpectLow
                    | ExpectHigh
                    | ExpectOff
                    | **ExpectValid** > (**Weak**) ; )
                ( **ScanDataType** ( *ScanDataType_enum* )+ ; )                                  (7)
                ( **ValueRange** INTEGER INTEGER (CORE_INSTANCE_NAME)+; )*                       (8)
                ( **UnusedRange** INTEGER INTEGER;)*                                             (9)
            } )* *// end DataType*

            ( **DisableState**                                                                  (10)
                < __ExpectOff__
                | **ExpectLow**
                | **ExpectHigh**
                | **ExpectValid**
                > ( **Weak** ) ; )

            ( **DriveRequirements** {                                                           (11)
                ( **TimingNonSensitive**; )
                ( **TimingSensitive** {
                    ( **Period** < **Min** | **Max** > *time_expr; )**
                    ( **Pulse** < **High** | **Low** > < **Min** | **Max** > *time_expr; )**
                    ( **Precision** *time_expr; )*
                    ( **EarliestTime** *time_expr; )*
                    ( **LatestTime** *time_expr; )*
                    ( **Reference** *sigref_expr* {
                        ( **SelfEdge** < **Leading** | **Trailing** | **LeadingTrailing** > (INTEGER); )
                        ( **ReferenceEdge** < **Leading** | **Trailing** | **LeadingTrailing** > (INTEGER); )
                        ( **Hold** *time_expr; )*
                        ( **Setup** *time_expr; )*
                        ( **Period** *real_expr*;)
                        ( **Precision** *time_expr; )*
                    } )* *// end Reference*
                    (**Waveform**;)
                } ) *// end TimingSensitive*
            } ) *// end DriveRequirements*

            ( **ElectricalProperty**                                                            (12)
                < __Digital__
                | **Analog**
                | **Power**
                | **Ground**
                > (ELECTRICAL_PROPERTY_ID) ; )

            ( **InputProperty**                                                                 (13)
                ( < **Edge**
                | **GlitchFree**
                | **PullDown**
                | **PullUp**
                | **ScanStable**
                | **ScanUnstable**
                | **SynchFF**
                | **SynchLatch**
                | **Transitions** (INTEGER)
                | **Window**
                | **User** USER_DEFINED
                > )+ ; )
```

```
( IsConnected <In|Out> (<Direct | Indirect>) {                                              (14)
   ( CoreSignal (sigref_expr) ; )                                                           (15)
   ( IsGatedBy <LogicAnd | LogicOr | LogicXor> logic_expr {                                 (16)
      ( LOGICSIGNAME {
         Type < Signal | StateElement Scan | StateElement NonScan | CoreSignal >;
         Name <SIGNAME | CELLNAME >;
      } )+ // end logicsigname
   } )* // end IsGatedBy
   ( IsGatedBy < Macro | Procedure > NAME ; )*
   ( TestAccess (<                                                                          (17)
      Control
      | Observe
      | User USER_DEFINED >)+
      < Macro | Procedure > NAME; )*
   ( < LaunchClock | CaptureClock | Reset > SIGNAME {                                       (18)
      ( <LeadingEdge | TrailingEdge> (LevelSensitive); )                                    (19)
      ( <Direct | Indirect>; )                                                              (20)
      ( StateAfterEvent <                                                                   (21)
         Connection
         | ExpectLow
         | ExpectHigh
         | ExpectUnknown
         | ExpectValid
         | Hold
         | Invert
         | ShiftState
         | User USER_DEFINED > ; )
   })* // end LaunchClock
   ( Signal (sigref_expr) ; )                                                               (22)
   ( StateElement <Scan | NonScan> (cellref_expr) ; )                                       (23)
   ( Transform {                                                                            (24)
      ( WaveformTable (WFT_NAME)+; )
      ( Invert ; )
      ( WFCMap FROM_WFC -> TO_WFC; )*
      ( DelayCycles INTEGER;)
   })* // end Transform

   ( Wrapper <IEEE1500 | None | User USER_DEFINED > ( CellID cell_enum) ; )                 (25)

} )* // end IsConnected

( IsDisabledBy <In|Out> Logic logic_expr {                                                  (26)
      ( LOGICSIGNAME {
         Type < Signal | StateElement Scan | StateElement NonScan | CoreSignal >;
         Name <SIGNAME | CELLNAME > ;
      })+
} )* // end IsDisabledBy
( IsDisabledBy <In|Out> < Macro | Procedure > NAME ; )*

( < LaunchClock | CaptureClock > SIGNAME {                                                  (27)
   ( <LeadingEdge | TrailingEdge> (LevelSensitive);)
})* // end LaunchClock

( OutputProperty                                                                            (28)
                ( < Edge
                | PullDown
                | PullUp
                | ScanStable
                | ScanUnstable
                | SynchFF
                | SynchLatch
                | ThreeState
                | TwoState0Z
                | TwoState1Z
                | Transition INTEGER
                | Window
                | User USER_DEFINED > )+ ;)

            ( StrobeRequirements {                                                      (29)
                ( TimingNonSensitive; )
                ( TimingSensitive {
                    ( Precision time_expr; )
                    ( EarliestTimeValid time_expr; )
                    ( LatestTimeValid time_expr; )
                    ( EarliestChange time_expr; )
                    ( Reference sigref_expr {
                        ( SelfEdge < Leading | Trailing | LeadingTrailing > INTEGER; )
                        ( ReferenceEdge < Leading | Trailing | LeadingTrailing > INTEGER; )
                        ( EarliestTimeValid time_expr; )
                        ( LatestTimeValid time_expr; )
                        ( EarliestChange time_expr;)
                        ( Precision time_expr; )
                    } )* // end Reference
                    (Waveform;)
                } ) // end TimingSensitive
            } ) // end StrobeRequirements

            ( ScanStyle < MultiplexedData | MultiplexedClock > (LevelSensitive) ; )     (30)

            ( Wrapper <IEEE1500 | None | User USER_DEFINED > ( PinID USER_DEFINED_PIN_ID ) ; )  (31)
        })+ // end sigref_expr
    } // end Internal
}} // end Environment, CTLMode
```

10.3 Internal block syntax descriptions

(1) **Internal**: This statement begins the internal block within a CTLMode block. The internal block contains statements that describe the internals of the design (inwards from the Signals), which is to be considered as a statement of facts about the test aspects of the design.

(2) *sigref_expr*: This is a signal or group of signals or an expression combining the first two entities. In the internal block, statements assign properties to the signals or signal groups that are part of the *sigref_expr*. Multiple instantiations of the same *sigref_expr* in a single internal block is not allowed. Resolution of information is down to individual signals that get the union of information from the blocks of information on the signal and the blocks on named signal groups in which the signal is part of. For example, in the same block of statements, one can see information on signal *mysignal* and information on signal group

allsignals[0..10], which includes *mysignal*. *mysignal* gets the combined information from the two blocks of information. *sigref_expr*'s, which break any signal group into its subentities by using the "-" or bit-indexing capability ([]), make the *sigref_expr* resolve to the individual signals that make up the resulting *sigref_expr*. Refer to the definition and examples of the InheritCTLMode statement for more details on interpreting *sigref_expr*'s and association of information to the entities in the *sigref_expr* (9.2). The exact definition of *sigref_expr* should be taken from IEEE Std 1450-1999 and its extension in this standard. The CTL provider shall ensure that signal information statements are consistent with each other when information is provided on a signal and when the signal is also part of a named signal group in the same block.

(3) **DataType**: This statement is used to define the behavior of the signals on which the DataType keyword is defined. The information in DataType shall be consistent with the way the signal is used in the test mode. Several statements such as the ActiveState are associated with the DataType. The values specified by these statements are expected to be consistent with the use of the associated signals. A mismatch between the information in this statement and reality is an error condition that may not be detected until the tests are being mapped to tester hardware. Multiple behaviors can be specified for any signal or set of signals (*sigref_expr*). All identified behaviors are simultaneously displayed by the signals when part of the same statement. If no DataType keyword is defined for a signal, the default of Unused shall be assumed. Multiple data types can be used to describe a signal. Some keywords are specializations of other generic keywords. For example, ScanDataIn is a specialization of TestData. If a subcategory keyword is used, the generic category is required to be specified. Multiple subcategory keywords can be used for the same generic category, as follows:

a) **Asynchronous**: A signal is asynchronous if its timing is not cyclized to a period.
b) **Synchronous**: A signal is synchronous when its timing is cyclized to a period.
c) **In**: It indicates that the signal is being used only as an input. An error condition arises if this keyword is used for signals that are defined as Out.
d) **Out**: It indicates that a signal is being used only as in output. An error condition arises if this keyword is used for signals defined as In.
e) **InOut**: It indicates that a signal is being used as input and output of the design. An error condition arises when the signal is defined as an In or Out.
f) **Constant**: An input or output signal that is constant during the CTLMode, including the establish mode and the terminate test mode phase. If the signal is not at the value specified before the beginning of the EstablishMode sequence, the signal is expected to be set within the first clock cycle of the EstablishMode sequence. The stability of this signal is critical for the validity of the test data in the test mode when this attribute is used for an input signal. If test data exist for the test mode, it is assumed to be consistent with this definition.
g) **TestMode**: This keyword is used to describe signals that are constant once the test mode is established until the beginning of the terminate test mode sequence. When specified on an input signal, the stability of this signal is critical for the validity of the test data in the mode. Output signals that remain stable after the test mode has been established until the beginning of the terminate test mode sequence can also be identified by this keyword. If test patterns exist for the CTLMode, it is assumed the test patterns treat the associated signal as per this definition.
h) **Unused**: A signal that is not used. Such signals are typically disconnected from the internal logic of the design in the current configuration of the design. These signals are not used by the establish test mode sequences and the terminate test mode sequences in addition to the operations performed in the test mode. This keyword is also used to identify redundant inputs/outputs that may be active in other configurations.
i) **UnusedDuringTest**: A signal that is not used for testing the core or the surrounding logic. This signal is unused only during the part of the test mode after the establish test mode sequence and before the terminate test mode sequence.
j) **Functional**: A signal that does not have a test function.
k) **TestControl**: A signal that controls some configuration of a test mode that typically needs to be switched infrequently during a test. Examples of such signals include enabling or disabling signals. Clocks also fall in this category. Tests in the test mode are expected to be using the associated

signals consistent with this definition. If tests do not exist in the test mode, then the description to be assumed by this statement is the intent of use by the tests.

l) **TestData**: Signals that are applied a stimulus by the test patterns such that the signal cannot be classified as TestMode or TestControl signals. The characteristic represents a fact when test data exist for the test mode, and it represents intent when the test data do not exist for the test mode.
m) **User** USER_DEFINED: A user-defined data type for extendability of CTL. One should not use this data type for describing signal characteristics already covered by other data types. The user-defined name is expected to follow the naming rules for names as defined by IEEE Std 1450-1999.
n) **CaptureClock**: A signal used as a clock for the design for test operations to capture values in memory elements through the non-scan paths. *Note*: This signal might not be considered a clock when using a different configuration (mode) of operation.
o) **CoreSelect**: A special type of control signal that determines whether the design is active and/or should participate in the patterns being applied.
p) **ClockEnable**: A signal that controls the gating of another signal that is identified to be a clock.
q) **InOutControl**: A signal that provides control to select the direction of one or more bidirectional signals.
r) **Oscillator**: A free running clock signal is identified as an Oscillator.
s) **OutDisable**: A signal of the design that controls the disabling of some set of outputs of the design such that the output is isolated from the internal logic of the design.
t) **OutEnable**: A signal of the design that enables a set of outputs of the design to be driven by values from the logic internal to the design.
u) **MasterClock**: A signal of the design that clocks non-scan data into single memory elements or into the master latch of a dual memory element.
v) **MemoryRead**: A signal that controls the Read input of a memory.
w) **MemoryWrite**: A signal that controls the Write input of a memory.
x) **SlaveClock**: A signal of the design that clocks non-scan data into a slave latch of a dual memory element.
y) **Reset**: A signal that changes the state of certain memory elements of the design to a predefined state. Set, Reset, and Clear signals of designs are all Reset signals in CTL.
z) **ScanEnable**: A signal that is not a test mode signal or a clock or a scan-in or scan-out of the design that controls the configuration of the scan chain.
aa) **ScanMasterClock**: A signal of the design that clocks scan data into single memory elements or into the master latch of a dual memory element.
ab) **ScanSlaveClock**: A signal of the design that clocks scan data into a slave latch of a dual memory element.
ac) **TestAlgorithm**: A signal that selects which test algorithm will be applied. This keyword is most appropriate for memory test methods where algorithmic patterns are applied.
ad) **TestInterrupt**: A signal that interrupts the operation of some test. For example, it could be a signal that interrupts the running of a built-in test algorithm.
ae) **TestPortSelect**: A signal that selects which of several possible test ports will be used. These ports are the same ports as that used in the Relation block of statements. Memories typically have multiple ports that may be activated with test port select signals.
af) **TestRun**: A signal that is used to trigger the execution of tests.
ag) **TestWrapperControl**: A signal that is used to control the operation of a test wrapper.
ah) **MemoryAddress**: The signal that is connected to the address signals of a memory. MemoryAddress can be further identified as Row and Column address.
ai) **Row**: The subset of the memory address that selects the topological row of the memory.
aj) **Column**: The subset of the memory address that selects the topological column of the memory.
ak) **MemoryData**: A signal that is connected to the data signals of a memory.
al) **Indicator**: Indicator is a special type of TestData signal that indicates the occurrence of a special event. TestFail and TestDone are special types of indicator signals.
am) **TestDone**: A signal that is used to indicate the completion of a test activity. In most examples, this would indicate the completion of some kind of BIST activity.

an) **TestFail**: A signal that is used to indicate that a failure occurred during a test activity. In most examples, this would be related to some kind of BIST activity.
ao) **TestInvalid**: A signal that can indicate that the current test results are invalid and that the test should be reapplied.
ap) **Regular**: This keyword describes the regularity in the test data that is applied to the signal. Regular test data are predictable. For example, it could be pseudo-random in nature or algorithmic.
aq) **ScanDataIn**: Signals that have or are intended to have test data that are serialized to load values into scan chains.
ar) **ScanDataOut**: Signals that have or are intended to have test data that are serialized to unload values from scan chains.

(4) **ActiveState**: This statement is used to specify the active value for a signal that is consistent with the data type of the signal. For example, if a signal is identified as a ScanEnable, then an ActiveState of ForceUp would mean that the scan chain enabled by the signal is in scan configuration when the signal is a logic “1”. Force* states are used for inputs to describe drive values, and Expect* states are used for outputs to describe expected values. The keyword can be replaced by the associated symbol for the same meaning. For example, ForceDown and D used for the ActiveState would have the same meaning:
a) **ForceDown** (D): Logic 0.
b) **ForceUp** (U): Logic 1.
c) **ForceOff** (Z): High impedance state.
d) **ForceValid** (N): Logic 1 or Logic 0.
e) **ExpectLow** (R): Logic 0.
f) **ExpectHigh** (G): Logic 1.
g) **ExpectOff** (Q): High impedance state.
h) **ExpectValid** : Logic 0 or Logic 1.

Weak: This signifies a low strength for the value being driven. Weak values can be overridden by strong values applied to the same signal from another source.

The ActiveState statement may be meaningless with certain data types. Although the syntax allows this, the use of the combinations are meaningless. For example, clocks do not have active states, and the synchronous keyword does not have an active state. Similarly, MemoryAddress and MemoryData also do not have any meaning for an ActiveState.

(5) **DataRateForProtocol**: This statement is used to specify the expected activity on the associated signal or signal group during the application of the test protocols in the current CTLMode block. The data rate is specifiable as the maximum number of events on the signal outside of the scan data for the protocols in the test mode. All protocols need not have the same number of values applied to signals. The average number of events on the signal outside of the scan data is also specifiable. No information on the data rate can be interpreted when information is not provided by this statement:
a) **Maximum** INTEGER: The expected maximum number of times a signal changes value per protocol for the patterns that are applied in the current configuration of the design.
b) **Average** INTEGER: The average number of times a signal changes value per protocol for the patterns that are applied in the current configuration of the design.

(6) **AssumedInitialState**: This statement is used to specify the expected state for a signal at the start of every test protocol of the test mode. Specifically, this is the state expected to appear on a signal at the end of execution of every protocol. For example, if a signal is identified as a clock signal, then an AssumedInitialState of ForceDown would mean that the stable state of the clock in the current test mode is a logic-0. This clock-OFF state is assumed by the protocols of the test mode. The keyword can be replaced by the associated symbol for the same meaning. For example, ForceDown and D used for the AssumedInitialState would have the same meaning:
a) **ForceDown** (D): Logic 0.
b) **ForceUp** (U): Logic 1.

c) **ForceOff** (Z): High impedance state.
d) **ForceValid** (N): Logic 1 or Logic 0.
e) **ExpectLow** (R): Logic 0.
f) **ExpectHigh** (G): Logic 1.
g) **ExpectOff** (Q): High impedance state.
h) **ExpectValid** : Logic 0 or Logic 1.

Weak: This signifies a low strength for the value being driven. Weak values can be overridden by strong values applied to the same signal from another source.

(7) **ScanDataType**: This statement is used to describe the type of scan data that is associated with the signal. The signal is required to be defined as a scan terminal (ScanDataIn or ScanDataOut) in the current mode. The data appearing on the signal is an unordered list of the following type of scan values:

a) **Internal**: Scan chains internal to the design.
b) **Boundary**: Scan chains at the boundary of the design.
c) **Instruction**: The scan chain that operates the instruction register that configures the design in different modes based on its state.
d) **DataGenerator**: The scan chain that accesses the state machine used to supply test patterns to the design in test methods that generate the stimulus internal to the design.
e) **ResponseCompactor**: The scan chain that accesses the state machine used to compact the response of test patterns that are applied to the design.
f) **Counter**: The scan chain that accesses a counter in the design used to monitor the number of tests being applied to the design.
g) **AddressGenerator**: The scan chain that accesses a state machine that supplies test patterns to the address inputs of a memory internal to the design.
h) **Bypass**: The scan chain that bypasses the scan chains of the design.
i) **Identification**: The scannable register that holds a unique code for the device that is an identifier for it.
j) **User** USER_DEFINED: A user-defined scan data type for extendability of CTL. It is not allowed to use this capability to describe scan chains types already covered by other keywords allowed in this statement. The user-defined name cannot be used to redefine any of the existing ScanDataType keywords. The USER_DEFINED name shall follow the definitions for user-defined names as defined by IEEE Std 1450-1999.

When this statement is used in conjuction with *cellref_expr*'s as part of ScanInternal, the information relates to the type of data in the associated scan cell. The cells identified in the *cellref_expr* are required to be partly defined as scan cells in scan chains active in the test mode. As no scan terminal exists in this use of the statement, there is no associated requirement of the DataType for the statement.

(8) **ValueRange** INTEGER INTEGER CORE_INSTANCE_NAME: This statement defines the allowed data value range for a group of signals. This is typically used when the signal group is a bus, such as an address or data bus. The left-most signal in the associated *sigref_expr* is the MSB, and the right-most signal is LSB. The integers that follow describe the decimal equivalent of the valid addresses as they correspond to the MSB-LSB order of the signals. The first integer represents the lower end of the range, and the second integer represents the higher end of the range. Multiple value ranges are specifiable to cover segmented ranges. The ranges specified by the multiple statements are required to be non-overlapping ranges. When a (or multiple) core instance name (CORE_INSTANCE_NAME) is used, the values taken on by the signals as defined by the ValueRange are to be associated with the core instance. For example, if the ValuesRange is specified for CoreSelect type of inputs, the values taken on by the signals defined by this statement identify which core is being selected if the CORE_INSTANCE_NAME is given.

(9) **UnusedRange** INTEGER INTEGER: This statement allows the definition of unused addresses in the allowed data value range specified by the ValueRange for a group of signals. This is typically used when the signal group is a bus, such as an address or data bus. By default, the left-most signal is the MSB and the right-most signal is LSB. The integers that follow describe the decimal equivalent of the valid addresses as

they correspond to the MSB-LSB order of the signals. The first integer represents the lower end of the range, and the second integer represents the higher end of the range. Multiple value ranges are specifiable to cover segmented ranges. The ranges specified by the multiple statements are required to be non-overlapping ranges.

(10) **DisableState**: This statement is coupled to the IsDisabledBy statement. This construct describes the value that would appear on the disabled signal that is referred to in the current scope of this statement. The following values are supported:

a) **ExpectHigh** (G): Logic 1. G can be used in place of the term ExpectHigh.
b) **ExpectLow** (R): Logic 0. R can be used in place of the term ExpectLow.
c) **ExpectValid**: Logic 1 or Logic 0.
d) **ExpectOff** (Q): High impedance state. Q can be used in place of the term ExpectOff.

Weak: This signifies a low strength for the value being driven. Weak values can be overridden by strong values applied to the same signal from another source.

(11) **DriveRequirements**: This statement is used to specify the required timing for input signals of the design. Through this statement, the flexibility of changing the timing of signals is described. The waveforms available in the test mode represent a valid example of the timings allowed on the signals of the design. If the **DriveRequirements** are not specified for a signal, **TimingNonSensitive** is to be assumed as the default condition:

a) **TimingNonSensitive**: This keyword expresses the fact that the signal has flexible timing requirements that allow for the waveform edge(s) for the associated signals to be moved further apart from preceding or succeeding signal edges without fear of pattern or protocol failures.
b) **TimingSensitive**: This keyword expresses the fact that the signal has specific timing requirements that the waveform edge(s) of the associated signals cannot be arbitrarily changed without affecting the validity of the patterns and protocols in the scope of this statement.
    1) **Period**: This statement denotes either requirement for a minimum (**Min**) or maximum (**Max**) clock period for the associated signal. This keyword is directly associated with periodic clock signals. Multiple statements of Period can be used to describe minimum and maximum constraints.
    2) **Pulse**: This statement denotes either the minimum (**Min**) or the maximum (**Max**) values for pulses on the signals. The information is describable for the low (**Low**) and high (**High**) states of the signal over multiple statements to ensure proper duty cycles.
    3) **Precision**: Through this statement the precision/resolution of the timings is specified. The time_expr represents a value that represents a plus and a minus of the timing information. All timing specified of the associated signal in the scope of this statement is expected to have the precision defined here.
    4) **EarliestTime**: This statement describes the earliest time within a test cycle (period) that the signal may transition. The time is relative to the start of the test cycle.
    5) **LatestTime**: This statement describes the latest time within a test cycle (period) that the signal may transition. The time is relative to the end of the test cycle.
    6) **Reference** *sigref_expr*: This statement defines specific signals in a *sigref_expr* for which the associated signals of the Internal statement have a timing relationship that falls under the category of **TimingSensitive**.

SelfEdge <**Leading**|**Trailing**|**LeadingTrailing**> INTEGER: This statement denotes which edges of the associated signals for this Internal statement have a timing sensitivity to the signals listed in the **Reference** statement. Leading, Trailing, or LeadingTrailing edges can be specified to identify the type of reference edge. If no integer is specified, all edges of the specified type are assumed to have the timing requirements. Every waveform defined for the signal has a state at the beginning of the period. The leading edge is defined as the edge that takes the signal from the starting state at the beginning of the period to the other logic level. Trailing edges are the opposite transitions to the leading edges. By specifying an integer, individual edges can be identified for the timing relationship. The integer refers to the *n*th edge of the identified type that is relative to the beginning of the test cycle (period). LeadingTrailing is used to identify the condition represented by either a Leading or a Trailing edge. If the SelfEdge statement is not specified, all events of the signal in its waveforms are to be assumed as the default.

ReferenceEdge <**Leading**|**Trailing**|**LeadingTrailing**> INTEGER: This statement identifies which edges of the Reference signals waveforms are the reference edges for the timing sensitivity. The interpretation of Leading, Trailing, and LeadingTrailing are similar to that defined for the SelfEdge statement. Similarly, the integer defines the nth edge of the identified type that is relative to the beginning of the test cycle (period). If a reference edge is not defined, all events of the signal in its waveforms are to be assumed as the default.

Hold: Specifies the hold time (through a time_expr) between the edges of the signal identified by the SelfEdge and the ReferenceEdge statement. The hold time represents a condition where the SelfEdge occurs after the ReferenceEdge.

Setup: Specifies the setup time (through a time_expr) between the edges of the signal identified by the SelfEdge and the ReferenceEdge statement. The setup time represents a condition where the SelfEdge occurs before the ReferenceEdge.

Period: Specifies the relationship (through a real_expr) between the period of the signal(s) on which the statement is defined as it relates to the period of the reference signal(s) in this statement. The real number specified in the real_expr multiplied by the period of the reference signal is required to be a valid value in the Period – Min/Max range specified for the signal.

Precision: This statement is analogous to the Precision of individual edges defined within the DriveRequirements block of statements. The precision defined by this statement denotes the precision/resolution of the timings between edges.

7) **Waveform**: This statement defines the condition that the timing of the signals is to be determined by the WaveformTables in the scope of the current statement (test mode). The timing cannot be changed outside of that specified by the waveforms.

(12) **ElectricalProperty**: This statement describes the electrical characteristics of the signal or set of signals in its scope. If not specified, the signal is assumed to be Digital:

a) **Digital**: Values that are interpreted to be discrete levels.
b) **Analog**: Values that are interpreted as contiguous values.
c) **Power**: Power level.
d) **Ground**: Ground level.

ELECTRICAL_PROPERTY_ID: This represents a name that allows for separating the different signals with the same ElectricalProperty into subcategories. The name used should be a valid name under the rules specified by 1450.0.

(13) **InputProperty**: Several characteristics can be defined on the associated input signal or group of signals through this statement. If not specified, no input property can be assumed:

a) **Edge**: This characteristic is used to describe the way a clock is used in the design. The clock with this keyword denotes the fact that the clock feeds a set of edge-triggered memory elements in the design.
b) **Window**: This characteristic is used to describe the way a clock is used in the design. The clock with this keyword denotes the fact that the clock feeds a set of level sensitive memory elements in the design.
c) **GlitchFree**: This keyword is used to describe that glitches or spikes must be prevented on the signal to ensure the expected behavior (for example, an asynchronous reset signal may need to be identified with this keyword).
d) **Transitions** INTEGER: This keyword is used to describe the need for critical transitions (required by the test data) at the associated signal. With an integer value, the number of critical transitions can be specified. If an integer is not specified with this keyword, the default value to be assumed is 1. The integer can take on values greater than or equal to 0. If the keyword Transition is not specified, then the default number of transitions is 0.
e) **PullUp**: This describes the existence of a PullUp device on a tristateable signal.
f) **PullDown**: This describes the existence of a PullDown device on a tristateable signal.
g) **SynchLatch**: This describes the existence of a latch in the design connected to the associated signal that is used to synchronize the timing of the signal.
h) **SynchFF**: This describes the existence of a flip-flop in the design connected to the associated signal that is used to synchronize the timing of the signal.
i) **ScanStable**: This denotes the fact that the signals of the design are held to a certain value during scan. On an input of the design, it specifies the need for the stable condition when the scan is performed external to the design in an embedded environment. On an output, it specifies the fact that the output is stable during the scan operation internal to the design.
j) **ScanUnstable**: This statement is analogous to the ScanStable in its definition with the focus of describing the instability of the associated signals during a scan operation. This keyword denotes the fact that the signals are changing values during scan.
k) **User** USER_DEFINED: A userdefined input property for extendability of CTL. It is not allowed to use this capability for describing signal characteristics already covered by other input properties. The user-defined name cannot be used to redefine any existing input property keywords. The user_defined name shall follow the definitions for user defined names as defined by IEEE Std 1450-1999.

(14) **IsConnected**: This block of statements describes connectivity information about signals. Connections are defined between two endpoints that must exist. That is, an IsConnected statement cannot be used without an endpoint description within the block. For a Direct type of connection, a value at the end of the connection is sensitized to reach the other end of the connection when the Gating Condition of the connection resolves to a logic-1 for a LogicAnd gating, resolves to a logic-0 for a LogicOr gating, and is always sensitized for a LogicXor gating. The IsConnected statement is an explicit definition of the connectivity between two points. If the IsConnected Statement is not used, and the signal is a ScanDataIn or ScanDataOut type of a signal, then implicit connections from the signals to the scan cells can be determined from the scan structures active in the test mode. The Enabling condition defined in the scan chain definition provides the necessary information for the gating condition of the connection. The syntax allows for the definition of an explicit IsConnected statement for the special case of the scan connection. The information available through this statement falls under the following categories:

A connection between the signal of the design and an internal scan cell, another signal of the design, an internal signal of the design that is defined in a CoreType block. If no connection point is defined within the connection block, an implicit connection is assumed to be present if the *sigref_expr* is a scan input or scan output. The implicit connection is defined in the scan structures statement. **In** and **Out** are used to describe the direction of the connection relative to the signal on which the information is being specified. "**In**" is used to describe a connection where the *sigref_expr* on which the IsConnected statement is used is an input signal (source of the connection), and "**Out**" is used to describe a connection where it is an output (sink of the connection). Multiple simultaneous connections are specified through multiple instantiations of the IsConnected statement within the block for the signal being referenced. If no IsConnected statement exists, no internal connection is to be assumed for the associated signal.

Direct: The default condition when not specified defines the two ends of the connection to be linked by a sensitized path where a value when applied at the input side of the connection would appear at the output side of the connection. The value can go through a transformation.

Indirect: Defines the existence of a nonsensitized combinational path between the two ends of the connection being described.

(15) **CoreSignal** *sigref_expr*: The connection of the *sigref_expr* in the scope of the statement is connected to internal signals defined by the CoreType block. The core_name and the signals referred to by this statement should exist in the scope of the current CTLMode block. The format of the names in *sigref_expr* is expected to be corename:(domainname::)signame. The following possibilities exist for the definition of the connection between the signals in the *sigref_expr* and the cells in the *cellname_list*:

a) One-to-one correspondence: The number of signals in the *sigref_expr* are the same as the number of core-signals. In this case the first signal is defined to be connected to the first core-signal, the second signal is defined to be connected to the second core-signal, and so on.
b) One-to-many correspondence: In this case, one signal is defined in the *sigref_expr* and more than one core-signal. In this case, the signal is assumed to be connected to all core-signals.
c) Many-to-one correspondence: In this case, many signals are defined and only core-signal is defined. In this case, all signals are defined to be connected to the core-signal.

(16) **IsGatedBy <LogicAnd | LogicOr | LogicXor>** *logic_expr*: Through this statement the gating condition for the connection in the scope of this statement is specified. Through a *LogicAnd*, *LogicOr*, and *LogicXor,* the gating condition of the connection can be described to be an AND gate, an OR gate, and an Exclusive-Or gate, respectively. The gating logic (which does not include the path that is described by the connection) that controls the gating condition is described through the *logic_expr*. The logic expression with locally defined LOGICSIGNAMES are used to define the boolean relationship between the gating signals defined within the IsGatedBy block. For example, a logic expression could be defined as “A&B|~C”, where A, B, and C are symbols whose binding is defined within the IsGatedBy block. The gating signals could be signals, scan signals of the design, non-scan symbolic named entities, or signals of core instances. Positive logic is used to define the gating condition. Information that does not allow for the complete evaluation of the gating logic expression is an error condition.

logic_expr: The logic expression is used to define the boolean relationship between symbolic logic signal names. For example, a logic expression could be defined as A&B|~C, where A, B, and C are symbols whose binding is defined within the IsGatedBy block. For details of the operators allowed in logic_expr, refer to the specifications in IEEE Std 1450.1-2005. The logic_expr is a quoted expression using single quotes.

LOGICSIGNAME: A locally defined name for use in the logic expression. The logicsigname should follow the naming rules for names defined in IEEE Std 1450-1999. The logicsigname begins a block that can contain the following statements:

Type: This statement is used to define the referenced element and shall be one of the following:

a) Signal: A valid signal of the design as defined in the Signals block of statements.
b) StateElement Scan: A valid scan cell name as defined in the Scan Structures block. The scan cell could be associated with a core instance. Scan cells are state elements that are part of a scan chain defined in a Scan Structure block of statements.
c) StateElement NonScan: A state element of the design that is identifiable with a unique name that can be used in a *cellref_expr*. The name cannot be the same as a scan cell name in any mode.
d) CoreSignal: A valid signal of a core instance.

Name: The name of the element as appropriate to the Type. That is, a signal will be named in the Signals or SignalGroups block, a StateElement Scan will be named in the ScanStructures block, and a CoreSignal will be defined in the CoreType block. NonScan StateElement is identified using a SYMBOLNAME. A symbolic name is a means of communicating a global point in the environment of the design that is to be considered unique across all designs in the environment. The SYMBOLNAME should follow the user-defined naming conventions for signals defined by IEEE Std 1450-1999.

IsGatedBy <**Macro | Procedure**> NAME: The enabling condition of a connection can be specified as the result of the execution of a macro or procedure with this statement. As a result of the execution of the sequence, the two ends of the connection are sensitized. A macro or procedure is an arbitrary sequence of events defined at externally accessible points of the design. The macro or procedure should be part of a MacroDefs or Procedures block that is in the scope of the current CTLMode block.

(17) **TestAccess <Control | Observe | User** USER_DEFINED**>**: This statement defines the mechanism to access the memory element internal to the design for the connection. A Control sequence of events allows for values to be made available in the associated state element and at one terminal of the connection. An Observe sequence of events allows for values to be accessible from the terminal of the connection to a measurable point of the design. Both Control and Observe may contain subsequences that involve activities such as Capture, Shift, or Transfer. Definitions of these events are provided in the PatternInformation block of statements. A user-defined keyword for the test access type of sequence is provided through the User user_defined keyword. It is not allowed to use this capability to describe the Control or Observe sequence of events as they are already covered by other keywords allowed in this statement. The user-defined name cannot be used to redefine Control or Observe keywords. The USER_DEFINED name shall follow the definitions for user-defined names as defined by IEEE Std 1450-1999.

<**Macro | Procedure**> NAME: The Macro or Procedure that provides the test access of type Control, Observe, or Control Observe is identified. The name specified should be a valid name defined within the scope of the current statement.

(18) **LaunchClock | CaptureClock| Reset**: This statement identifies the Launch/Capture clock or reset signal relative to the connection established (SIGNAME). The LaunchClock is used to define information about the clocking of the source side of the connection, and CaptureClock is used to define information about the clocking of the sink side of the connection. Thus, an In type connection uses CaptureClock to define the clocking of the entity at the end of the connection internal to the signals in *sigref_expr* on which this statement appears (clocking at the receiving end of the connection). An Out type connection uses the LaunchClock to define the clocking of the entity at the end of the connection internal to the signals in *sigref_expr* on which this statement appears (clocking at the source end of the connection). Using LaunchClock/CaptureClock for the source or sink that is not a memory element defines the ability to connect a memory element that satisfies that property. The clocking condition associates with the value as defined by the DataType-AssumedInitialState of the output/input signal (SIGNAME) in the scope of this statement. The Reset keyword is used to identify the reset signal (SIGNAME) associated with the memory element internal to the connection. When sepecified on an In type of connection, the reset signal affects the memory element at the sink of the connection. When specified on an Out type of connection, the reset signal affects the memory element at the source of the connection. If not specified, no information can be interpreted about the sensitivity of the entity at the end of the connection to clocks or resets of the design. The type of clocking or reset event that affects the value is describe by the following keywords:

(19) **LeadingEdge**: The event of the signal that takes the signal from the AssumedInitialState to the opposite state.

TrailingEdge: The event of the signal that takes the signal from the opposite state of the AssumedInitialState to the AssumedInitialState.

LevelSensitive: The event that refers to the constant level of the signal at the state after the type of edge (LeadingEdge, TrailingEdge) occurs. The associated memory element could change values during this identified state.

(20) **Direct**: The clock signal is a direct connection from a primary input signal.

Indirect: The clock signal is connected via logic from a primary input signal.

(21) **StateAfterEvent**: It defines what happens to the state of the memory element that is internal to the design with respect to the connection. The information provided by this statement reflects the result of changing values on clocks. The next state of the associated memory element could be one of the following:

a) **Connection**: The value that is available through the connection established by IsConnected is captured in the memory element.
b) **ExpectLow**: A logic-0.
c) **ExpectHigh**: A logic-1.
d) **ExpectUnknown**: A logic-X.
e) **ExpectValid**: A logic-0 or a logic-1.
f) **Hold**: Previous state of the memory element.
g) **Invert**: The inversion of the previous state of the memory element.
h) **ShiftState**: The value in the scan cell previous to the current memory element (which is part of a scan chain) in the shift order.
i) **User** USER_DEFINED: A user-defined next state for extendability of CTL. It is not allowed to use this capability to describe next states already covered by other keywords allowed in this statement. The user-defined name cannot be used to redefine any existing StateAfterEvent keywords. The USER_DEFINED name shall follow the definitions for user defined names as defined by IEEE Std 1450-1999.

(22) **Signal** *sigref_expr*: The connection of the *sigref_expr* in the scope of the statement is connected to other signals of the design that are defined in this statement. Let us call the *sigref_expr* in the scope of this statement the *outer_sigref_expr* and the *sigref_expr* defined using this statement the *inner_sigref_expr*. For an In type of connection, the *outer_sigref_expr* represents the source signals of the connection and the *inner_sigref_expr* represents the sink signals of the connection. For an Out type of connection, the *inner_sigref_expr* represents the source signals and the *outer_sigref_expr* represnts the sink signals. The following possibilities exist for the definition of the connection between the *outer_sigref_expr* and the *inner_sigref_expr*:

a) One-to-one correspondence: The number of signals in the *outer_sigref_expr* are the same as the number of signals in the *inner_sigref_expr*. In this case, the first signal of the *outer_sigref_expr* is defined to be connected to the first signal of the *inner_sigref_expr*, the second signal of the *outer_sigref_expr* is defined to be connected to the second signal of the *inner_sigref_expr*, and so on.
b) One-to-many correspondence: In this case, one signal is defined in the *outer_sigref_expr* and more than one signal is defined in the *inner_sigref_expr*. In this case, the signal in the *outer_sigref_expr* is assumed to be connected to all signals in the *inner_sigref_expr*.
a) Many-to-one correspondence: In this case, many signals are defined in the *outer_sigref_expr* and only one signal is defined in the *inner_sigref_expr*. In this case, all signals of the *outer_sigref_expr* are defined to be connected to the single signal identified in the *inner_sigref_expr*.

(23) **StateElement < Scan | NonScan >** CELLNAME_LIST: The connection of the *sigref_expr* in the scope of the statement is connected to a scan chain that is defined in the ScanStructures block that is in the scope of the current CTLMode block when *Scan* is used. The following possibilities exist for the definition of the connection between the signals in the *sigref_expr* and the cells in the *cellname_list*:

a) One-to-one correspondence: The number of signals in the *sigref_expr* are the same as the number of cells in the *cellname_list*. In this case, the first signal is defined to be connected to the first cell in the *cellname_list*, the second signal is defined to be connected to the second cell in the *cellname_list*, and so on.

b) One-to-many correspondence: In this case, one signal is defined in the *sigref_expr* and more than one cell is defined in the *cellname_list*. In this case, the signal is assumed to be connected to all cells in the *cellname_list*.
c) Many-to-one correspondence: In this case, many signals are defined and only one scan cell is defined in the *cellname_list*. In this case, all the signals defined to be connected to the cell identified in the *cellname_list*.

When *NonScan* is used, no cell name is expected. If a cellname is used for a NonScan cell, then the cell name cannot appear as a scan cell in any test mode.

(24) **Transform**: The logic relationship between the source and the sink of the connection is defined by the Transform statement. The transform represents the change to the waveform of the signal as it is propagated from the source to the sink of the connection that is establish. When the associated connection is gated, the transformation is defined for the following cases:

LogicAnd gating: The gating logic resolves to a Logic-1 to allow for the connection to be sensitized.
LogicOr gating: The gating logic is at a Logic-0 to allow for the connection to be sensitized.
LogicXor gating: Although any gating condition allows for the connection to be sensitized, the transformation is defined to the case where the gating logic resolves to a logic-0.

If Transform is missing or an empty transform block is defined, a direct non-inverted relationship is assumed between the two ends of the connection. The transform is to be interpreted for all connections defined in the current block.

WaveformTable: Define the waveform table that contains the WFCs to be transformed. This waveform table shall be in the scope of the current CTLMode block.

Invert: The connection reflects an inversion in the logic values represented by simple waveform characters between the two points that are connected. If this keyword is missing, no information can be interpreted. Although the inversion is clear for the LogicAnd and LogicOr gating condition of a connection, the LogicXor case needs to be defined. In the case of a LogicXor gating of the connection, the inversion is defined for the condition when the gating signal is at a logic-0. That is, no inversion is accounted for by the gating signals logic value in the definition of the Invert statement.

WFCMap FROM_WFC -> TO_WFC: Through this statement, the mapping of waveform characters between the two points of the connection are given. The FROM_WFC reflects the waveform character of the signals in the *sigref_expr* on which the IsConnected statement is defined. The TO_WFC reflects the waveform character of the signals identified for the connection within the IsConnected block. If the signals at the end of the connection do not have their own waveform characters defined, the TO_WFC is to be resolved from the waveform characters that are defined for signals in *sigref_expr*, which is the same mechanism for the FROM_WFC's. A separate WFCMap statemenet should be used for each WFC that needs to be mapped.

DelayCycles INTEGER: The number of clock periods elapsed in the connection for a value at one end of the connection to appear at the other end of the connection. By default a single clock period is to be assumed.

(25) **Wrapper**: The wrapper keyword is used to describe standardized information about scan wrappers around designs that are used to isolate embedded designs from their environment for the purposes of test. On a per connection on the *sigref_expr* basis, the use of IEEE Std 1500-2005, another user-defined wrapper technology, or no wrapper technology can be specified. When the CellID refers to a scan wrapper cell, it is required that the connection be to a scan cell. When the CellID refers to combinational logic, then the associated connection is defined to pass through the logic defined in the wrapper-cellID. This statement should be consistent with the Wrapper statement if it is present in the top level of the CTLMode block. This statement should also be consistent with any Wrapper statement on the wrapper statement on scan cells that are defined in the ScanInternal block of statements. This Wrapper statement differs from the one on *sigref_expr*'s outside the IsConnected block in that this statement is used to identify wrapper cells as opposed to wrapper control signals:

IEEE1500: It is used to signify compliance of the *sigref_expr* to the hardware structure defined by IEEE Std 1500-2005.

None: It is used to signify that the *sigref_expr* is not associated with any of the wrapper technology that comes with the design. This is the default condition for the Wrapper statement.

User USER_DEFINED: It is used to identify the use of a wrapper technology that is defined by another standard or a nonstandardized wrapper.

Wrapper technology defines certain special named implementations of cells or reserves the use of some signals. These are identifiable through the following keywords:

CellID: This keyword allows for the identification of a cell that is taken in context of the type of wrapper identified. Keywords exist to identify the special implementations of IEEE 1500 wrapper cells. The specific definitions of the cell names are defined with the wrapper technology of IEEE Std 1500-2005.

CELL_ENUM: <(WC_S | WH_S) <D|F># | WH_C |WH_CI>

(_C<<I|O|B><I|O|U>|N>)(_U<D|F>)(_O)(<_G0|_G1>)

WC_S : Identifies the beginning of the shift path description of the wrapper cell.

WH_S: This defines harness logic constructed out of sequential elements but is not a wrapper cell as described by special rules in IEEE Std 1500-2005.

WH_C: It describes harnessing logic constructed out of combinational logic. For details on harnessing logic, refer to IEEE Std 1500-2005.

WH_CI: It describes harnessing logic constructed out of combinational logic where the data between the input and the output of the logic are inverted. For details on harnessing logic, refer to IEEE Std 1500-2005.

_C: It identifies the beginning of the capture mechanism description of the wrapper cell.

_U: Identifies the beginning of the update mechanism description of the cell.

_O: Identifies the cell as being an observe only cell.

_G0: Identifies the cell as one that provides a logic-0 value for the safe operation of the design to which the output of the cell is connected.

_G1: Identifies the cell as one that provides a logic-1 value for the safe operation of the design to which the output of the cell is connected.

D: The registers being described are only used for test purposes and are not part of the functional logic of the design. When used with WC_S, it describes a dedicated shift path register. When used with _U, it describes a dedicated update register.

F: The registers are functional registers that are reused for test operations. When used with WC_S, it describes a shared shift path register. When used with _U, it describes a shared update register.

I: When preceded by _C, it indicates that the captured value for the cell is coming from the functional input of the wrapper cell. When preceded by I, O, or B, it indicates that the value is being captured into the first register of the shift path (the one closest to the input of the wrapper cell).

O: When preceded by _C, it indicates that the captured value for the cell is coming from the functional output of the wrapper cell. When preceded by I, O, or B, it indicates that the value is being captured into the last register of the shift path (the one closest to the scan out of the wrapper cell).

B: Indicates that the captured value is coming from either the functional input or the functional output of the wrapper cell. This is the equivalent of the combined I and O functions.

U: Indicates that the value is captured into the update register of the wrapper cell.

N: Indicates that the wrapper cell does not support the capture operation. For example, this would be the case for a control-only wrapper cell.

#: Indicates a positive decimal number that represents the number of state elements in the cell that is part of the shift path.

(26) **IsDisabledBy** <**In|Out**> **Logic** *logic_expr*: This statement is used to describe the disabling condition for the signals in the scope of this statement. The statement allows for disabling to be defined for Input (In) or Output (Out) signals of the design. If IsDisabledBy is not specified on a signal, it should be assumed that the signal cannot be disabled in the current configuration. The disabling signals are specified using positive logic as a logic expression using symbolic names in the verilog syntax.

logic_expr: The logic expression is used to define the boolean relationship between the disabling signals defined within the IsDisabledBy block. For example, a logic expression could be defined as (A & B | ~C), where A, B, and C are symbols whose binding is defined within the IsDisabledBy block. The disabling signals could be signals or scan signals of the design. Positive logic is used to define the disabling condition, which means the disabling condition occurs whenever the logic expression evaluates to logic-1. Information that does not allow for the complete evaluation of the disabling logic expression is an error condition.

LOGICSIGNAME: A locally defined name for use in the logic expression. The logicsigname begins a block that can contain the following statements:

Type: This statement is used define the referenced element and shall be one of the following:

a) **Signal**: A valid signal of the design as defined in the Signals block of statements.

b) **StateElement Scan**: A valid scan cell name as defined in the Scan Structures block. The scan cell could be associated with a core instance. Scan cells are state elements that are part of a scan chain defined in a Scan Structure block of statements.

c) **StateElement NonScan**: A state element of the design that is identifiable with a unique symbol name. The state element cannot be a scan cell name in any mode.

d) **CoreSignal**: A valid signal of a core instance.

Name: The name of the element as appropriate to the Type; i.e., a signal will be named in the Signals or SignalGroups block, a StateElement will be named in the ScanStructures block, and a CoreSignal is defined in the CoreType block.

IsDisabledBy <**In|Out**> <**Macro|Procedure**> NAME: The disabling condition of a connection can be specified as the execution of a macro or procedure with this statement. A macro or procedure is an arbitrary sequence of events defined at externally accessible points of the design. The macro/procedure referred to by name should be part of a MacroDefs/Procedures block that is in the scope of the current CTLMode block. The same is true for a procedure. The **In** and **Out** reflect the direction of the signal of the design for which the disabling mechanism is specified.

(27) **LaunchClock|CaptureClock**: This block definition is essentially the same as the one defined in the IsConnected block. Whereas the one above is placed within an “IsConnected” block, this one allows for a disassociated launch/capture block to be specified. A disassociated Launch/Capture block can be used for

scan chain clocks where the connection is defined in the ScanStructures block or when the details of a connection are not important for the information. This statement has limited syntax to that of the corresponding statement in the IsConnected block.

(28) **OutputProperty**: Several characteristics can be defined on the associated output signal or group of signals through this statement. If not specified, no output property can be assumed.

a) **Edge**: This characteristic is used to describe the way the associated signal or group of signals can be sampled for values. Edge is used for sampling that is based on a value available at a particular point in the period.
b) **Window**: This characteristic is used to describe the way the associated signal or group of signals can be sampled for values. StrobeWindow is used for sampling that is based on a value available during an interval in the period.
c) **PullUp**: This describes the existence of a PullUp device on a tristateable signal.
d) **PullDown**: This describes the existence of a PullDown device on a tristateable signal.
e) **ThreeState**: This describes the ability of the signal to take on three logic values, namely, logic-1, logic-0, and logic-Z.
f) **TwoState0Z**: This describes the ability of the signal to take on two logic values, namely, logic-0 and logic-Z.
g) **TwoState1Z**: This describes the ability of the signal to take on two logic values, namely, logic-1 and logic-Z.
h) **SynchLatch**: This describes the existence of a latch in the design that is used to synchronize the timing of the signal.
i) **SynchFF**: This describes the existence of a flip-flop in the design that is used to synchronize the timing of the signal.
j) **ScanStable**: This describes the activity level (stable) of the associated signal during the scan operations internal to the design being described.
k) **ScanUnstable**: This describes the activity level (unstable) of the associated signal during the scan operations internal to the design being described. A commonly used description is that the outputs are wiggling during the scan operations of the design.
l) **Transition** INTEGER: This statement defines the ability to launch transitions from internal memory elements of the design that would be available through an IsConnected statement. By default it should be assumed that no critical transition can be launched from the associated outputs in *sigref_expr*.
m) **User** USER_DEFINED: A user-defined output property for extendability of CTL. It is not allowed to use this capability for describing signal characteristics already covered by other output properties. The user-defined name cannot be used to redefine any existing output property keywords. The user_defined name shall follow the definitions for user-defined names as defined by IEEE Std 1450-1999.

(29) **StrobeRequirements**: This statement is used to specify the required timing accuracy for a measure of events on signals (outputs) of the design. If the strobe accuracy is not specified, non-critical timing is to be assumed as the default condition. This statement mirrors the DriveRequirements statement with the similar semantics for the syntax:

a) **TimingNonSensitive**: This keyword expresses the fact that the signal can be strobed at any time during the clock period without affecting the validity of the test data associated with the signal.
b) **TimingSensitive**: This keyword expresses the fact that the signal has specific timing requirements for measuring values on the signal. Measure events in the waveform(s) of the associated signals cannot be arbitrarily changed without affecting the validity of the patterns and protocols in the scope of this statement:
    1) **Precision**: Through this statement the precision/resolution of the timings is specified. The *time_expr* represents an value that represents a plus and minus of the timing information. All strobe timing events specified for the associated signal in the scope of this statement is expected to have the precision defined here.

2) **EarliestTimeValid**: This statement describes the earliest time within a test cycle (period) that the signal becomes valid. The time is relative to the start of the test cycle.
3) **LatestTimeValid**: This statement describes the latest time within a test cycle (period) that the signal remains valid. The time is relative to the end of the test cycle.
4) **EarliestChange**: This statement describes the earliest time within a test cycle (period) that the signal is expected to change.
5) **Reference** *sigref_expr*: This statement defines specific signals in a *sigref_expr* for which the associated signals of the statement have a timing relationship which falls under the category of **TimingSensitive**:

SelfEdge <**Leading**|**Trailing**|**LeadingTrailing**> INTEGER: This statement denotes which edges of the associated signals on which this statement is defined have a timing sensitivity to the signals in the **Reference** statement. Leading, Trailing, or LeadingTrailing edges are identifiable to specify the type of reference edge. If no integer is specified, all edges of the specified type are assumed to have the timing requirements. Every waveform defined for the signal has a state at the beginning of the period. The leading edge is defined as the edge that takes the signal from the starting state at the beginning of the period to the other logic level. Trailing edges are the opposite transitions to the leading edges. Through an integer, individual edges can be identified for the timing relationship. The integer refers to the nth edge of the identified type that is relative to the beginning of the test cycle (period). LeadingTrailing is used to identify the condition represented by Leading or Trailing. If the SelfEdge statement is not specified, all events of the signal in its waveforms are to be assumed as the default.

ReferenceEdge <**Leading**|**Trailing**|**LeadingTrailing**> INTEGER: This statement identifies which edges of the Reference signals waveforms are the reference edges for the timing sensitivity. The interpretation of Leading, Trailing, and LeadingTrailing are similar to that defined for the SelfEdge statement. Similarly the integer defines the nth edge of the identified type that is relative to the beginning of the test cycle (period). If a reference edge is not defined, all events of the signal in its waveforms are to be assumed as the default.

EarliestTimeValid: Specifies the time after which the signal is valid (through a time_expr) between the edges of the signal identified by the SelfEdge and the ReferenceEdge statement. The EarliestTimeValid time represents a condition where the SelfEdge occurs after the ReferenceEdge. A negative number represents a value where the SelfEdge occurs before the ReferenceEdge.

LatestTimeValid: Specifies the end of the time window of the signal being valid through a time_expr. The time represets an interval between the edges of the signal identified by the SelfEdge and the ReferenceEdge statement. The LatestTimeValid represents a condition where the SelfEdge occurs after the ReferenceEdge. Thus, a negative number for the time_expr would specify the condition that the SelfEdge's LatestTimeValid occurs before the ReferenceEdge.

EarliestChange: Specifies the earliest time the signal is expected to change relative to the ReferenceEdge. A positive number represents the event where the SelfEdge occurs after the ReferenceEdge.

Precision: This statement is analogous to the Precision of individual edges defined within the DriveRequirements block of statements. The precision defined by this statement denotes the precision/resolution of the timings between edges.

2) **Waveform**: This statement defines the condition that the timing of the signals is to be determined by the WaveformTables in the scope of the current statement (test mode). The timing cannot be changed outside of that specified by the waveforms.

(30) **ScanStyle:** This statement is used to specify the scan style for each scan-in or scan-out signal. The information from this statement can be useful to a core integrator when it may become necessary to catenate scan-chains.

MultiplexedClock: This indicates that the scan chain operates with the clock determining the source of the data being applied during the scan operation.

MultiplexedData: This indicates that the scan chain operates with a non-clock test control signal and determines the source of the data during the scan operation.

LevelSensitive: A race-free shift operation that relies on a single active clock signal at any given time. Any two latches that have a sensitized path between them operate with clocks that can never overlap regardless of the timing of the signals involved.

(31) **Wrapper**: The wrapper keyword is used to describe standardized information about scan wrappers around designs that are used to isolate embedded designs from their environment for the purposes of test. On a per *sigref_expr* basis, the use of IEEE Std 1500-2005, another user-defined wrapper technology, or no wrapper technology can be specified. This statement should be consistent with the Wrapper statement if it is present in the top level of the CTLMode block. This Wrapper statement differs from the one within the IsConnected block in that this statement is used to identify wrapper control signals as opposed to wrapper cells.

IEEE1500: It is used to signify compliance of the *sigref_expr* to the hardware structure defined by IEEE Std 1500-2005.

None: It is used to signify that the *sigref_expr* is not associated with any of the wrapper technology that comes with the design. This is the default condition for the Wrapper statement.

User USER_DEFINED: It is used to identify the use of a wrapper technology that is defined by another standard or a nonstandardized wrapper.

PinID: This keyword allows for the identification of a special signal in context of the type of wrapper identified. These signals are defined along with the wrapper technology in the associated standards. For example, some PinIDs for IEEE Std 1500-2005 are ShiftWR and SelectWIR. This keyword should not be used in conjunction with the None selection of the Wrapper.

10.4 Internal BlockSyntax examples

The Internal block of statements contains all of the information about the design from the signals inwards. As designs can vary dramatically, the information in this subclause is rich in keywords to encompass all test information possible for the designs. User-defined keywords allows for unanticipated needs. All keywords cannot and should not be used for a single design. In the following example, a sample usage of some statements is shown. Based on this, the reader should easily interpret the usage of keywords that are allowed by the syntax but not shown in the example.

A single design with signals scan_in[0..7], scan_out[0..7], clk, clk2, scan_enable, functional_ins[0..15] and functional_outs[0..20] is being described.

```
Environment {
   CTLMode internal_mode1 {
     TestMode InternalTest;
       Internal {
       // scan_in[0..7] are scan in signals for eight internal scan chains. These scan-ins are also
       // functional inputs of the design. Furthermore, the test data require an extra stimulus outside of
       // the scan data as specified by the DataRateForProtocol.
       scan_in[0..7] {
         DataType Functional TestData ScanDataIn {
           ScanDataType Internal;
           DataRateForProtocol Maximum 1;
         }
       }

       // scan_out[0..7] are scan out signals for eight internal scan chains. These are test-only signals
       scan_out[0..7] {
         DataType TestData ScanDataOut {
           ScanDataType Internal;
         }
       }

       // clk is a functional and a test clock that is assumed to be at a logic-0 at the beginning of every
       // protocol in the test mode being described. Because of the functional attribute, the clock can be
       // used to capture values in test sequences. The ScanMasterClock attribute also describes clk to
       // be used during scan operations.
       clk {
         DataType Functional TestControl ScanMasterClock {
           AssumedInitialState ForceDown;
         }
       }

       // clk2 is a dedicated test clock. An example of this could be the clock of a control block.
       // clk2 is expected to be glitch-free and stable during scan operations external to the design.
       clk2 {
         DataType TestControl {
           AssumedInitialState ForceDown;
         }
         InputProperty ScanStable;
       }

       // The scan enable signal is defined to enable the scan chain configurations with a logic-1.
       scan_enable {
         DataType TestControl ScanEnable {
           ActiveState ForceUp;
         }
       }

       // Functional inputs and outputs of the design that receive test data (broad side stimulus).
       'functional_ins[0..15]+functional_outs[0..20]' {
         DataType Functional TestData;
       }
     } // end Internal block
   } // end CTLMode internal_mode1
} // end environment_block
```

```
// The following examples show the restrictions on sigref_expr's as
// defined by this standard.
// sigref_expr: Multiple instantiation of the same sigref_expr in a
// single internal block is not allowed. A signal can be part of
// multiple sigref_expr when it participates in named signal groups

// The following snippet of an example is not allowed in the CTL
Environment { CTLMode { Internal {
   clk { /* information on clk */ }
   'clk+clk2' { /* information on clk and clk2 */ }
}}}
// clk has multiple block definitions.


// The following snippet of an example is allowed in the CTL
SignalGroups { clocks[0..1]='clk+clk2';}
Environment { CTLMode { Internal {
   clk { /* information on clk */ }
   clocks[0..1] { /* information on clk and clk2 relevant to the use
                   of the signals through the groupname.*/ }
}}}
// clk and clocks have only one block associated with each.
```

```
// sigref_expr's that break any signal group into its subentities
// by using the "-" or bit-indexing capability ([]) make the sigref_expr
// resolve to the individual signals that make up the resulting
// sigref_expr.

// The following snippet of an example is allowed in the CTL
Environment { CTLMode { Internal {
   clocks[0..1] { /* information on clk and clk2 */ }
   'clocks-clocks[1]' { /* information on clk */ }
}}}
// 'clocks-clocks[1]' resolves to clk as the group is bit-indexed. Thus,
// clocks and clk have only one block of information each.
```

Consider the example of two input signals, signal1 and signal2, which have the following timing relationship:

a) signal1 has can rise a minimum of 5 ns before a rising transition on signal2.
b) signal1 can fall after a minimum of 3 ns after a rising transition of signal2.

The following describes this timining relationship, which is shown in Figure 6:

```
Signals { signal1 In; signal2 In;}
Environment { CTLMode mymode { Internal {
   signal1 {
     DriveRequirements {
       TimingSensitive {
         Reference signal2 {
           ReferenceEdge Leading;
           SelfEdge Leading;
           Setup '5ns';
         }
         Reference signal2 {
           ReferenceEdge Leading;
           SelfEdge Trailing;
           Hold '3ns';
         }
       }
     }
   }
}}}
```

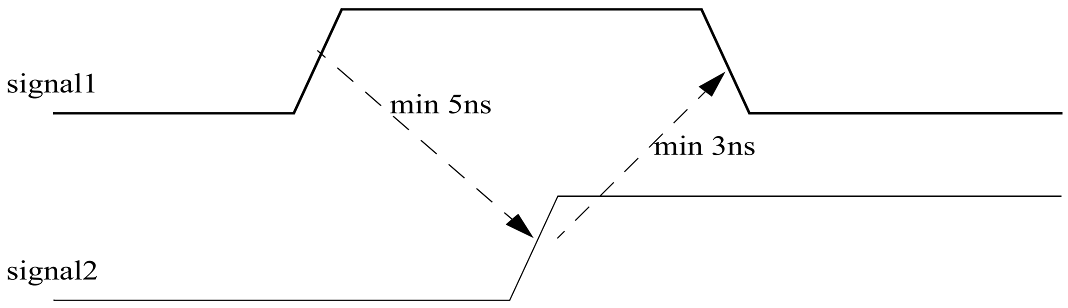


Figure 6—Example of a timing relationship

11. CTLMode—ScanInternal block

11.1 General

This block of statements is used to describe information on scan cells that are defined in ScanStructures or CoreInstance-ScanStructures. The syntax refers to the *data_type_enum* and the *ScanDataType_enum* defined in 10.2.

11.2 ScanInternal syntax

Environment { **CTLMode** (CTLMODE_NAME) {

ScanInternal { (1)

(*cellref_expr* { (2)

(**DataType** (*data_type_enum*)+ ;) (3)
(**DataType** (*data_type_enum*)+ {

```
        ( ActiveState                                                                  (4)
            < ForceDown
            | ForceUp
            | ForceOff
            | ForceValid
            | ExpectLow
            | ExpectHigh
            | ExpectOff
            | ExpectValid
            > ; )
        ( ScanDataType ( ScanDataType_enum )+ ; )                                      (5)
        ( ValueRange INTEGER INTEGER (CORE_INSTANCE_NAME)+; )*                         (6)
        ( UnusedRange INTEGER INTEGER;)*                                               (7)
    } )* // end DataType

    ( IsConnected < In|Out > {                                                         (8)
        ( CoreSignal sigref_expr ; )                                                   (9)
        ( StateElement <Scan | NonScan> (cellref_expr) ; )                            (10)

        ( IsGatedBy <LogicAnd | LogicOr | LogicXor> logic_expr {                      (11)
            ( LOGICSIGNAME {
                Type < Signal | StateElement Scan | StateElement NonScan | CoreSignal >;
                Name <SIGNAME | CELLNAME | SYMBOLNAME>;
            } )+ // end logicsigname
        } )* // end IsGatedBy
        ( IsGatedBy < Macro | Procedure > NAME ; )*

        ( < LaunchClock | CaptureClock | Reset > SIGNAME {                            (12)
            ( <LeadingEdge | TrailingEdge> (LevelSensitive); )
            ( <Direct | Indirect>; )
            ( StateAfterEvent <
                Connection
                | ExpectLow
                | ExpectHigh
                | ExpectUnknown
                | ExpectValid
                | Hold
                | Invert
                | ShiftState
                | User USER_DEFINED > ; )
        })* // end LaunchClock

        (ScanDataType (ScanDataType_enum)+);                                          (13)

        ( TestAccess <                                                                (14)
            (Control
            | Observe
            | User USER_DEFINED )+ >
            < Macro | Procedure > NAME; )*

        ( Transform {                                                                 (15)
            ( WaveformTable (WFT_NAME)+; )
            ( Invert ; )
            ( WFCMap FROM_WFC -> TO_WFC; )*
```

```
                  ( DelayCycles INTEGER;)
               } )* // end Transform

            } )* // end IsConnected
            ( Wrapper <IEEE1500 | None | User USER_DEFINED > ( CellID cell_enum) ; )          (16)
         } )+ // end cellref_expr
      } // end ScanInternal
}} // end Environment, CTLMode
```

11.3 ScanInternal block syntax descriptions

(1) **ScanInternal**: This statement begins the named ScanInternal block within a CTLMode block. The ScanInternal block contains statements that describe the internals of the design between the scan chains of the design and the logic away from the signals of the design. There shall be only one ScanInternal block that defines all connections to all scan chains.

(2) *cellref_expr*: This is a cell name or group of cell names. In the ScanInternal block, statements are assigned to the cells part of the *cellref_expr*. All rules of the information in *cellref_expr*'s are parallel to that of *sigref_expr*'s. Multiple instantiation of the same *cellref_expr* in the ScanInternal block is not allowed. A cell name or group name can appear only once in the *cellref_expr*'s of a single block. A cell can be part of multiple *cellref_expr* when the cell is part of named groups, and the information linked to the *cellref_expr* should only be interpreted when the named entity of the *cellref_expr* is used. Any bit-indexing or use of the "-" operator to break a named cell group makes the entire *cellref_expr* resolve to the individual cells. All cell names shall be unique across all scan chains and fall in the same name space as scan chain names. Thus, a *cellref_expr* can contain cell names, cell group names, and scan chain names. The cell names, cell group names, and scan chain names can be defined in the ScanStructures block (outside a core) or in the ScanStructures block inside a core. See the definition of the CoreType statement.

The definitions of the syntax in this clause should be taken from the Internal block. The semantics of the statements is identical with a context of cell-names.

The Wrapper statement semantics differs from that in the Internal block in the fact that it is providing information on about the cells of the *cellref_expr*, and it is not associated with a connection. Thus, a combinational wrapper cell is not relevant in the context of the statement on *cellref_expr*'s. If the Wrapper statement is used in the Internal block and in the ScanInternal block to refer to the same cell, the information must be consistant.

11.4 ScanInternal block syntax example

The following syntax shows the definition of six scan cells, namely, c[0], c[1], c[2], c[3], c[4], and c[5]. Then information is provided for the scan cells in the environment:

```
ScanStructures {
   ScanCells c[0..5];
}
Environment {
   CTLMode {
     ScanInternal {
       c[0] { /* information on c[0] */}
       'c[1..5]' { /* information on c[1..5] */}
     }
   }
}

Examples of information on scan cells.
DataType TestFail; // the cell contains a value that indicates the
                // failure of a test.

DataType TestDone { ActiveState ForceUp;} // the cell contains a value
               // that indicates the end of execution of a test. The
               // condition is reached if the cell has a value of
               // logic-1.

DataType TestDone { ActiveState U;} // Same as ActiveState ForceUp.

IsConnected In { CoreSignal core1:sig1; Transform { Invert;}} // the cell
               // has a path sensitized to an internal core of the design
               // (to its sig1), and the value gets inverted in the path.

// The cell has a sensitized path to another scan cell named c[0]. This
// path is sensitized when sigA of the design is at a logic-1, and the
// signal on the internal core of the design named sig1 is also at a
// logic-1. The value in state element c[0] can be observed by operating
// the macro named mymacro_to_observe_scan_cell.
IsConnected In { StateElement Scan c[0];
   IsGatedBy LogicAnd ‘a&b’ {
     a { Type Signal; Name sigA;}
     b { Type CoreSignal; Name core1:sig1; }}
   TestAccess Observe Macro mymacro_to_observe_scan_cell; }

// the scan cell has a non-scan element on its input side. The path is
// always sensitized.
IsConnected Out { StateElement NonScan; }

// the scan cell has a sensitized path from scan cell named c[1] on its
// input side.
IsConnected Out { StateElement Scan c[1]; }

// the scan cell has a sensitized path on its input side from a
// scan cell that is part of a core internal to the design.
// The core instance name is core1, and the scan cell name is c[5] of the
// core1.
IsConnected Out { StateElement Scan core1:c[5]; }

// the scan cell has a sensitized path from a internal core signal of
// the design. The path is sensitzed when sigB of the design is at
// a logic-1. The scan cell changes state on the leading edge of clk1.
// After the leading edge of clk1, the scan cell goes to an unknown
// state.
IsConnected Out { CoreSignal core1:sig2;
   IsGatedBy LogicAnd a { a { Type Signal; Name sigB;}}
   LaunchClock clk1 { LeadingEdge; StateAfterEvent ExpectUnknown; }}
```

12. CTLMode—CoreInternal block

12.1 General

This block of statements is used to describe information on the signals of CoreInstances defined in the design. The signals of the core instances are referenced in the *sigref_expr*, and information is provided for it. The syntax refers to the *data_type_enum* that is defined in 10.2.

12.2 CoreInternal syntax

```
Environment { CTLMode (CTLMODE_NAME) {

    CoreInternal {                                                          (1)
       ( sigref_expr { // format = coreinstance:signame                     (2)
                                                                            (3)
          ( DataType ( data_type_enum )+ ; )                                (4)
          ( DataType ( data_type_enum )+ {
             ( ActiveState
                < ForceDown
                | ForceUp
                | ForceOff
                | ForceValid
                | ExpectLow
                | ExpectHigh
                | ExpectOff
                | ExpectValid
                >;)
             ( ScanDataType ( ScanDataType_enum )+ ; )                      (5)
             ( ValueRange INTEGER INTEGER (CORE_INSTANCE_NAME)+; )*         (6)
             ( UnusedRange INTEGER INTEGER;)*                               (7)

          } )* // end DataType

          ( IsConnected < In|Out > {                                        (8)

             ( CoreSignal sigref_expr ; )                                   (9)

             ( IsGatedBy <LogicAnd | LogicOr | LogicXor> logic_expr {       (10)
                ( LOGICSIGNAME {
                   Type < Signal | StateElement Scan | StateElement NonScan | CoreSignal >;
                   Name <SIGNAME | CELLNAME | SYMBOLNAME>;
                } )+ // end logicsigname
             } )* // end IsGatedBy
             ( IsGatedBy < Macro | Procedure > NAME ; )*

             ( TestAccess                                                   (11)
                (Control
                | Observe
                | User USER_DEFINED )+
                < Macro | Procedure > NAME; )*

             ( Transform {                                                  (12)
                ( WaveformTable (WFT_NAME)+; )
                ( Invert ; )
                ( WFCMap FROM_WFC -> TO_WFC; )*
                ( DelayCycles INTEGER;)
              } )* // end Transform

          } )* // end IsConnected

      } )+ // end cellref_expr
    } // end CoreInternal
}} // end Environment, CTLMode
```

12.3 CoreInternal block syntax descriptions

(1) **CoreInternal**: This block of statements allows for information to be defined on CoreInstance-signals and CoreInstance-SignalGroups. Apart from this difference, this block has the same function as the Internal block or the ScanInternal block of statements. Information here is between the boundaries of two cores in the design or about the boundary of the core. There shall be only one CoreInternal block in a single CTLMode block of statements.

(2) *sigref_expr*: The signals and signal groups in the *sigref_expr* have a format of *coreinstance:signame, coreinstance:signalgroupname,* and *coreinstance:domainname::signalgroupname.*

The definitions of the syntax in this subclause should be taken from the Internal block. The semantics of the statements is identical with a context of coreinstance:signames.

12.4 CoreInternal block syntax examples

```
CoreType A {
   Signals { ins[0..5] In; outs[0..3] Out; }
}
CoreInstance A { I1; I2;}
Environment {
   CTLMode {
     CoreInternal {
       I1:ins[0] { /* information on i[0] of I1 */ }
       'I1:outs[0..3]+I2:outs[0..3]' {
           // information on the outputs of I1 and I2
       }
     }
   }
}
```

Examples of information on core signals. These are similar to scan cells. The examples are intentionally the same to show the similarity of the constructs.

```
DataType TestFail;
DataType TestDone { ActiveState ForceUp;}
DataType TestDone { ActiveState U;}
IsConnected In { CoreSignal I2:ins[1]; Transform { Invert;}}
IsConnected In { CoreSignal I1:ins[0..3];
```

```
   IsGatedBy LogicAnd a&b {
     a { Type Signal; Name sigA;}
     b { Type CoreSignal; Name I2:outs[3]; }}
   TestAccess Observe Macro mymacro_to_observe_value_from core_input; }
IsConnected Out { CoreSignal I1:ins[1];
   IsGatedBy LogicAnd a { a { Type Signal; Name sigB;}}}
```

13. CTLMode—Relation Block

13.1 General

The Relation block is used to define relationships between signals and/or signal groups. The default is that there is no relationship between signals. The allowed syntax is as follows.

13.2 Relation syntax

Environment { **CTLMode** (CTLMODE_NAME) {
  Relation { (1)
    (**Bus** *sigref_expr* (BUSNAME);)* (2)
    (**Common** *sigref_expr* ;)* (3)
    (**Corresponding** *sigref_expr* ;)* (4)
    (**Differential** *sigref_expr* (*sigref_expr*) ;)* (5)
    (**Equivalent** *sigref_expr* (*sigref_expr*);)* (6)
    (**Independent** *sigref_expr* (*sigref_expr (***Set***))*;)* (7)
    (**InOutSet** *signame signame* (*signame*);)* *// enable, output name, input name* (8)
    (**MuxSet** *sigref_expr* (*sigref_expr* { *tag* })+ ;)* (9)
    (**NotEquivalent** *sigref_expr sigref_expr*;)* (10)
    (**OneCold** *sigref_expr*;) * (11)
    (**OneHot** *sigref_expr*;) * (12)
    (**Port** *sigref_expr (integer)* ;)* *// port data, port select value* (13)
    (**ZeroOneCold** *sigref_expr*;) * (14)
    (**ZeroOneHot** *sigref_expr*;) * (15)
  } *// end Relation*
}} *// end Environment, CTLMode*

13.3 Relation block syntax descriptions

(1) **Relation**: This statement begins the relation block, which allows for the definition of relationships between various signals and/or signal groups (*sigref_expr*) that are in the scope of CTL. These signals/signal groups are exclusive of those defined in the CoreType block. The information defined in the relationship is applicable to the current CTLMode block and is available to other CTLMode blocks through the inheritance mechanisms defined in CTL.

(2) Bus *sigref_expr (*BUSNAME*)*: The signals contained in the *sigref_expr* are part of a group of signals that are treated as a single entity or a bus. The bus, which contains the identified signals, can be named (BUSNAME) using the naming convention for names as defined by IEEE Std 1450-1999. Examples of this entity are the Address Bus and Data Bus of Memories. This keyword differentiates between arbitrary groupings of signals created with the bused syntax in CTL (name[]) and actual buses that exist. Buses typically have an ordering to the signals in the group. The ordering information should be obtained from the one defined by the grouping mechanism for buses. For example, signals a[31..0] represent a set of signals that have an MSB, a[31], and an LSB of a[0]. This statement does not require the signals in the *sigref_exp*r to have the [] notation.

(3) **Common** *sigref_expr*: The signals contained in the *sigref_expr* of this statement are to be connected together external to the core. For example, two signals of the current design, namely A and B, are identified as common if the logic external to the design in the embedded environment is to connect them to the same source. This statement provides the opposite meaning to the Independent statement in the Relation block.

(4) **Corresponding** *sigref_expr*: The signals identified in the *sigref_expr* of this statement are expected to be connected externally to points that have similar characteristics. For example, two signals of the current design, namely A and B, that is to be embedded in an SoC could be either connected to an input of the SoC or could be connected to a scan-element in the SoC. Although both are possible, the corresponding statement would limit the choice of the connection to either both A and B are connected to inputs of the SoC or both A and B are connected to scan-elements on the SoC. The case where one signal comes from one of the two choices and the other signal comes from the other choice is not allowed with the Corresponding statement in effect.

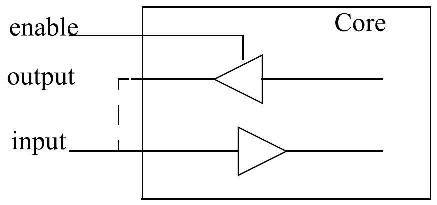
(5) **Differential** *sigref_expr* (*sigref_expr)*: This statement is used to identify differential signals on the design. If only one *sigref_expr* is used, all signals in the *sigref_expr* are differential signals with no relationship among the signals. If a second *sigref_expr* is defined, both sets of signals are identified as differential with a one-to-one relationship (in the same order) between the signals in the two *sigref_expr*'s to identify the signal that has opposite waveforms.

(6) **Equivalent** *sigref_expr* (*sigref_expr)*: This statement links signals that function exactly the same and have the same values at all times during the configuration being described. These signals may or may not be connected to the same source in the SoC. When one *sigref_expr* is used, all signals within it have the same values. When two sets of *sigref_expr*'s are used, the two sets of *sigref_expr*'s shall have the same number of signals such that there is a one-to-one correspondence established between the signals in the order they are specified. Each corresponding signal pair has the same values at all times during the configuration being described.

(7) **Independent** *sigref_expr* (*sigref_expr (***Set***))*: The signals identified in this statement represent signals that are expected to be managed externally to the core in a way that they are completely isolated from one another. The signals identified cannot be connected to the same third point/net/signal externally. Therefore they should not be connected on the SoC. There are three forms that this statement can express:

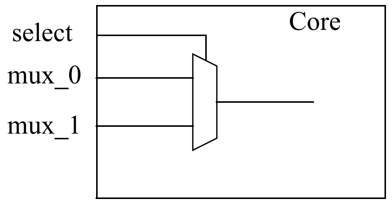
a) **Independent** *sigref_expr:* All signals in the expressions are expected to be isolated from all others in the group.
b) **Independent** *sigref_expr sigref_expr:* All signals in the first expression are expected to be isolated from the signals of the second expression with one-to-one ordering relationship.
c) **Independent** *sigref_expr sigref_expr* **Set**: All signals in the first expression are expected to be isolated from all signals in the second expression.

(8) **InOutSet** *signame signame* (*signame*): The signals specified in this statement form an input/output set that has not been combined by a wrapper cell. The first signame is the I/O enable signal. The second signal name is the output signal and the output driver is turned on when the I/O enable signal is a "1." The third signal is optional and is the name of the input signal that is enabled when the output driver is turned off by the enable signal being a "0."



(9) **NotEquivalent** *sigref_expr sigref_expr*: The number of signals in the two *sigref_expr*'s should be the same. There shall be at least two signals in each *sigref_expr*. There is a one-to-one correspondence established between the two sets of signals based on the order in which they are specified. The logical values taken on by the two signals of each corresponding pair cannot be the same for all pairs. Thus, if the two sets are address busses of a multiport memory, then this statement could be used to specify that the two address busses can never take on the same address.

(10) **MuxSet** *sigref_expr* (*sigref_expr* { *tag* })+: The signals specified in this statement are all part of a multiplexor set. The signal or set of signals (*sigref_expr*) that is specified first is the select portion of the multiplexor that determines which of the inputs to follow is connected to the output of the multiplexor. Following the control signals are sets of signals identified in *sigref_expr*'s that represent inputs to the multiplexor. These inputs can be individual signals or sets of signals that represent busses. Following each input *sigref_expr* is a tag that represents the binary value on the select signals that picks the associated input signals of the multiplexor. Thus, the tag represents a string of 0's and 1's such that the values have a one-to-one correspondence (in the same order) with the select signals of the first *sigref_expr* of this statement. Tag lengths that do not match the number of select signals is an error condition.



MuxSet select mux_0 { 0 } mux_1 { 1 };

(11) **OneCold** *sigref_expr:* The signals specified in this statement are expected to have values of all "1's" except one of them being "0."

(12) **OneHot** *sigref_expr:* The signals specified in this statement are expected to have values of all "0's" except one of them except one of them being "1."

(13) **Port** *sigref_expr (integer)*: The signals identified in the *sigref_expr* are related to each other as signals that correspond to a single port of a multiported entity within the design being represented. The port can be given an integer identifier that can be used to identify this set of signals. This integer is expected to be unique for across all ports defined within the same scope of the test mode information.

(14) **ZeroOneCold** *sigref_expr:* The signals specified in this statement are expected to have values of all 1's or *OneCold* characteristic.

(15) **ZeroOneHot** *sigref_expr:* The signals specified in this statement are expected to have values of all 0's or *OneHot* characteristic.

13.4 Relation block syntax example

In the following example, some signals are defined for a design and relationships are assumed for the purposes of showing examples of the syntax:

```
Signals {
   A In; B In; C In; D In; E In; F Out;
}
Environment {
   CTLMode {
     Relation {
       // A and B are to be connected up to the same signals in the SoC
       Common 'A+B';
```

```
// A and C should be connected to similar constructs on the SoC
Corresponding 'A+C';

// A and B are differential signals but no relationship defined to
// each other.
Differential 'A+B';

// A and B are always opposite values to each other.
Differential A B;

// A and B are differential and take on opposite values and C-D
// have a similar relationship as A and B have to each other.
Differential 'A+C' 'B+D';

// A and B take on the same values all the time, and B and D take
// on the same values all the time.
Equivalent 'A+C' 'B+D';

// A and B are to be connected to different signals in the SoC
Independent 'A+B';

// A and B are to be connected to different signals in the SoC and
// C and D are to be connected to different signals on the SoC
Independent 'A+C' 'B+D';

// A and C are to remain independent from B and D in the SoC
Independent 'A+C' 'B+D' Set;

// The three signals are part of a bidirectional signal where
// F and A are connected
InOutSet E F A;

// The three signals are part of a multiplexer internal to the
// design. E = enable.
MuxSet E A F;

// The two sets of signals (addr busses) cannot have the same
// logical address.
NotEquivalent 'p1addr[0..2]' 'p2addr[0..2]';

// The signals ABC can take on the following values: 110, 011,
// 101
OneCold 'A+B+C';

// The signals ABC can take on the following values: 100, 010,
// 001
OneHot 'A+B+C';

// The signals ABC can take on the following values: 110, 011,
// 101, 111
ZeroOneCold 'A+B+C';

// The signals ABC can take on the following values: 100, 010,
        // 001, 000
        ZeroOneHot 'A+B+C';

        // The signals A and B are part of a port of an internal memory
        // element.
        Port 'A+B';

        // The signals A and B are part of a port of an internal memory
        // element, and the port is labeled 1.
        Port 'A+B' 1;
    }
}
```

14. CTLMode—ScanRelation block

14.1 General

The ScanRelation block is used to define relationships between scan cells (cellnames or cell groups) that are specifiable in a *cellref_expr*. The default is that there is no relationship between cells. The allowed syntax is given in this clause.

14.2 ScanRelation syntax

Environment { **CTLMode** (CTLMODE_NAME) {
    ScanRelation { (1)
        (**Differential** *cellref_expr* (*cellref_expr*) ;)* (2)
        (**Equivalent** *cellref_expr* (*cellref_expr*);)* (3)
        (**OneCold** *cellref_expr*;) * (4)
        (**OneHot** *cellref_expr*;) * (5)
        (**ZeroOneCold** *cellref_expr*;) * (6)
        (**ZeroOneHot** *cellref_expr*;) * (7)
    } *// end ScanRelation*
}} *// end Environment, CTLMode*

14.3 ScanRelation block syntax descriptions

The ScanRelation block mimics the capabilities of the Relation block. The only difference is in the fact that the relationship is defined between scan cells. As this is the case, the definitions and example usage of the syntax should be taken from the Relation block definitions.

15. CTLMode—External block

15.1 General

CTL is used to describe designs that are to be embedded in a larger design called the SoC. The External Block of statements defines information that lies outside the design hierarchy to which the CTL is written such that the correct connections can be made on the SoC. On a signal-by-signal basis, the information about embedding the design is described. When the design is embedded, the information in the External block of the embedded designs CTL can be used for verification purposes. The syntax refers to the *data_type_enum* and the *cell_enum* that are defined in 10.2.

15.2 External statement syntax

```
Environment { CTLMode (CTLMODE_NAME) {

    External {                                                              (1)
      ( sigref_expr {                                                       (2)

        ( AllowSharedConnection {                                           (3)
            ( Core NAME (sigref_expr) ; )*
            ( DataType (data_type_enum)+ ; )
            ( Family (NAME)+ ; )
            ( OutputFunction <And | Or | Xor> ; )
            ( Self sigref_expr ; )
            ( Vendor (NAME)+ ; )
        } )* // end AllowSharedConnection

        ( ConnectTo {                                                       (4)
            ( Core NAME sigref_expr; )
            ( DataType (data_type_enum)+ ; )
            ( Fanout INTEGER; )
            ( Instruction; )
            ( NoRebuffering;)
            ( Symbolic (SYMBOLIC_NAME)+; )
            ( TAM; )
            ( Termination < TerminateHigh | TerminateLow | TerminateOff | TerminateUnknown >
                    (TERMINATION_VALUE);)
            ( Safe; )
            ( WBR; )
            ( Wrapper
                <IEEE1500 | None | User USER_DEFINED >
                (< CellID cell_enum | PinID USER_DEFINED_PIN_ID >) ; )
        } )* // end ConnectTo

      } )+ // end sigref_expr
    } // end External
}} // end Environment, CTLMode
```

15.3 External block syntax descriptions

(1) **External**: This statement begins the external block within the CTLMode block. The external block contains statements used to provide information to the core integrator about the expected environment outside the core boundary. As the external environment of the core is yet to be created, the contents of this block are to be considered as suggestions to the integrator.

(2) *sigref_expr*: This is a signal or group of signals or an expression combining the first two entities. In the external block, statements assign properties to the signals or signal groups that are part of the *sigref_expr*. Multiple instantiations of the same *sigref_expr* in a single external block are not allowed. Resolution of

information is down to individual signals that get the union of information from the blocks of information on the signal and the blocks on named signal groups of which the signal is part. For example, in the same block of statements, one can see information on signal *mysignal* and information on signal group *allsignals[0..10]*, which includes *mysignal*. *mysignal* gets the combined information from the two blocks of information. *sigref_expr*'s, which break any signal group into its subentities by using the "-" or bit-indexing capability ([]), make the *sigref_expr* resolve to the individual signals that make up the resulting *sigref_expr*. Refer to the definition and examples of the InheritCTLMode statement for more details on interpreting *sigref_expr*'s and association of information to the entities in the *sigref_expr* (9.2). The exact definition of *sigref_expr* should be taken from IEEE Std 1450-1999 and its extension in this standard. The CTL provider shall ensure that signal information statements are consistent with each other when information is provided on a signal and when the signal is part of a named signal group in the same block.

(3) **AllowSharedConnection**: Through this statement the allowed parallelism in connecting and hence testing different cores can be specified. If specific signals are specified on entities external to the design, there should be a one-to-one mapping in the same order between the signals that this statement is attributing and the signals of the other external entity. No parallelism can be assumed if not explicitly specified by this statement. In the AllowSharedConnection block, multiple statements can be specified and a logic AND of the associated information should be assumed between the information within the block.

Core *sigref_expr*: The shared connection is defined with signals on another Core with core_name. The other Core should be defined using the FileReference statement in the environment block and identifies with a file type equal to Core. The file name and the name should be specified according to IEEE Std 1450 and IEEE Std 1450.1-2005. The format of the *sigref_expr* is of the format corename:signame.

DataType *data_type_enum*: The signals with DataType referred to by the data_type_enum are valid for parallel testing. All signals identified in the Internal block of the CTLMode for the associated core are referred to by this statement.

Family NAME: This refers to cores that are linked to a certain family. The cores referred to by this statement are cores that have the Family statement with the same name as specified here in their top-level CTLMode block.

OutputFunction And|Or|Xor: This statement refers to the logic function allowed to combine the outputs of different cores. Outputs can only be combined using the logical AND, OR, or XOR functions.

Self *sigref_expr*: The shared connection is defined with signals on the current design.

Vendor NAME: This refers to cores that are linked to some Vendor (foundry, tool, etc.). The cores referred to by this statement are cores that have the Vendor statement with the same name as specified here in their top-level CTLMode block.

(4) **ConnectTo**: This statement allows the core provider to provide suggestions to the integrator on connections outside the core. Multiple statements can exist within a ConnectTo block. A logic AND is to be assumed between the information provided by each statement in the same block of information. A logic-OR condition is to be assumed between multiple ConnectTo statements to define an either/or recommendation.

Core *sigref_expr*: A connection to a specific signal on a design specified by the NAME and the *sigref_expr* is defined by this statement. The NAME should be identified through the FileReference statement in the environment (syntax defined by IEEE Std 1450.1-2005) as another design. The NAME is the name of the design that could include the entire path to access it. *sigref_expr* identifies an existing signal or set of signals of the design. If a group of signals are identified, a one-to-one mapping (in the same order) is assumed between the *sigref_expr* on which the ConnectTo statement is being assigned and the *sigref_expr* of the signals identified on NAME.

DataType *data_type_enum*: Signals that have characteristics defined by the *data_type_enum*. The definition of the data types should be obtained from the internal block statements definition of the DataTypes.

Fanout INTEGER: This statement defines the number of external gates that the associated signal is allowed to drive. The number is only allowed to be an integer.

Instruction: The outputs of the instruction register of wrapper technology that configures the structures inside a design. This reflects the ability to control the associated signal of the design through the state of an instruction register external to the design.

NoRebuffering: No logic gates can be added to the connection of the signal. This is common for signals where the I/O Pads are already implemented in the design being described.

Symbolic SYMBOLNAME: A symbolic name that is to be considered as a means of communicating a global point in the environment of the design that is to be considered unique across all designs in the environment. The SYMBOLNAME should follow the user-defined naming conventions for signals defined by IEEE Std 1450-1999.

TAM: The test access mechanism put in place on the SoC outside of the embedded design to transport data to individual designs. Example TAMs are buses and scan-chains.

Termination: It indicates the need for some form of termination to be applied to the associated signal. The termination could be provided by a design construct external to the design or through an ATE.

> **TerminateHigh**: Indicates that the signal is to be terminated to a logic-1 to resolve the high-impedance states on the signal.
>
> **TerminateLow**: Indicates that the signal is to be terminated to a logic-0 to resolve the high-impedance states on the signal.
>
> **TerminateOff**: Indicates that the signal should not be terminated to resolve floating states.
>
> **TerminateUnknown**: Indicates that some termination is required to resolve the floating states on the signal. However, the type of termination is unknown and any form of termination could be applied to the signal to resolve high-impedance states.
>
> TERMINATION_VALUE: A termination value that is specified as a *time_expr* (refer to 6.13 of IEEE Std 1450-1999). For example 50 Ohms would be specified as “50Ohm”.

Safe: The functional (user-defined logic) outside the design that is configured in such a way that no conflicting values can occur at the associated signals of the design when they are changing values during the operation of the design in the current mode.

WBR: A wrapper boundary register that is a scannable memory element.

Wrapper: Using this statement, a specific wrapper and associated connections can be recommended to be instantiated outside the design.
IEEE1500: An IEEE Std 1500-2005 compliant wrapper.
None: No wrapper construct.
User USER_DEFINED: A user-specified wrapper can be named. The name follows the legal naming conventions set by IEEE Std 1450-1999.

CellID *cell_enum*: This keyword allows for the identification of a cell that is taken in context of the type of wrapper identified. The cell_enum allows for the description of the behavior of the cell. The definition of the cell_enum should be obtained from the Internal Block statements definition of the same. This keyword should not be used in conjunction with the None selection for Wrapper.

PinID USER_DEFINED_PIN_ID: This keyword allows for the identification of a special signal in context of the type of wrapper identified. The keywords to identify the special signals are defined in the associated standard's document. The USER_DEFINED_PIN_ID should be constructed using the rules for names as defined by 1450.0. This keyword should not be used in conjunction with the None selection of the Wrapper.

15.4 External block syntax example

Consider a fictitious design with input signals a, b, c, d, and e and an output signal f. Without worrying about any specific implementation or feasibility of the information, some examples are provided below to show how the syntax could be used:

```
Signals {
   a In; b In; c In; d In; e In; f Out;
}
Environment {
   CTLMode {
     External {
       a {
         // signal a and b of the current design being described
         // can connect to the same design entity externally.
         AllowSharedConnection { Self b;}

         // Signal a is recommended to be connected to an input terminal
         // in the embedded environment.
         ConnectTo { DataType In;}
       }
     }
   }
}
```

This code shows the exact location of the External block of statements and the AllowSharedConnection and ConnectTo statements in CTL.

16. CTLMode—PatternInformation block

16.1 PatternInformation syntax

The pattern information block is used to attach extra information to various types of pattern-related elements such as Pattern blocks, PatternBurst blocks, Procedures, and Macros. The information contained here can describe usage, purpose, or additional details such as fault coverage. The syntax refers to *pattern_or_burst_enum* and *exec_enum* that are defined in 9.2.

procedure_or_macro_enum =
< **Capture**
| **Control**
| **DoTest**

```
    | DoTestOverlap
    | Hold
    | Instruction
    | MemoryPrecharge
    | MemoryRead
    | MemoryReadModifyWrite
    | MemoryRefresh
    | MemoryWrite
    | ModeControl
    | Observe
    | Operate
    | ShiftIn
    | ShiftOut
    | Transfer
    | Update
    | User USER_DEFINED >

identifier_event_enum=
    <  Capture
    | Control
    | ControlObserve
    | DataFromCurrentActivity
    | DataFromPriorActivity
    | DataFromCurrentAndPriorActivity
    | Hold
    | Measure
    | Observe
    | TestPatternUnit
    | Reference
    | Transfer
    | Update
    | User USER_DEFINED >

Environment { CTLMode (CTLMODE_NAME) {

    PatternInformation {                                                   (1)

      ( < Pattern | PatternBurst > (pat_or_burst_name) {                   (2)
         (Purpose (pattern_or_burst_enum)+ ;)
         ( CoreInstance (CORE_INSTANCE_NAME)+; )
         ( CycleCount integer;)
         ( Power power_expr < Average | Maximum > ; )*
         ( Fault { } )* // see below for Fault block syntax
         ( FileName FILE_NAME ;)
         ( ForeignPatterns {
             ( BlockName NAME; )
             ( BeginLine LINE_NUM; )
             ( EndLine LINE_NUM; )
             ( BeginLabel LABEL; )
             ( EndLabel LABEL; )
         } ) // end ForeignPatterns

         ( Identifiers { } )* // see below for Identifiers block syntax
         ( Protocol <Macro | Procedure> MACRO_OR_PROC_NAME (SETUP_MACRO_OR_PROC_NAME); )
```

```
        } )* // end Pattern | PatternBurst

        ( PatternExec (EXEC_NAME) {                                          (3)
            (Purpose (exec_enum)+ ;)
            ( PatternBurst (BURST_NAME)+ ; )
            ( CycleCount integer;)
            ( Power power_expr < Average | Maximum > ; )*
            ( Fault { } )* // see below for Fault block syntax
        } )* // end PatternExec

        ( < Procedure | Macro > PROCEDURE_OR_MACRO_NAME {                    (4)
            ( Purpose ( procedure_or_macro_enum )+ ;)
            ( ScanChain (CHAINNAME)+;)*
            ( UseByPattern (pattern_or_burst_enum)+; )
            ( Identifiers { } )* // see below for Identifiers block syntax
        } )* // end Procedure | Macro

        ( WaveformTable (WFT)* {                                             (5)
            (Purpose (< Shift | Capture | Data | User USER_DEFINED >)+ ;)
            ( WaveformChar (WFC)+
                (< Level | Pulse | DoublePulse | Complex >)
                ( < Critical | NonCritical >)
                ( Measure); )*
        } )* // end WaveformTable

    } // end PatternInformation
}} // end Environment, CTLMode

Identifiers (PATTERN_OR_BURST_ENUM)* {                                       (6)
    ( EventType identifier_event_enum {
        <( < Label | Xref > LABEL_ID {
            ( < Prefix | Complete > ; )
            ( <Begin | During | End> ; )
            ( SequenceNumber INTEGER; )
            ( EventValue (time_expr)+; )
        } )+// end Label | Xref
        | ( Variable (VAR_NAME (values)*); )+>
    } )* // end EventType
} )* // end Identifiers

Fault {                                                                      (7)
    ( Type
        < Toggle
        | StuckAt
        | StuckOpen
        | Transition
        | Path
        | Bridge
        | PseudoStuckAt
        | User USER_DEFINED > (<Collapsed | UnCollapsed | Estimated>);)
    ( Boundary < Cell | Primitive >; )
    ( FaultCount integer; )
    ( FaultsDetected integer ; )
    ( FaultsDetected integer {
        (Simulation integer;)
        (Implication integer;)
        (Robustly integer;)
    })

    ( FaultsPossiblyDetected integer;) // detection credit with X
    ( FaultsPossiblyDetected integer {
        (PossibleX integer ; )
        (Oscillatory integer ; )
        (Hypertrophic integer ; )
    } )
    ( FaultsUntestable integer;)
    ( FaultsUntestable integer {
        (Dangling integer;)
        (FixedValues integer;)
        (Blocked integer;)
        (Redundant integer;)
    })
    ( FaultsNotDetected integer;)
    ( FaultsNotDetected integer {
        (Uncontrolled integer;)
        (Unobserved integer;)
        (ATPGlimitations integer; )
        (Oscillatory integer ; )
        (Hypertrophic integer ; )
    })
    ( MultiplyDetected integer {
        FaultsDetected ...
        FaultsPossiblyDetected ...
        FaultsUntestable ...
        FaultsNotDetected ...
    })*
} // end Fault
```

16.2 PatternInformation block syntax descriptions

(1) **PatternInformation**: This statement begins the block of statements that contains the pattern-related information about the current configuration of the design.

(2) **Pattern | PatternBurst** *pat_or_burst_name*: This statement begins the reference to Patterns and PatternBursts of the current mode. The *pat_or_burst_name* of this statement is expected to be a valid construct in the current CTL. If foreign patterns are referred to, the *pat_or_burst_name* need not be specified. If the foreign statement is not used, the *pat_or_burst_name* must exist and the definition of the pattern or burst is expected to be found (i.e., missing definition of the construct is an error).

Purpose: This statement allows for a type assigned to the patterns.

a) **IDDQ**: Patterns that are used for measuring current.
b) **LogicBIST**: Patterns that are applied using logic BIST.
c) **MemoryBIST**: Patterns that are applied using Memory BIST.
d) **Padding**: Patterns that can be used to increase the number of patterns for synchronization.
e) **Parametric**: Patterns that support analog measurements.

f) **AtSpeed**: Patterns that are applied at speed to the product.
g) **Scan**: Patterns that use the scan structures of a product.
h) **ChainContinuity**: Patterns that check the scan structures in the product.
i) **EstablishMode**: Patterns used to configure the design in a mode.
j) **TerminateMode**: Patterns that are used to exit a configuration of the design.
k) **Endurance**: Patterns that stress the design by changing the basic parameters of a test.
l) **Retention**: Patterns that check the ability to retain state.
m) **CompatibilityInformation**: A PatternBurst that is used only to show which PatternBursts or Patterns can be executed in parallel, that is, are compatible. This PatternBurst is never executed.
n) **User** USER_DEFINED: A user-defined type for extendability of CTL. It is not allowed to use this capability for describing pattern types already covered by other keywords in this category. The user-defined name cannot be used to redefine any of the existing pattern type keywords. The user_defined name shall follow the definitions for user-defined names as defined by IEEE Std 1450-1999.

CoreInstance CORE_INSTANCE_NAME: The patterns may be applicable to one or more hierarchies in the design. This statement is used to link the patterns to the hierarchies being tested.

CORE_INSTANCE_NAME: This name shall be a valid name of a core instance defined in the current CTL. The core instance could be embedded in multiple hierarchies; in which case, its name is a concatenation of the core_instance_name's beginning from the highest level in the hierarchy. The concatenated names are separated by a ":".

CycleCount integer: This statement identifies the number of cycles (or periods) required to execute the associated pattern data.

Power: This keyword begins a statement that specifies either the average or maximum power usage of the patterns.
a) **Average**: Keyword that specifies that the power expression is for average power.
b) **Maximum**: Keyword that specifies that the power expression is for maximum power.

Fault{}: This statement begins the set of statements that add more information about the Patterns or PattternBursts being referred to. See the definition of the Fault block for details.

FileName FILE_NAME: This statement defines the name of the file that contains the patterns. The file_name shall be specified using the rules defined in IEEE Std 1450.1-2005. The file_name shall be defined in the current environment block using the FileReference statement, identifying the file as a TestPattern type.

ForeignPatterns: This keyword begins the block of statements to identify the set of test patterns that are representative of this block of patterns but are not in the CTL language. As this construct ties in patterns that do not follow the rules defined by CTL, the tools support for the identified patterns may not be available.
a) **BlockName** BLOCK_NAME This statement defines which block of patterns to use from the foreign file. This is applicable to pattern file formats that have structure and allow patterns to be grouped into blocks or clauses.
b) **BeginLine** LINE_NUM: This statement is used to define the beginning line number of the patterns. The line_num is an integer and is inclusive of the line being referred to. If missing and there is no Begin Label statement, then the beginning of the file is to be assumed. An error is to be assumed if both Begin Line and Begin Label are used.
c) **EndLine** LINE_NUM: This statement is used to define the ending line number of the patterns. The line_num is an integer and is inclusive of the line being referred to. If missing and there is no End Label statement, then the end of the file is to be assumed. An error is to be assumed if both Begin Line and Begin Label are used.
d) **BeginLabel** LABEL: This statement is used to define the beginning of the patterns referred to by this block of statements. The line beginning with a unique label (unique to the file) defines the first

pattern of the set of patterns being referred to. The label is an existing label in the file being referenced. This statement cannot be used in conjunction with the BeginLine statement. If BeginLabel and BeginLine are missing, the start of the file is the beginning of the patterns of this block.

e) **EndLabel** LABEL: This statement is used to define the end of the patterns referred to by this block of statements. The line beginning with a unique label (unique to the file) defines the last pattern of the set of patterns being referred to. The label is an existing label in the file being referenced. This statement cannot be used in conjunction with the EndLine statement. If EndLabel and EndLine are missing, the end of the file is the end of the patterns of this block.

Identifiers {}: This statement begins the set of statement that add more information about the Patterns or PattternBursts being referred to. See the definition of the Identifiers block for details.

Protocol <**Macro** | **Procedure**> MACRO_OR_PROC_NAME SETUP_MACRO_OR_PROC_NAME: Through this statement, the protocol (Macro or Procedure) invoked by the patterns is defined. This is required to match the information in the PatternBursts that invoke the associated Patterns. As patterns are required in CTL to call only one type of protocol, this statement does not allow more than one macro or procedure to be identified. This statement is required if the information is not in a PatternBurst. The MACRO_OR_PROC_NAME represents the name of the Macro/Procedure that is to be used by the P statement in the associated Pattern. The SETUP_MACRO_OR_PROC_NAME represents the name of the Macro/ Procedure that is to be used by the Setup statement in the Pattern.

(3) **PatternExec** EXEC_NAME: The top-level construct in STIL that contains a set of patterns/sequences for the configuration of the design being described. The EXEC_NAME is a user-specified name that shall follow the rules of IEEE Std 1450-1999. This statement is a reference to the PatternExec defined outside the Environment block of statements. An error is to be assumed if the definition is not found.

Purpose *exec_enum*: The statement that describes the content of the PatternExec referred to by the PatternExec statement.

a) **Diagnostic**: Patterns/Sequences of the design configuration that are to be used for Diagnostics.
b) **Production**: Patterns/Sequences of the design configuration that are to be used for Production.
c) **Characterization**: Patterns/Sequences of the design configuration that are to be used for Characterization.
d) **Verification:** Patterns/Sequences of the design configuration that are to be used for the validation of the patterns. Validation typically involves a full timing-based simulation to verify the validity of expected responses of the test patterns.

PatternBurst burst_name: The highest level pattern bursts in the calling hierarchy of the pattern exec that are to be executed for the current design configuration (test mode). If no PatternBurst statement exists, then all patterns in the PatternExec are to be executed.

Power: This keyword begins a statement that specifies either the average or the maximum power usage of the patterns.

a) **Average**: Keyword that specifies that the power expression is for average power.
b) **Maximum**: Keyword that specifies that the power expression is for maximum power.

Fault{}: This statement begins the set of statements that add more information about the Patterns or PattternBursts being referred to. See the definition of the Fault block for details.

CycleCount integer: This statement identifies the number of cycles (or periods) required to execute the associated pattern data.

(4) **Procedure** | **Macro** PROC_OR_MACRO_NAME: This statement begins the reference to the Macro or Procedures that exist in the scope of the current CTLMode block as defined by the DomainReferences. That

is, the PROC_OR_MACRO_NAME of this statement is expected to be a valid construct in the MacroDefs/ Procedures blocks included in the test mode.

Purpose: This statement allows for a type assigned to the procedure or macro.

a) **Control**: A sequence that allows for stimulus values to be put on a set of nets in the design.
b) **Observe**: A sequence that allows for the values on a set of nets of the design to be externally available.
c) **ControlObserve**: A sequence that allows for the control and observe as defined by the Control keyword and the Observe keyword.
d) **DoTest**: A sequence or set of sequences that applies a complete test pattern (test pattern unit). A complete test pattern typically comprises a scan load; some number of clocks that either advance the data or launch/capture the data, followed by a scan unload. This type of test pattern sequence is preferred over DoTestOverlap.
e) **DoTestOverlap**: A sequence or set of sequences that applies a complete test pattern (test pattern unit). Overlapped test pattern protocols represented by this keyword allow for the Scan Operation of adjacent test patterns to occur simultaneously. The overlapped test protocol is required to take in a redundant parameter of the output signals that was observed by the previous test pattern but could be overlapped if scanned later.
f) **Operate**: A sequence that when executed makes the associated hardware perform its function.
g) **Hold**: A sequence that allows for the state of the design to be suspended such that the following patterns can be applied without any disruption.
h) **Transparent**: A sequence that allows for values to pass through certain logic and are available intact with maybe an inversion at the end of the sequence.
i) **Instruction**: A sequence that is used to operate the instruction register of a design.
j) **Capture**: A sequence that allows for some values on the nets of a design to be captured in memory elements.
k) **Update**: A sequence that allows for some values to be made available to a portion of the design after a period of isolation.
l) **ModeControl**: A sequence that is used to configure the design into a mode.
m) **MemoryPrecharge**: A sequence that precharges a memory.
n) **MemoryRefresh**: A sequence that refreshes a memory.
o) **MemoryRead**: A sequence that allows for the access of values in a memory.
p) **MemoryWrite**: A sequence that allows for values to be stored into a memory.
q) **MemoryReadModifyWrite**: A sequence that allows for values to be read and then written into a memory.
r) **Launch**: A sequence that performs a circuit setup and then launches values from memory elements onto certain nets in a design. This is used in test patterns that are detecting transition or path delay faults.
s) **Transfer**: A sequence that transfers data from a memory element that is not part of the scan chain to one that is, or from a memory element that is part of the scan chain to one that is not.
t) **ShiftIn**: This is similar to the Control purpose except that the values are placed on nets in the design by only using a scan shift mechanism.
u) **ShiftOut**: This is similar to the Observe purpose, except that the values are made externally available by only using a scan shift mechanism.

ScanChain (CHAINNAME)+: This statement lists the scan chains that are operated by the Procedure or Macro. The list of chains in this statement identifies the scan chains that operate in parallel. Multiple use of this statement identifies the different parallel sets of scan chains operated by the protocol.

UseByPattern *pattern_or_burst_enum*: Through this statement, the possible use of the current sequence is specified. The type of patterns that can call this Macro or Procedure is specified as part of the pattern_or_burst_enum.

Identifiers {}: This statement begins the set of statements that add more information about the Procedures or Macros being referred to. See the definition of the Identifiers block for details.

(5) **WaveformTable** (WFT)*: This statement begins the block of statements that defines the conventions used to define the timing information in CTL. The WFT (table name) is a reference to the table name that is within the scope of the current CTLMode block of statements as defined by the DomainReferences. The CTL is erroneous if the table name is not found within the scope of the test mode being described. If no table name is specified in this statement, the convention is used across all of the timing information available in the current CTLMode block of statements.

Purpose: This statement defines the purpose of the WaveformTable and is assigned one of the following enumerated identifiers:

a) **Shift**: This waveform table is used when shifting data in registers or scan chains.
b) **Capture**: This waveform table is used when capturing data into registers or scan chains.
c) **Data**: This waveform table is used when applying test data to data inputs of a design.
d) **User** USER_DEFINED: A user-defined usage of the usage of the waveforms defined in the WaveformTable. The user-defined name should follow the naming conventions defined in IEEE Std 1450-1999 for names. This name cannot be the same as the keywords already allowed in this statement.

WaveformChar:

a) **Level**: This WaveformChar represents a single data value or signal level.
b) **Pulse**: This WaveformChar is used to represent a pulse waveform.
c) **DoublePulse**: This WaveformChar is used to represent a double pulse waveform.
d) **Complex**: This WaveformChar is used to represent a complex waveform that cannot be described as a Level, Pulse, or DoublePulse.
e) **Critical**: The timing described by this WaveformChar is critical timing.
f) **NonCritical** : The timing described by this WaveformChar is noncritical.
g) **Measure**: This WaveformChar is used when comparing the measured value on an output or bidirectional signal.

(6) **Identifiers** (*pattern_or_burst_enum*)*: The identifiers block of statements is used to add additional information to identifiers or events for constructs that allow the Identifier statement (such as Procedures, Macros, Patterns, or PatternBursts). Identifiers allows for information about special labeled/tagged statements to describe special events. Through the pattern_or_burst_enum, the information is made relevant to specific types of patterns and bursts. The definition of the types of patterns or bursts identifiable in CTL is provided in the definition of the Purpose statement for Pattern and PatternBurst. The identifier's information is meta information on the associated construct to provide meaning that is appropriate to activities such as synchronization of patterns from multiple embedded designs on the SoC. In addition to information on labels, the identifiers can provide information on Variables used in the associated construct (e.g., macro or procedure).

EventType *identifier_event_enum*: Several events are identifiable through this statement to be associated with either a label or a variable but not both. The events (listed below) are valid for the associated construct (for example, Macro or Procedure) for use in patterns that apply to the pattern_or_burst_enum in which this statement occurs. When the event is associated with a variable, the Variable statement is used. When the event is associated with a statement in the associated construct (such as macro), a label on the statement is identified. The event type cannot be used to simultaneously define information about variable contents and statements. The following are valid identifier event enums:

a) **Capture**: This event type associates the identifier with an activity where a value is stored in a memory element broadside.
b) **Control**: This marks an event where a known value can be set in a memory element. Shift-in for scan elements is a special case for control.
c) **Observe:** An event that allows for the values on a set of nets of the design to be externally available.

d) **ControlObserve:** This event type associates the identifier with an activity that both controls and observes internal memory elements of a design as defined by both the Control and the Observe event types.
e) **DataFromCurrentActivity:** Protocols are invoked multiple times sequentially by patterns. Each invocation is a test pattern unit. Due to optimizations such as overlapping the scan operation of one test unit with an adjacent test unit, the data that are applied in any invocation may/may not be associated with the current test unit. DataFromCurrentActivity is used to identify the fact that the data being used are associated with the current invocation. When used as information on a variable, all contents of the variable are from the current test pattern unit. When associated with a statement, all values used in the statement are from the current test pattern unit.
f) **DataFromPriorActivity:** This event is used to specify that the data being used are associated with the prior invocation of the protocol. When used as information on a variable, all contents of the variable are from the previous test pattern unit. When associated with a statement, all values used in the statement are from the previous test pattern unit.
g) **DataFromCurrentAndPriorActivity:** This event specifies that the data being used are associated with both the prior and the current invocation of the test protocol.
h) **Hold**: This event identifies the ability to insert more clock periods without corrupting the intended function of the protocol. In conjunction with the Begin, During, or End statement, the location of the new clock periods is identified. Begin defines the additional cycles to be inserted before the beginning of the identified statement. During (only valid for complex statements like Shift) identifies the ability to insert additional clock cycles between any two clock cycles of the multicycle statement. End defines the additional cycles to be inserted after the execution of the identified statement.
i) **Measure:** This event identifies statements where a measurement of some value is to be performed. For example, this statement would identify where an IDDQ measurement would take place.
j) **TestPatternUnit**: A test pattern unit is a complete set of stimulus, responses, and sequencing of the associated values such that failures are tested by the entity. This is typically associated with an ATPG tools output that includes tests for faults such as the stuck-at faults or transition faults. The test pattern unit in this case would include a scan-in/scan-out operation and some stimulus and capture events. BIST methods typically loop over multiple test pattern units. Functional tests have test pattern units that do not involve scan operations. This keyword identifies this entity.
k) **Transfer**: This specifies an event that transfers values from the off-shift path memory elements to the shift path.
l) **Update**: This specifies an event that updates memory elements off the shift path with value from the shift path.

Label | Xref LABEL_ID: Through this statement, a label or tag (LABEL_ID) is identified in the associated construct and information is provided about the statement associated with it. The information provided through this block of statements is only to be interpreted when the label or tag matches the label or tag in the associated construct (such as Macro or Procedure). Depending on the use of PreFix or Complete defined below, the label or tag can be a subset or exact match. If this statement exists and no matching identifier is found in the associated construct, the information is redundant and meaningless and is an error condition.

a) **PreFix** : Specifies that the label_id is only a prefix. All labels that start with this prefix are identified.
b) **Complete**: Identifies the label_id as being complete. Only a label that exactly matches the specified label_id is identified. This is the default condition.
c) **Begin**: Specifies that the identifier event occurs at the beginning of the execution of the associated statement.
d) **During**: Specifies that the identifier event occurs during the associated statement. This is the default condition.
e) **End**: Specifies that the identifier event occurs at the end of the associated statement.

f) **SequenceNumber** integer: This statement is used to provide a relative ordering of identifiers that is global across all identifiers of a certain pattern-enum and event type in a mode. One can provide Iddq measure points with relative priority.

g) **EventValue** (time_expr)+: This statement is used to provide some numeric value that is then associated with the identifier event. For example, for a Measure event, this could specify Iddq current to be measured when used within a PatternType of Iddq.

Variable (VAR_NAME (values)*): This identifies the EventType to be associated with a variable instead of a Label or Xref. This statement is followed by an optional variable name (VAR_NAME) used in the associated construct (such as Macro, Procedure). The VAR_NAME shall be a valid VAR_NAME in the scope of the current test mode as defined by the DomainReferences. If no variable name is identified, all variables in the associated construct (e.g., Macro) would be associated with the EventType.

The variable name can be followed by an optional list of values that the variable can take on. Variables hold waveform characters that represent logic values with timing information. The values in this statement represent the logic levels that the waveforms achieve. The values can be specified with strings created by using 1, 0, and x, where each position in the string represents a single bit value. (Example use scenario: The associated variable could reside in an instruction register of the IEEE 1500 architecture. In that case, the values specified by the string provide the valid instructions for that register.)
EventTypes that are allowed from the pre-defined enumerations to be used for variables are: DataFromCurrentActivity, DataFromPriorActivity, and DataFromCurrentAndPriorActivity.

(7) **Fault**: This statement begins the optional block of statements that can be used to define the fault coverage information for a set of Patterns or PatternBursts. Multiple fault blocks can be used to describe fault coverage information for different fault measurements.

Type: This statement declares to which type of fault the fault coverage information relates. All information contained in this Fault block relates only to this fault type. If a set of Patterns or PatternBursts can have fault coverage information for more than one fault type, then multiple Fault blocks should be used. The default fault type is uncollapsed stuck-at faults.

a) **Toggle**: The toggle fault type indicates that detected nodes can be driven to both a high and a low state.
b) **StuckAt**: The single stuck-at fault type indicates that detected nodes can be driven to both a high and a low state, and the result of these states can be properly observed.
c) **StuckOpen**: This fault type is used to model a single broken line in CMOS circuits such that the faulty circuit exhibits a memory effect. Multiple stuck-at tests are required to cover a stuck-open fault.
d) **Transition**: The transition fault type is used to model large delay defects on nodes in the design.
e) **Path**: The fault type is the path delay fault that is used to model delay defects along a specific path in the design. This fault type does not have a specific fault site, but it is associated with the entire path.
f) **Bridge**: The bridge fault type is used to model the effect of two nodes in a design being shorted together.
g) **PseudoStuckAt**: The pseudo stuck-at fault type is similar to the stuck-at fault type except that the change of state needs only to be observed at the output of cell boundaries. This is used mainly for IDDQ tests.
h) **User** USER_DEFINED.

< Collapsed | UnCollapsed | Estimated >: This optional clause for the **Type** statement further clarifies the fault type. The fault information can be collapsed, uncollapsed, or estimated.

Boundary < **Cell** | **Primitive** >: This statement specifies whether faults in the core are inserted only on the cell boundaries or if they are also inserted on primitives within cells.

FaultCount *integer*: Specifies the total number of faults of the specified fault type present in the core.

FaultsDetected *integer*: Specifies the total number of faults of the specified fault type detected by the Pattern or PatternBurst. The number of detected faults can be further divided into subcategories used to specify the detection method. The total count for the subcategories should equal the number of faults specified here.

a) **Simulation** *integer*: A subcategory of **FaultsDetected** that enumerates the number of faults explicitly detected by simulation.
b) **Implication** *integer*: The number of faults with detection credit given by *a priori* markoff, for example, faults detected by shifting of the scan chains.
c) **Robustly** *integer*: This subcategory can only be used for the **Path** fault type. This indicates that the path delay fault type was detected independent of glitches.

FaultsPossiblyDetected *integer*: This statement indicates the number of faults with possible detection credit. This fault category can be divided into subcategories. The total count of faults listed in the subcategories should equal the number of faults specified here.

a) **PossibleX** *integer*: A subcategory of **FaultsPossiblyDetected** that specifies the number of faults given possible detect credit due to a low to unknown or a high to unknown difference at the observe point.
b) **Oscillatory** *integer*: This statement lists the number of faults that have an unstable circuit status. These faults would require a great deal of computation to calculate for detection status. Note that this subcategory can be used as a **FaultsPossiblyDetected** subcategory or as a **FaultsNotDetected** subcategory.
c) **Hypertrophic** *integer*: The hypertrophic fault subcategory enumerates faults that cause extensive areas of the entire design to diverge from good state status. These faults would also require a great deal of computation to calculate the detection status. Note that this subcategory can be used as a **FaultsPossiblyDetected** subcategory or as a **FaultsNotDetected** subcategory.

FaultsUntestable *integer*: This is a category of faults that cannot be detected by the ATPG tool that was used. The tool has exhausted its solution space and has proved that the fault cannot be detected. This fault category can be divided into subcategories. The total count of faults listed in the subcategories should equal the number of faults specified here:

a) **Dangling** *integer*: Disconnected logic (input or output)
b) **FixedValues** *integer*: Faults that are on logic that have fixed values
c) **Blocked** *integer*: Faults that cannot be detected because of fixed values in logic
d) **Redundant** *integer*: Cannot be controlled and observed simultaneously

FaultsNotDetected *integer*: This category contains the faults that were not detected but were not proven untestable. This fault category can be divided into subcategories. The total count of faults listed in the subcategories should equal the number of faults specified here.

a) **Uncontrolled** *integer*: Could not excite the fault
b) **Unobserved** *intege* : Fault is excited, but could not observe effects of the fault
c) **ATPGlimitations** *integer*: atpg completed search, but some other method could possibly detect the fault
d) **Oscillatory** *integer*: See above
e) **Hypertrophic** *integer*: See above

MultiplyDetected *integer*: The MultiplyDetected block is used to specify the fault information when multiple detect fault simulation is used. The integer specifies the number of times faults are detected. Within this block, the **FaultsDetected, FaultsPossiblyDetected, FaultsUntestable,** and **FaultsNotDetected** statements or blocks are used to specify the fault information numbers, which are multiply detected the number of times specified.

16.3 PatternInformation block syntax example

Example showing multiple CTLMode blocks with pattern information. Note that several details are not shown. For example, Patterns and Timing are not shown. Statements are also incomplete as marked with "...":

```
Procedures {
   mode_setup {
     W tp1;
     V {...} // details not shown
   }
}
PatternBurst mode1_burst {
   PatList {
     begin_mode1 { Protocol Procedure mode_setup;}
     mode1_test { Protocol ... ;} // details not shown.
   }
}
PatternBurst isolate_burst {
   PatList {
     isolate_pat {Protocol ... ;} // details not shown
   }
}
PatternBurst Top {
   PatSet {
     mode1_burst;
     isolate_burst;
   }
}
PatternExec {
   PatternBurst Top;
}

Environment {
   CTLMode {
     ... // details not shown
   }
   CTLMode mode1 {
     TestMode InternalTest;
     PatternInformation {
       PatternExec CoreExec {
         Purpose Production;
         PatternBurst mode1_burst;
         Fault {
           Type StuckAt;
           FaultCount 2000;
           FaultsDetected 1900 {
             Simulation 1850;
             Implicatin 50;
           }
           FaultsUntestable 20;
           FaultsNotDetected 80;
         }
       }
```

```
        Procedure mode_setup {
          Purpose ModeControl;
        }
        Pattern begin_mode1 {
          Purpose EstablishMode;
          Protocol Procedure mode_setup;
        }
      }
    }
    CTLMode isolate{
      TestMode Isolate;
      PatternInformation {
        PatternExec CoreExec {
          Purpose Production;
          PatternBurst isolate_burst;
        }
        Procedure mode_setup {
          Purpose ModeControl;
          Protocol Procedure mode_setup;
        }
        Pattern isolate_pat{
          Purpose EstablishMode;
        }
      }
    }
}
```

Example showing Macro purpose and Identifiers use the following:

```
MacroDefs {
   load_unload {
     W shift_tp;
     V { "CLK" = 0; "SE" = 1;}
     Shift { V { _si_ = #; _so_ = #; "CLK" = P;} }
   }
   launch_cap {
     W capture_tp;
     launch: V { "SE" = 0; _pi_ = %; "CLK" = P; }
     capture: V { _po_ = %; "CLK" = P; }
   }
   capture {
     W shift_tp;
     capture: V { "SE" = 0; _pi_ = %; _po_ = %; "CLK" = P; }
   }
   basic_pat {
     Macro load_unload { _si_ = #; _so_ = #; }
     Macro capture { _pi_ = %; _po_ = %; }
   }
   transition_pat {
     Macro load_unload { _si_ = #; _so_ = #; }
     Macro launch_cap { _pi_ = %; _po_ = % ; }
   }
}
```

```
Environment unwrapped_design {
   CTLMode {
     ... // details not shown
   }
   CTLMode internal_test {
     TestMode InternalTest;
     PatternInformation {
       Macro load_unload {
         Purpose ControlObserve;
       }
       Macro launch_cap{
         Purpose Launch Capture;
         Identifiers {
           EventType Launch {
             Label launch { Complete; During; }
           }
           EventType Capture {
             Label capture { Complete; During; }
           }
         }
       }
       Macro capture{
         Purpose Capture;
         Identifiers {
           EventType Capture {
             Label capture { Complete; During; }
           }
           EventType Hold {
             Label capture { Complete; End; }
           }
         }
       }
       Procedure test_setup {
         Purpose ModeControl;
         UseByPattern test_setup_pat;
       }
       Macro basic_pat{
         Purpose DoTestOverlap;
         Identifiers {
           EventType DataFromPriorActivity {
             Variable _so_;
           }
         }
       }
       Macro transition_pat{
         Purpose DoTestOverlap;
       }
     }
   }
}
```

Example showing complex fault information, as follows:

```
Environment {
   CTLMode mode_A {
     TestMode InternalTest;
     PatternInformation {
       PatternExec CoreExec {
         Purpose Production;
         PatternBurst modeA_burst;
         Fault {
           Type StuckAt;
           FaultCount 2000;
           FaultsDetected 1900 {
             Simulation 1850;
             Implication 50;
           }
           FaultsUntestable 20 {
             FixedValues 10;
             Redundant 10;
           }
           FaultsNotDetected 70 {
             Uncontrollable 30;
             Unobservable 40;
           }
           FaultsPossiblyDetected 10;

           MultiplyDetected 2 {
             FaultsDetected 1700;
             FaultsUntestable 20;
             FaultsNotDetected 180 {
               Uncontrollable 100;
               Unobservable 80;
             }
           }
           MultiplyDetected 3 {
             FaultsDetected 1500;
           }
         }
       }
     }
   }
}
```

Example showing use of WaveformTable block, as follows:

```
Timing CoreTiming {
   WaveformTable tset_1 {
     Period ‘100ns’;
     Waveforms {
       _pi_ { 01X { ‘0ns’ D/U/N; }}
       _po_ { LHXZ { ‘10ns’ lhXt; ‘20ns’ X; }}
       slow_clk { 0P { ‘0ns’ D; ‘20ns’ D/U; ‘80ns’ D; }}
       fast_clk { 0F { ‘0ns’ D; ‘20ns’ D/U; ‘30ns’ D; ‘40ns’ D/U; 50ns
D;}}
     }
   }
}
Environment {
   CTLMode mode_A {
     TestMode InternalTest;
     DomainReferences { Timing CoreTiming; }
     PatternInformation {
       PatternExec CoreExec {
         Purpose Production;
         PatternBurst modeA_burst;
         WaveformTable tset_1 {
           Purpose Capture;
           WaveformChar P Pulse NonCritical;
           WaveformChar F DoublePulse Critical;
         }
       }
     }
   }
}
```

Index