CTL the Language for Describing Core-Based Test

Rohit Kapur
Synopsys Inc.

Maurice Lousberg
Philips

Tony Taylor
Synopsys Inc.

Brion Keller
IBM

Paul Reuter
Mentor Graphics

Douglas Kay
Cisco

Abstract

As part of an industry wide effort IEEE is in the process of standardizing the elements of test technology such that plug & play can be achieved when testing SoC designs. This standard under development is a language namely, Core Test Language (CTL), which is introduced in this paper. CTL describes all necessary information for test pattern reuse and the needs of test during system integration. CTL syntax and its link to STIL are explained with examples.

1 Introduction

Design Reuse Methodologies [1] 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 across teams that are in close proximity to each other, the teams can get together with frequent formal meetings, share common documentation or discuss problems informally in coffee breaks. 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 partition (or core) providers and the integrators needs to be standardized. This is where industry-wide standards step in. A Core Test Language (CTL) is being developed to address the interoperability needs of SoC test [2]. IEEE P1500 [3][4] is proposing a standard wrapper with a CTL interface to isolate the cores from the embedded environment. As a precursor to this, the VSI Alliance [5] is defining a similar hardware architecture to be replaced by the P1500 hardware when it becomes an IEEE Standard.

This paper is focusses on the Core Test Language (CTL). Through this mechanism 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 the usual test tasks as though the core was a white box with test patterns to be reused. As shown in Figure 1, CTL describes all the 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.

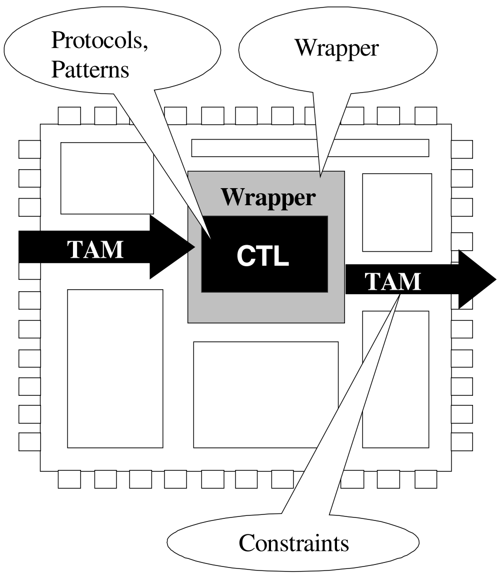


Figure 1: *Usage of the CTL representation of the core in the creation of a generic test access architecture.*

Figure 1 shows a usage of CTL where the information about the core is represented in CTL. Using the CTL of the core a wrapper can be constructed, and the appropriate Test Access Mechanism (TAM) can be determined based upon the test constraints in the CTL of the core [6][7][8]. Once all the structures are in place the test patterns that are also a part of CTL can be re-targeted to the boundary of the SoC. In this example the core described by the CTL did not have a wrapper inside the core.

CTL is the language to support all the information that the core provider needs to give the system integrator such that the integrator can successfully test the embedded core and any user-defined logic around the core. This language is broad enough to describe P1500, VSI-A and even IEEE 1149.1 [9] hardware as described in BSDL [10]. The language constructs being 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 SoC.

Since CTL is the formal means by which a core provider supplies test information to the system integrator, it is important for this mechanism to not restrict the way cores are designed. CTL is designed to work with cores that come with any type of DfT methodology. Furthermore, CTL also works with any type of test methodology (structural tests that use the stuck-at or delay fault model, Iddq tests and functional tests).

In this paper we show how aspects of the information that describes the test details of a core are presented in CTL. Since CTL builds on STIL [11], a brief introduction is given in Section 2 on some of the concepts in STIL needed to understand the examples of this paper. In Section 3 the overall structure of the information in CTL is described. CTL information includes

- Design Configuration Information.
- Structural Information.
- Test Pattern Information.

These concepts are described in Sections 4, 5, and 6 respectively. The details are explained using a running example of a wrapped core right through the paper. At times information is replicated across CTL code to highlight concepts. The example is the unified CTL across the different sections describing the wrapped core. The unification of the different pieces of CTL will result in some redundancies which are to be removed in the unification of the code.

The full details of CTL syntax are avoided in order to highlight the main features of the language. *Bold words highlight the keywords that have a special meaning that is defined by the standards which in this case are CTL and IEEE 1450.* No attempt is made in this paper to provide all the associated parameters of the keywords that are presented.

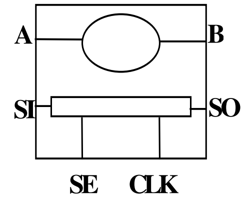
2 Some Basics of STIL

STIL constructs are the building blocks of CTL. STIL being a test pattern language, it is very good for defining all of the sequence information that needs to be described in CTL. In this section some basics of STIL are discussed such that the CTL examples in this paper can be easily understood.

2.1 Signals

The I/O signals of a core define the boundary to which test patterns are written. STIL defines a block of statements named Signals, within which the interface of the design is specified.

As shown in Figure 2, along with the signal name, a direction can be assigned to the signal to describe its use as Input (*In*), Output (*Out*) or bi-directional (*InOut*, not shown in Figure 2). The I/O signal names must match the names of the pins of the core block as defined by its netlist.




```
Signals {
  A In;
  SI In;
  SE In;
  CLK In;
  B Out;
  SO Out;
}
```

Figure 2: *Example core without a wrapper and the associated signals definition.*

2.2 Patterns

Patterns define the stimuli to and responses from the core occurring in a sequence. That is, patterns are sequences of vectors that contain logic values to be applied to Signals.

Figure 3 is an example pattern (all_pats) that applies a scan operation in a flattened format where values are applied to the input SI repeatedly with the right values for the conditioning signal SE and clock CLK. After scan, input A is applied some stimulus, and a response measured on signal B. After a single clock pulse, a scan operation is performed by repeated measures of the SO signal and application of the CLK with signal SE conditioned to allow for the scan operation. The example introduces syntax for the vectors, namely *V*. Each *V* statement defines one test vector that defines the state to be applied to each signal. It should be noted that this example ignores details of the overlapped operation of scan in and scan out across consecutive test patterns and waveform information required for patterns.

```
Pattern all_pats {
  V { SE=1; CLK=0; }
  V { SI=1; CLK=1;} V {SI=0; CLK=1;}
  V { SE=0; CLK=0; A=1; B=0; }
  V { CLK=1;}
  V { SE=1; CLK=0;}
  V { SO=1; CLK=1;} V{SO=1; CLK=1;}
  V { SE=0; CLK=0;}
}
```

Figure 3: *Example pattern of an ATPG generated single test where the scan-in and scan-out operation are not overlapped.*

2.3 Macros

The test pattern of Figure 3, can be also be written as shown in Figure 4 using a Macro definition.

```
MacroDefs {
  seq {
    C { SE=1; CLK=0; }
    Shift { V { SI=#; CLK=1;}}
    V { SE=0; CLK=0; A=#; B=#; }
    V { CLK=1;}
    C { SE=1; CLK=0;}
    Shift { V { SO=#; CLK=1;}}
  }
}
Pattern all_pats {
  Macro seq { A=1; SI=10; B=0; SO=11;}
}
```

Figure 4: *Example of a single ATPG test pattern where the protocol information is separated from the data portion of the test.*

The pattern of Figure 4 is written to use a protocol or template called a Macro (named seq) that applies the data defined by the pattern in a certain sequence. The *C* statement is a Condition statement that defines the assumed value on signals that are not specified. The first *Shift* statement is used to apply the two values applied to SI as defined by the pattern. Similarly, the second *Shift* statement is used for SO the scan out. Since this example is the same as that in Figure 3, it also ignores details of the overlapped operation of scan in and scan out across consecutive test patterns and waveform information required for patterns.

2.4 Timing

The above explanations used 0 and 1 as logic values. However, these values are applied with certain timing relative to a clock period and are not simple logic values. In the parlance of STIL, these are waveform characters that are defined in a waveform table on a per signal basis.

Figure 5, shows an example definition of the waveforms for the running example in this section. The waveform characters "0" and "1" are assigned values with timing information. In the example, 0 and 1 for all logic signals is a logical 0 and 1, where the value is applied at the beginning of the period and is held to that value. However the same waveform characters were used differently for clock signals. A waveform character 1 was used for the CLK signal as a pulse that happens somewhere within the clock period (a better name would have been P). An example definition of the 1 for the clock is shown in Figure 5 as a pulse of width 5ns that happens after 60ns in a clock period of 100ns. Figure 5 uses *D* and *U* to represent drive events for "down" and "up". Similarly, *L* and *H* are used for expect events for "low" and "high" values. The waveform characters 0 and 1 are also defined for the other signals for completeness (though the waveforms are only drawn for the CLK 1, and signal A being a logic 0 and logic 1).

```
WaveformTable time1 {
  Period '100ns';
  Waveforms {
     CLK { 1{'0ns' D; '60ns' U; '65ns' D}}
     CLK { 0{'0ns' D;}}
     'A+SE+SI' { 01 {'0ns' D/U;}}
     'B+SO' { 01 {'0ns' L/H;}}
  }
}
```

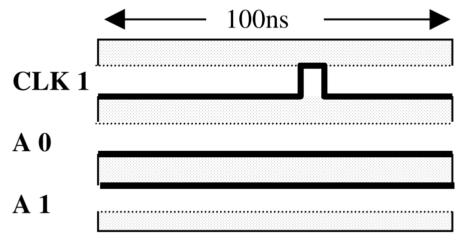


Figure 5: *Waveform Tables defining the waveform characters used in the patterns and macros.*

Now that the waveform table concept has been introduced, the test patterns of Figure 3 and Figure 4 should be augmented with a statement that references the constructs in Figure 5 (***W*** *time1;*).

3 CTL

Information in CTL is organized around a mode (configuration) of the core being represented. Figure 6, is a pictorial representation of the CTL structure.

The signals of the core being described are global across all modes. In each mode the signals are used in different ways and are attributed to describe the different needs of the configuration. Every mode of the core has a mechanism to initialize the mode. The initialization sequences are described in CTL on a per mode basis. Some modes of the core contain test patterns with their associated timing information, constraints, and statistics. Some modes contain structural information such that the structure can be used to create patterns at another level of integration of the design. CTL is designed to describe all these needs of different modes of the core. For any given test mode of a core, typically only a subset of the full CTL syntax will be required to adequately describe the attributes of the mode.

Using the basic constructs identified in Figure 6, CTL has the ability to describe the following.

- Controls to configure the core for testing the core and the surrounding SoC logic.
- Requirements and constraints on the implementation of SoC-level interfaces to the core.
- Inclusion of test data specific for the core, but defined independently from any particular use of the core.

These allow for complete representation of the information needed to test the core within an SoC.

The Language: The primary task of CTL is to describe information that allows for the reuse of test patterns. This brings a natural affinity for Test Patterns to CTL. IEEE has standardized a format called STIL to represent test patterns. The standard is known as IEEE 1450 [11]. It became the natural choice for the syntax for CTL information. CTL builds on STIL by leveraging off pattern information available in STIL whenever appropriate without replicating or creating any new syntax for the same. Since CTL describes a number of concepts that a test pattern language is not intended to describe, new syntax is created for CTL in a STIL-friendly way.

CTL can be viewed as a superset of STIL in terms of the syntax. Furthermore, some constructs in CTL can also be viewed as a meta-language of STIL when it comes to describing the patterns since it defines how the test data described in STIL should be interpreted.

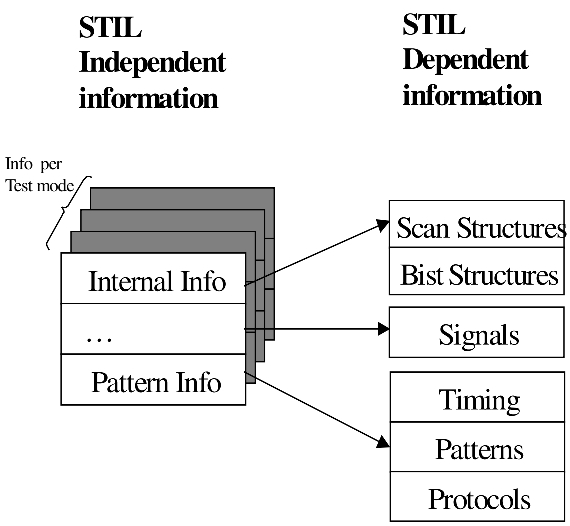


Figure 6: *CTL structure depicting multiple test modes of a single core.*

CTL Syntax: For the scope of the examples in this paper, CTL information for a mode is partitioned into two blocks namely, Internal and PatternInformation (shown in Figure 6).

1. The *Internal* block of statements provides information describing the core boundary relative to the I/O signals of the core. For example, a signal used as a clock by the core would be described in an *Internal* block.
2. The *PatternInformation* block of statements describes the test patterns and related constructs needed for the test mode.

There are numerous other blocks of statements in CTL that are not covered, since the complete syntax of CTL is not the focus of this paper.

While every syntax description of STIL is part of CTL, there is a distinct difference in the way the pieces of the CTL information are represented relative to the origin of the syntax itself. Figure 6, depicts the relationship between the STIL portion of the CTL syntax and the syntax that is not part of STIL.

4 CTL for Design Configurations

A typical SoC's design for testability (DFT) would follow a methodology where individual cores are isolated from other logic during test of the SoC. This isolation can be accomplished by using a boundary-scan like wrapper added to the periphery of the core.

Once bounded, the new wrapped core would support multiple modes (configurations) to allow for internal testing of the original core embedded in its wrapper and the test of the logic external to the wrapped core. (*Note*: *Although a wrapper is used in the example, CTL is not limited to describing wrapped cores*.)

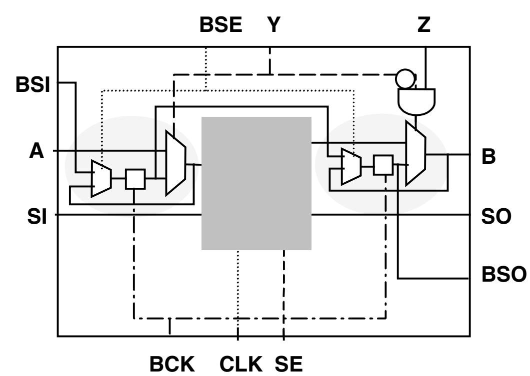
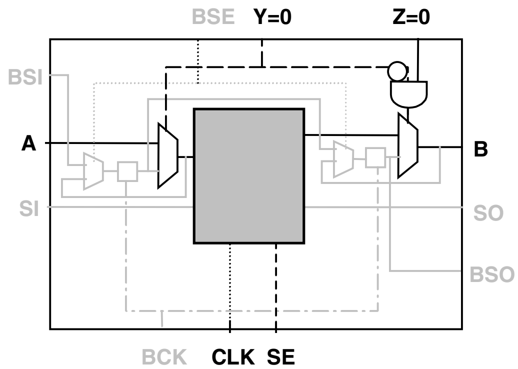


Figure 7: *An example of a core with a wrapper. The core has multiple modes of operation.*

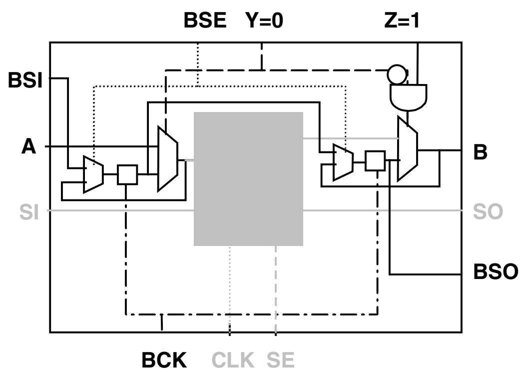
Figure 7 shows a wrapped core supporting multiple modes of operation. In this example the shaded box represents the original (unwrapped) core as an embedded core of the wrapped core. The picture represents the core with an internal wrapper built using flip-flops, multiplexers and some gating logic. *The context of this example is the complete core, which includes the wrapper*.

Figure 8, is a pictorial view of different modes of the core. The different modes of the wrapped core are to be described in CTL. Figure 9 shows partial CTL code for the associated modes of Figure 8.

A) *Normal Mode.*



B) *External Test Mode.*



C) *Internal Test Mode.*

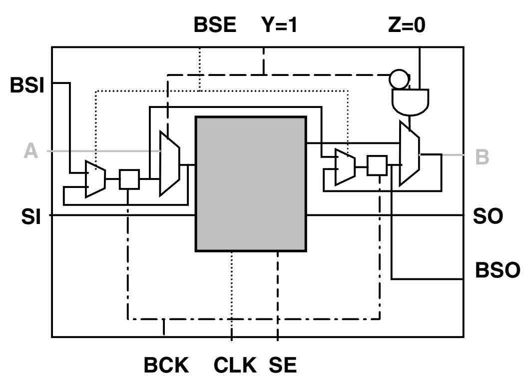


Figure 8: *Modes of operation of the wrapped core to show three configurations A) Normal Mode B) External Test Mode and C) Internal Test Mode*

```
Signals {
  A In; B Out;
  Y In; Z In;
  SE In; BSE In;
  CLK In; BCK In;
  SI In { ScanIn 2; }  SO Out { ScanOut 2; }
  BSI In { ScanIn 2; } BSO Out { ScanOut 2; }
}
```

```
Timing T1 {
  WaveformTable W1 {
    Period '100ns';
    Waveforms {
      'Y+Z' { 01 {'0ns' D/U;}}
}}}
```

```
MacroDefs {
  M1 {
    W  W1;
    V { Y=#; Z=#; }
}}
```

```
Environment wrapped_core {                          1
  CTL myN {                                         2
    TestMode Normal;                                3
    Internal {
      'Y+Z' { DataType TestMode;                    4
              ActiveState ForceDown; }}
    PatternInformation {
      Pattern P1 {                                  5
        Purpose EstablishMode;}}}

  CTL myE {                                         6
    TestMode ExternalTest;                          7
    Internal {
      Y { DataType TestMode;                        8
          ActiveState ForceDown;}
      Z { DataType TestMode;                        9
          ActiveState ForceUp;}}
    PatternInformation {
      Pattern P2 {                                  10
        Purpose EstablishMode;}}}

  CTL myI {                                         11
    TestMode InternalTest;                          12
    Internal {
      Y { DataType TestMode;                        13
          ActiveState ForceUp;}
      Z { DataType TestMode;                        14
          ActiveState ForceDown;}}
    PatternInformation {
      Pattern P3 {                                  15
        Purpose EstablishMode;}}}
}
```

```
Pattern P1 {                                        16
  Macro M1  { Y=0;  Z=0; }
}
Pattern P2 {                                        17
  Macro M1 {Y=0; Z=1;}
}
Pattern P3 {                                        18
  Macro M1 {Y=1; Z=0;}
}
```

Figure 9: *Partial CTL code to describe the initialization of the three modes of the core.*

The three modes are defined in the **Environment** (1) block of statements to be *myN* (2), *myE* (6) and *myI* (11). Using a **TestMode** statement, these modes are identified to be the configurations for the following.

- **Normal,** functional operation of the core (3).
- **ExternalTest,** the configuration that allows for controllability and observability of the logic outside of the core in the integrated SoC (7).
- **InternalTest,** the mode that allows for testing of the logic inside the wrapper (12).

Each of the modes defined for the wrapped core have different initialization sequences. Three different patterns P1 (16), P2 (17) and P3 (18) are defined to create three different initialization sequences as required for the three modes. In this example the initialization sequences are the simple application of values to the signals Y and Z of the core. A common protocol is defined via the Macro M1 to enable each of the three modes. The macro takes in two parameters (Y, Z), that are provided by the initialization patterns P1, P2, P3 of the three modes (16, 17, 18).

The initialization patterns are linked to the modes in the **PatternInformation** block of statements. The **Normal** mode uses pattern P1 (5), the **ExternalTest** mode uses pattern P2 (10) and the **InternalTest** mode uses pattern P3 (15) to establish their configuration. These patterns are identified to be the initialization patterns through the **Purpose** statement. In this case, the three patterns are identified to be **EstablishMode** patterns.

After the initialization patterns are executed the core is in the desired mode. However, some signals may need to be maintained at a fixed value to ensure that the core stays in the mode. The **DataType TestMode** (4, 8, 9, 13, 14) on the signals in the **Internal** Block is used to highlight the constant signal after initialization is complete. The **ActiveState** defines the driving value on the associated signal for the mode.

5 CTL for Structural Information

A core could contain some design elements that are reused on the SoC. These design elements need to be described in CTL for reuse without giving away actual implementation details that might be construed as Intellectual Property (IP).

Consider the scan chain in the wrapper of the core used in the examples of this paper. Figure 10, highlights the scan elements (cells) built out of flip-flops, and multiplexers. There are two scan cells in this chain with a scan input BSI and a scan output BSO. The chain is clocked by the BCK signal and the BSE signal determines if the scan chain is configured for a scan operation. This scan chain is described in the CTL of Figure 11.

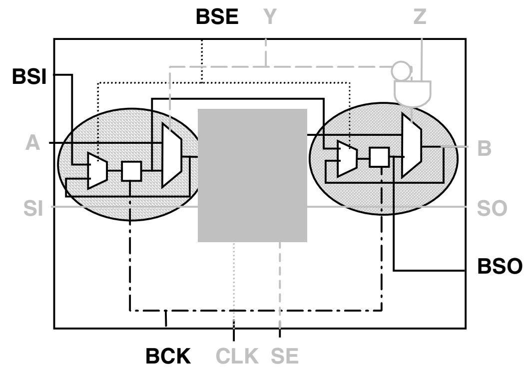


Figure 10: *An example of a scan chain of a core that is to be described in CTL.*

The **ScanStructures** block (36) is used to describe the scan chain and its components. The cells are represented by a symbol (implementation independent) that represents the state in the scan cell (41). In this case two scan cells exist on the scan chain and are named c[0] and c[1]. The values in the scan chain are accessible through the scan input (1, 20, 38) and scan output (1, 25, 39) by using the Macro M2 (33) defined in CTL. M2 is labelled as a **ControlObserve** (31, 32) macro in the **PatternInformation** block (30).

The **Internal** block (3) is used to provide details about every signal (4, 9, 14, 17, 20, 25) as related to the scan and boundary structure. Signal A is connected to scan cell c[0] of chain bc1 and the leading edge of the BCK signal affects the capture of the value (4, 5, 6, 7). Once the value is captured in the scan cell it is observable using Macro M2 as specified by the **TestAccess** statement (8). Similarly, the connections of Signal B is defined (9, 10, 11, 12, 13). Since B is an output, the **LaunchClock** statement is used to define the clock off which the signal is launched (12) and the **TestAccess** statement (13) is used to define the mechanism to control values in the scan cell. In this example, test signals are assigned various data types with associated active values as demanded by the example. BSE is identified as a scan enable signal with a logic-0 needed to allow for the scan operation (14, 15, 16). BCK is identified as a scan clock with an associated state (17, 18, 19). The scan chain input and output signals are identified to be the scan data signals of a boundary scan chain of the design (20, 23, 24, 25, 28, 29).

6 CTL for Test Pattern Information

A core provided as a black box would need to come with test patterns that can be reused to complete the SoC test. CTL provides the test patterns in a restricted STIL syntax, to enable reuse.

```
Signals {
  A In; B Out;
  BSE In;
  BCK In;
  BSI In { ScanIn 2; } BSO Out { ScanOut 2; }        1
}
```

```
Environment wrapped_core {                           2
  CTL {
    Internal {                                       3
      A {                                            4
        IsConnected In {                             5
          StateElement Scan c[0];                    6
          CaptureClock LeadingEdge BCK;              7
          TestAccess Observe Macro M2;}}             8
      B {                                            9
        IsConnected Out {                            10
          StateElement Scan c[1];                    11
          LaunchClock LeadingEdge BCK;               12
          TestAccess Control Macro M2;}}             13
      BSE {                                          14
        DataType ScanEnable;                         15
        ActiveState ForceDown;}                      16
      BCK {                                          17
        DataType ScanMasterClock;                    18
        ActiveState ForceUp;}                        19
      BSI {                                          20
        IsConnected In {                             21
          IsEnabledBy Logic ~a {
            a { Type Signal; Name BSE; }}}           22
        DataType ScanDataIn;                         23
        ScanDataType Boundary;}                      24
      BSO {                                          25
        IsConnected Out {                            26
          IsEnabledBy Logic ~a {
            a { Type Signal; Name BSE; }}}           27
        DataType ScanDataOut;                        28
        ScanDataType Boundary;}}                     29
    PatternInformation {                             30
      Macro M2 {                                     31
        Purpose ControlObserve; }}                   32
}}
```

```
MacroDefs {
  M2 {                                               33
    W W2;
    C { BSE=0; BCK=0;}                               34
    Shift {V { BSI=#; BSO=#; BCK=P;}                 35
}}
```

```
ScanStructures {                                     36
  bc1 {                                              37
    ScanIn BSI;                                      38
    ScanOut BSO;                                     39
    ScanLength 2;                                    40
    ScanCells c[0..1];                               41
    ScanMasterClock BCK;                             42
}}
```

Figure 11: *Partial CTL code to describe information for the wrapper scan chain of the example.*

```
Signals {
  BSE In; SE In; BCK In; CLK In;
  SI In { ScanIn 2; } SO Out { ScanOut 2; }
  BSI In { ScanIn 2; } BSO Out { ScanOut 2; }
}
```

```
SignalGroups {
  Ins='BSE+SE+SI+BSI'{DefaultState ForceOff;}
  Clocks='BCK+CLK'{DefaultState ForceDown;}
  Enables='BSE+SE';
  Outs='SO+BSO';
  A WFC; B WFC;
}
```

```
Timing T1 {
  WaveformTable W2 {
   Period '100ns';
   Waveforms {
     Clocks{0P{'0ns' D/D;'60ns' D/U; '65ns' D/D;}}
     Ins { 01X { '0ns' D/U/X; }}
     Outs { 01X { '0ns' L/H/X; }}
}}
```

```
MacroDefs {
  seq {
    W W2;
    C { Enables=00; Clocks=00; Ins=XXXX;}
    Shift {V { BSI='A' X; SI=#; Clocks=PP;}
    V { Enables=11; Clocks=PP; }
    V { Enables=00; Clocks=00;}
    Shift { V { BSO='B' X; SO=#; Clocks=PP;}}}
  M1 { W W1; Y=#; Z=#; }
}
```

```
Environment wrapped_core {                           1
  CTL myI {                                          2
    TestMode InternalTest;                           3
    PatternInformation {                             4
      PatternExec topPat {                           5
        Purpose Production;                          6
        PatternBurst pats;                           7
      }
      PatternBurst pats {                            8
        Purpose Scan;                                9
        Fault {                                      10
          Type StuckAt;                              11
          FaultCount 100;                            12
          FaultsDetected 96;}}                       13
      Macro seq { Purpose DoTest;}                   14
}}}
```

```
PatternExec topPat {Timing T1;PatternBurst pats;}
PatternBurst pats {
  PatList {
    P3;
    all_pats;
}}
Pattern P3 { Macro M1 {Y=1; Z=0; }}
Pattern all_pats {
  Macro seq { A=1; SI=10; B=0; SO=11;}
  Macro seq { A=0; SI=00; B=1; SO=01;}
  Macro seq { A=1; SI=01; B=0; SO=11;}
  Macro seq { A=1; SI=00; B=1; SO=10;}
}
```

Figure 12: *Partial CTL code to describe test patterns that come with a core, for internal testing of the core.*

STIL is the preferred format for the patterns that come with a core. However, test patterns in STIL come in many flavors and CTL requires that the data and protocol portion of the patterns be separated [12]. Figure 12 shows example patterns for the (wrapped) core that has been used in the previous sections. The patterns are defined in a **PatternExec** topPat. The **PatternInformation** block (4) identifies this top level pattern to be used for **Production** test (5, 6). The actual patterns of the **PatternExec** exist in **PatternBurst**s and are referenced in the PatternInformation block of statements (7, 8**).** The **PatternBurst** in this example contains patterns that are applied through *Scan* (8, 9). The fault coverage of the patterns is described by CTL to be 96% of the **StuckAt** fault model (10, 11, 12, 13).

As mentioned earlier, the sequence information is separated from the Patterns in CTL. As a result of this separation the Macro which represents the sequence is the entity that gets modified to reflect the embedded environment. The macro that needs to change is identified in this example as a **DoTest** macro, implying that the sequence represented by the macro is the application of a single test (14). In this case, the sequence is called non-overlapping, as the scan-in and scan-out operation of consecutive tests do not overlap.

7 Beyond the Example

The example used describes the basic constructs for CTL needed to describe information that comes with a core. The keywords used here are a limited subset of the keywords available in the language.

CTL relies on protocols (Macros) to describe the needs for every configuration of the core. The protocols differ from design to design and can be of any length. This is the fundamental mechanism behind CTL's design independence.

There are numerous other aspects to SoC designs that have not been covered in this document. Most of these aspects deal with different design and test methodologies, such as Logic BIST, or diagnostic issues. Of all the possible subjects one deserves special mention here. CTL is designed to allow for system integration activities to be performed within the realm of CTL itself. The conversion of test patterns that come with a core to the SoC level can be done by changing the Macros associated with the patterns. This discussion is beyond the scope of this paper and the reader can contact the authors for more details on this subject.

8 Conclusions

SoC test itself has been performed in numerous ways in the past. The problems encountered are well known and solutions have existed in the industry that work for a very controlled environment that is mostly available in a single company. CTL [2] is a new language that is created to allow for a general SoC strategy that goes beyond controlled environments.

In this paper, the basic capabilities of the Core Test Language are described. Examples are used to describe

- Design Configuration Information.
- Structural Information.
- Test Pattern Information.

This paper should not be interpreted to be an exhaustive description of CTL. There are numerous syntax items of CTL that were not introduced. The CTL used reflects the latest syntax of the standardization effort upon the time of writing this paper and is subject to changes before it is finalized.

9 Acknowlegements

The authors would like to thank Erik Jan Marinissen, Greg Maston and Ben Bennetts for their constructive criticism of this paper.

10 References

[1] Michael Keating, Pierre Bricaud, *Reuse Methodology Manual for System-on-a-Chip Designs,* Kluwer Academic Publishers, Norwell, Massachusetts.

[2] CTL Web Site, http://grouper.ieee.org/groups/ctl/

[3] IEEE P1500 Web Site, http://grouper.ieee.org/groups/1500/

[4] Erik Jan Marinissen, Yervant Zorian, Rohit Kapur, Tony Taylor, and Lee Whetsel, "Towards a Standard for Embedded Core Test: An Example," in *Proceedings IEEE International Test Conference (ITC)*, 1999, pages 616-627.

[5] VSI Alliance Web Site, http://www.vsi.org/

[6] Erik Jan Marinissen and Maurice Lousberg, "Macro Test: A Liberal Test Approach for Embedded Reusable Cores," in *Digest of Papers of IEEE International Workshop on Testing Embedded Core Based Systems (TECS),* 1997, pages 1.2-1.9.

[7] Erik Jan Marinissen, Rohit Kapur and Yervant Zorian, "On Using IEEE P1500 SECT for Test Plug-n-Play," in *Proceedings of the International Test Conference (ITC)*, 2000, pages 770-777.

[8] Erik Jan Marinissen, Robert Arendsen, Gerard Bos, Hans Dingemanse, Maurice Lousberg, and Clemens Wouters, "A Structured And Scalable Mechanism for Test Access to Embedded Reusable Cores," in *Proceedings IEEE International Test Conference (ITC)*, 1998, pages 284-293.

[9] IEEE Computer Society, *IEEE Standard Test Access Port and Boundary-Scan Architecture – IEEE Standard 1149.1-1990,* IEEE New York, 1990.

[10] IEEE Computer Society, *Supplement to IEEE Std 1149.1-1990, IEEE Standard Test Access Port and Boundary-Scan Architecture*, *IEEE Std 1149.1b-1994*, IEEE New York, March 1995.

[11] IEEE Computer Society, *IEEE Standard Test Interface Language (STIL) for Digital Test Vector Data – Language Manual – IEEE Std. 1450.0-1999*. IEEE New York, September 1999.

[12] Erik Jan Marinissen and Maurice Lousberg, "The Role of Test Protocols in Testing Embedded-Core-Based System IC's," in *Proceedings of the IEEE European Test Workshop (ETW)*, May 1999, pages 70-75.