TRC Expanded Memory Checks
p1450.3 Proposal

Revision 1.0
April 7, 2005

Table of Contents

1. Background - Test Memory Checks and Concerns

One type of Tester Constraint check is a check for memory usage in a Test environment. Memory usage may be a key aspect of a test program to validate, because ATE often has operational constraints with loading, or reloading, test memory. Identifying when test data “fits” into one memory load, or more importantly when test data does not fit, can affect what test data will be applied by the ATE or indicate additional operations in order to apply additional data.

Tester Memory Checks are not easily defined in a generic Test Constraint environment. ATE varies in all parameters of tester memory, by tester architecture and by specific tester configuration. Different ATE architectures will define different segments or types of memory, which are applied in different ways and therefore the rules for how these different blocks are consumed will vary. Even on the same ATE, there may be different configurations of memory or different depths to the same memory for different test situations. For instance, some pins of a device under test may see more memory than other pins. Furthermore, the ultimate application that defines how memory will be allocated will be the external program responsible for either compiling test data for a specific tester, or loading the test data onto the ATE for execution. Codifying all possible configurations of memory and rules of consumption in a generic construct is not possible.

Therefore, this effort will expand the current TRC constructs defined the working drafts to provide sufficiently robust definition for memory consumption, to provide a first-level or second-level approximation to the memory consumption for specific, defined, contexts.

2. Current TRC Memory Check Constructs

2.1 Syntax

The current p1450.3 draft proposals, from Revision 0.7 through Revision 0.9 (latest Working Group draft proposal) defines several statements to support or that affect “memory checks”. These statements may appear in several contexts in a TRC description, as shown in the syntax description that follows.

```
Environment name {
     TRC name {
          PatternCharacteristics name {
               MaxVectors value ;
               InstructionCharacteristics {
                    LoopCharacteristics {            (differ in drafts 0.7,0.9)
                         MaxIteration value ;
                         MinIteration value ;
                         MaxNest value ;
                         VectorModulus value ;       (P1450.3 draft 0.9 only)
                    }
               }
               VectorModulus value ;                 (P1450.3 draft 0.9 only)
          }                          // end PatternCharacteristics
          WaveformCharacteristics name {
               WaveformSelectMemory value ;
```

```
            [many Max* statements imply memory issues as well]
        }                           // end WaveformCharacteristics
        PeriodCharacteristics name {
            PeriodSelectMemory value ;
            [many Max* statements imply memory issues as well]
        }                           // end PeriodCharacteristics

        SignalCharacteristics type {
            MaxVectorMemory value ;
            MaxScanMemory value ;
            MaxCaptureMemory value ;
            PatternCharacteristics name ;
            WaveformCharacteristics name ;
            PeriodCharacteristics name ;
        }                           // end SignalCharacteristics
    }                               // end TRC
}                                   // end Environment
```

2.2 Semantics

There are two basic ‘memory’ values provided in these rules, MaxVectors or MaxVectorMemory, and MaxScanMemory. There are no semantics defined on how STIL files are to be counted to identify a limit to test against these values. Current assumptions interpret each V statement to contribute 1 count against the MaxVectors/MaxVectorMemory, including the V statements contained under nested constructs (Loops, Procedure and Macro calls, Shifts, etc.). The MaxScanMemory can be counted in a myriad of undefined mechanisms (total data across all signals in a Shift, maximum length of data per Shift, etc.) and therefore this value is not checked at this time.

Other memory references in the current proposal (WaveformSelectMemory, PeriodSelectMemory, and MaxCaptureMemory) are not defined as how to be counted. Again, the semantics of what defines consumption of these constructs is not provided in the draft specification.

All MaxVectorMemory values defined under SignalCharacteristics, and all MaxVectors values defined under PatternCharacteristics, are checked against the Vector-count generated for a pattern set in TetraMAX. There is (currently) no notion of scoping or applicability of any defined values, all defined values present in an Environment block parsed as a TRC ruleset are checked against a single derived count of V statements.

2.3 Limitations of the current Memory checks

There are several issues with the current memory checks:

1) There is no definition of what constitutes consumption of these *Memory values.
2) There is no mechanism to identify differentiated consumption for different test environments; a single counted value must be applied against all defined *Memory limits.
3) The environment under which a check should be performed is currently not defined; at this time each defined limit is checked against the same counted value, without a notion of con-

text to apply for each defined *Memory limit.

The second issue is not just an issue across different test environments, but is a restriction on individual test environments as well. For example, some testers support both differentiated memory consumption based on signal type, and multiple types of memory to hold scan data or sequence operations, each of which have different rules for consumption.

The proposed constructs are structured to address these specific issues, with the hope that this will extend to other test environments as well.

3. Proposed Constructs

The first issue identified in section 2.3 on page 2 can be solved with sufficient definition. The second issue requires additional constructs to support differentiation of the counts, and for test environments that support differentiation of the memory based on different types or attributes on signals, the third issue needs to be addressed.

One requirement of this operation is to support different memory values for different pin types. The existing constructs of defining MaxVectorMemory, scoped under SignalCharacteristics, supports a mechanism for direct definition of multiple maximum values in different SignalCharacteristics blocks.

The MaxScanMemory value is not seen to be useful in this context, as the values counted will be Vector-oriented for all pin types.

The SignalCharacteristics MaxVectorMemory supports a single MaxVectorMemory definition per SignalCharacteristics block. Since named PatternCharacteristics blocks can be referenced from the SignalCharacteristics block, and each PatternCharacteristics block may contain a MaxVectors value, the proposal here is to extend the PatternCharacteristics reference in SignalCharacteristics, to support multiple references to define potentially multiple memory contexts and counting environments.

Relating pin types to SignalCharacteristics blocks will be defined below.

3.1 Syntax Extensions

To support pin-specific maximum vector counts, the “MaxVectors” count needs to support some sort of differentiation based on types or contexts that Vectors are found under. This is provided by defining a series of additional statements that have the generic form “MaxVectorsCount<type>” followed by a list of 1 or more attributes. The <types> are: `Vectors`, `MacroCalls`, `ProcedureCalls`, `Shifts`, and `Loops`. The attributes are: an integer value, `inline` which means to count the constructs contained in this block as if expanded as the equivalent set of Vectors generated at runtime, and `once` which counts the contents as if unfolded once, although the semantics of `once` and `inline` are different between Macro/Procedures and Shift/Loop contexts. The following table highlights the attributes that are pertinent to each type of statement, and the respective effects of

each attribute for that construct.

Table 1: MaxVectorsCount Statements

statement	attributes	effect - increment the count by:
`MaxVectorsCountVectors`	integer	this integer value for each V statement
`MaxVectorsCountMacroCalls`	integer	this integer value for each Macro call
	inline	the effects of the contents of this Macro, on each call to each Macro. The contents of the Macro are counted as if the Macro is expanded each time it is called.
	once	the effects of the contents of this Macro, **once** for each unique Macro name (does not count repeat calls to previously-called Macros) The contents of the macro are counted as if they contribute **once** to the memory space.
`MaxVectorsCountProcedure-Calls`	integer	this integer value for each Procedure call
	inline	the effects of the contents of this Procedure, on each call to each Procedure. The contents of the Procedure are counted as if the Procedure is expanded each time it is called.
	once	the effects of the contents of this Procedure, **once** for each unique Procedure name.
`MaxVectorsCountShifts`	integer	this integer value for each Shift block
	once	1 iteration of the effects of the contents of the Shift block. The contents of the Shift block are counted once and are not dependent on the Shift data.
	inline	the net count of contents of the Shift block after iterating/unfolding the Shift with the data presented to the Shift. The contents of the Shift block are expanded as if the Shift executed linearly through the Shift data.
`MaxVectorsCountLoops`	integer	this integer value for each Loop block
	once	1 iteration of the effects of the contents of the Loop block. The number of loop iterations does not contribute to this count, only the effects of one pass through the statements inside the Loop.
	inline	the net count of contents of the Loop block after iterating/unfolding the Loop by the count value (flatten/expand the loop)

There may be more than one attribute present for each statement, in particular an integer value may be present in combination with `once` and `inline` attributes, to indicate an incremental amount of memory consumed in addition to the effects of the contents of the construct.

The italicized text below identifies the proposed additional statements:

```
Environment name {
    TRC name {
        PatternCharacteristics name {
            MaxVectors integer_value ;
            VectorModulus integer_value ;
            MaxVectorsCountVectors MaxVectorsCount_attributes+ ;
            MaxVectorsCountMacroCalls MaxVectorsCount_attributes+ ;
            MaxVectorsCountProcedureCalls MaxVectorsCount_attributes+ ;
            MaxVectorsCountShifts MaxVectorsCount_attributes+ ;
            MaxVectorsCountLoops MaxVectorsCount_attributes+ ;
            InstructionCharacteristics {
                LoopCharacteristics {
                    VectorModulus integer_value ;
                }
            }
        }                           // end PatternCharacteristics name
    }                               // end TRC name
}                                   // end Environment name
```

where *`MaxVectorsCount_attributes`* is from the set: `integer_value, once, inline.`

A typical application environment will define multiple PatternCharacteristics blocks. These PatternCharacteristics blocks must be referenced from SignalCharacteristics blocks, where each SignalCharacteristics block contains a set of signal types. These signal types must be present in the STIL test before the constraints of this PatternCharacteristics block will be considered.

Table 2, “SignalCharacteristics Types,” on page 5 identifies the types of signals currently defined for SignalCharacteristics blocks, and the semantics to identify when a signal of this type exists in a STIL test.

Table 2: SignalCharacteristics Types

type	intent
Data	any Signal with more than one WFC defined in any WFT contained in the STIL test and **not** also identified as Clock. If “Data In”, then only signals that have more than 1 WFC each containing drive events. If “Data Out”, then only signals that have more than 1 WFC each containing compare events.
Clock	any Signal with more than 1 drive event other than “Z” in any single WFC in any WFT contained in the STIL test. --spec issue: define Clock In and Clock Out.
Scan	any Signal that has a ScanIn or ScanOut attribute, defined on that Signal or from any SignalGroup that contains that Signal, in the STIL test.

Table 2: SignalCharacteristics Types

type	intent
In	Signal of type In -- spec issue: unless “Data In” or “Clock In”, in which case the respective first term overrides as the type classification?
Out	Signal of type Out -- spec issue: unless “Data Out” or “Clock Out”, in which case the respective first term overrides as the type classification?
InOut	Signal of type InOut
SplitIO	--spec issue: indicates the signal consumes two tester “tracks” to support independent drive and compare operations, however I don’t know what attributes of STIL information are necessary to define a STIL Signal needs this.
Asynchronous	ignored, at least by me.
-none-	always applied to a STIL test

For instance:

```
Environment all_tricks { TRC trick1 {
    PatternCharacteristics “single-memory” {...}
    PatternCharacteristics “bidi-memory” {...}
    SignalCharacteristics In Out { PatternCharacteristics “single-memory”; }
    SignalCharacteristics InOut { PatternCharacteristics “bidi-memory”; }}}
```

In this example, the checks defined for “bidi-memory” will not be applied to a STIL test unless that test contains Signals of type InOut, and likewise the checks for “single-memory” will only be applied if the STIL test contains at least one Signal of type In or Out.

3.2 Limitations to this Approach

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

3.3 Application Example

The following example demonstrates one application of the proposed constructs. The PatternCharacteristics block is used to define the MaxVectors available for this “type” of Pattern, although more accurately it represents the attributes of a specific type of memory.

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

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

```
Environment TRC {
    TRC “_example_” {
        PatternCharacteristics “sequencer” {
            MaxVectors 1024 ;
            MaxVectorsCountVectors 0 ;
                // All macro and procedure bodies are expanded
                // into the Parallel Vector space for each call
            MaxVectorsCountMacroCalls 0 ;
            MaxVectorsCountProcedureCalls 0 ;
                // Shift and Loop operations generate a seq-jump and
                // iteration operation, and a seq-”return” to the main flow
            MaxVectorsCountShifts 2 ;
            MaxVectorsCountLoops 2 ;
        }                           // end PatternCharacteristics “sequencer”
        PatternCharacteristics “inlinemem” {
            MaxVectors 1G ;
            VectorModulus (7 * 4) ;
            MaxVectorsCountVectors 1 ;
                // All macro and procedures are expanded
                // into the Parallel Vector space for each call
            MaxVectorsCountMacroCalls inline ;
            MaxVectorsCountProcedureCalls inline ;
                // Loops and Shifts are defined once;
                // iteration is controlled by sequencer operations.
            MaxVectorsCountShifts once ;
            MaxVectorsCountLoops once ;
            InstructionCharacteristics {
                LoopCharacteristics {
                    VectorModulus 7 ;
                }
            }
        }                           // end PatternCharacteristics “inlinemem”
        PatternCharacteristics “scanmem” {
            MaxVectors 10G ;
            VectorModulus (7 * 4) ;
            MaxVectorsCountVectors 1 ;
            MaxVectorsCountMacroCalls inline ;
            MaxVectorsCountProcedureCalls inline ;
                // Shift data consumes the number of iterations
                // defined by the equivalent Vectors generated
```

```
                MaxVectorsCountShifts inline ;
                    // Loops are defined once;
                    // iteration is controlled by sequencer operations.
                MaxVectorsCountLoops once ;
                InstructionCharacteristics {
                    LoopCharacteristics {
                        VectorModulus 7 ;
                    }
                }
            }                           // end PatternCharacteristics
            SignalCharacteristics Data {
                PatternCharacteristics “inlinemem” ;
            }                           // end SignalCharacteristics Data
            SignalCharacteristics Scan Clock {
                PatternCharacteristics “scanmem” ;
            }                           // end SignalCharacteristics Scan & Clock
            SignalCharacteristics {
                PatternCharacteristics “sequencer” ;
            }                           // end SignalCharacteristics generic

        }                               // end TRC
    }                                   // end Environment
```