

UUT Test Requirements Capture in any language you like... and still compliant with the IEEE Signal Definition and Test Description Standard

Mr. Keith Ellis, Apsys Ltd, (44) 1 438 821555, keith.ellis@apsys.co.uk
Mr. Chris Gorringe, Racal Instruments Ltd, (44) 1202 872800, chris@racalinst.co.uk
Mr. Jim Langlois, RAF Wyton

ABSTRACT

This paper examines how to create consistent, portable, usable UUT Requirement Specifications utilizing COTS computer languages such as Basic, Java, C or C++. It goes on to describe the benefits of using standard, common building blocks that represent Signals, and identifies the importance of allowing users to extend their own Signal components. It concludes by showing several examples of the same, real requirement captured in different languages, and how these can be ported from one to another...

Keywords: ATP, test requirements, signal descriptions.

1 INTRODUCTION

It has long been a goal to minimize the effort to describe test requirements and directly produce test programs. Ideally, we would like any test requirements to be directly usable within a test program, or be usable with the minimum of additional effort.

The new IEEE Signal Definition and Test Description (SDTD) standard, more commonly known as A2k, provides a rich set of features for defining and controlling signals, which can be directly utilized by test requirements and test programs alike. As well as providing building blocks for signal modeling, the SDTD standard also provides a Test Procedure Language (TPL) layer to allow UUT test requirements to be captured in a succinct and stylized English manner. This language layer does not prevent users from capturing UUT test requirements in any programmable language. In fact, a goal of the standard is to allow consistent UUT test requirements to be captured across different computer languages including C/C++, Basic, the new emerging .NET language platforms, or even data description languages such as XML. However, for those users needing to capture requirements outside specific computer languages, the TPL provides a pseudo language definition, offering a stylized English method to capture and port test requirements across test programming environments.

The success of this approach is centered on providing the semantics of signal definition, so that different language descriptions of signals still contain the same meaning. Being able to define the

same UUT test requirement in multiple languages with the same meaning implies that we are not forced to port programs from one machine source to another. Instead we can adopt a binary transfer where the new platform need only run the binary program, for example through adding a C compiler or basic interpreter to the new platform.

As well as defining how the same requirement can be represented in different test programming environments, the flexibility of the SDTD standard, allows traditional test program elements to be converted into standard signal definitions, where these signal definitions allow us to capture test requirements of the original test program. In order to achieve such a reverse engineering process we need to map each instrument or test program command that affects the UUT into signal components. In the past this has been near impossible because of the fixed structure of the available standard signal definitions. The SDTD standard does not have this fixed signal structure, but provides building blocks through which users can define their own signals, and signal behavior, whilst still being able to predict their behavior. The notion of test programs defining their own set of signals allows test programs written in ATE test languages to be ported onto different signal ATE platforms.

For new programs the SDTD standard defines a set of signals equivalent to the ATLAS716-95 nouns. This allows programmers familiar with ATLAS to use their existing skill set. The ability to use existing signal definitions or build new ones from standard building blocks or library elements, whilst having a formal definition of the signal, allows signal portability to be achieved across platforms and is the major contributor to test program portability.

2 TEST REQUIREMENT CAPTURE

Despite several shortcomings, attributable to its bespoke nature and poor system implementation, ATLAS has for years been the only viable language to capture test requirements. These shortcomings have led users to adopt COTS programming languages as test programming languages, diluting the information contained in a true test requirement and thus losing one of the original objectives, namely portability.

The new approach encompassed within the SDTD standard is to allow COTS languages to be used as the carrier language for a signal definition environment by extracting the signal definitions from a specific ATLAS language and making them available to all¹ COTS languages. The SDTD standard achieves this by providing a description of the signal building block templates, called **SignalFunctions**, together with their controllable properties, in a language neutral form, which can be mapped into COTS language environments. This language neutral form is called an IDL (Interface Definition Language) file, and allows computer language environments such as C/C++ Basic to generate their own header and input files, e.g. .H,.INC,.PAS, that allows access to these signal building blocks. Some language development environments (IDEs) are powerful enough that they can use the IDL description directly and extend their programming features, missing out the need for additional interfacing header files.

¹ There are certain criteria a COTS language must meet to support this.

The SDTD standard allows the use of any of these COTS languages, to define common signal descriptions for UUT testing, where these signal descriptions become the UUT Test Requirements.

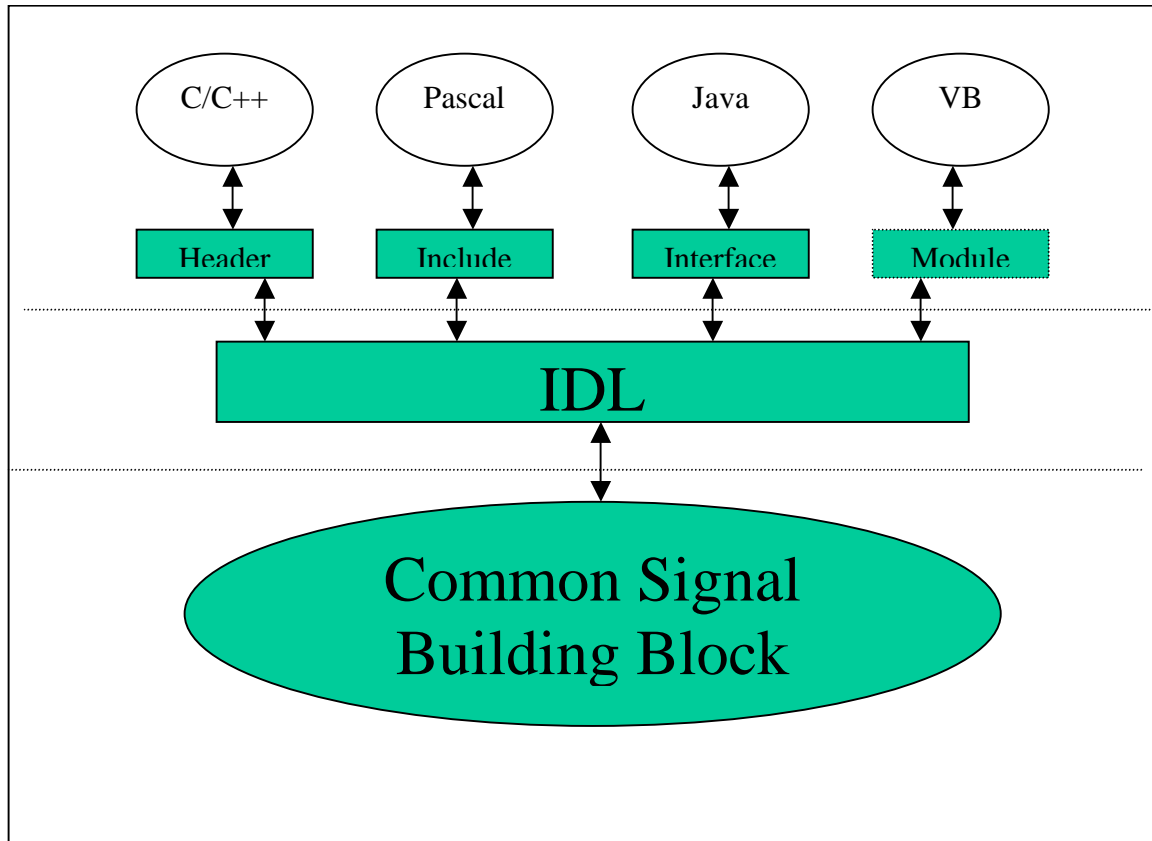


Figure 1. Multiple Languages use Common Signals

The SDTD standard provides for both signal control and signal definition. Signal definition is achieved in two ways by use of the signal building block templates:

- Using existing library signal components, from a TSF (Test Signal Framework) library
- Building your own signal components to describe custom or unique signals

2.1 Using existing TSF components

This is the simplest way of defining test requirements and is used when somebody else has defined the signal template already. In this case to define a signal we create an object of the **SignalFunction** class and just fill in the parameters.

If we take a simple example, of wanting an input to the UUT being a 1 mV, 1 kHz, AC_SIGNAL. Within our carrier language we require to create an AC_SIGNAL (object), instantiated with parameters (properties) `ac_ampl="1 V"` `freq="1 kHz"`, and that we want our signal to go to some edge connection on the UUT.

This can be shown in a block diagram design:

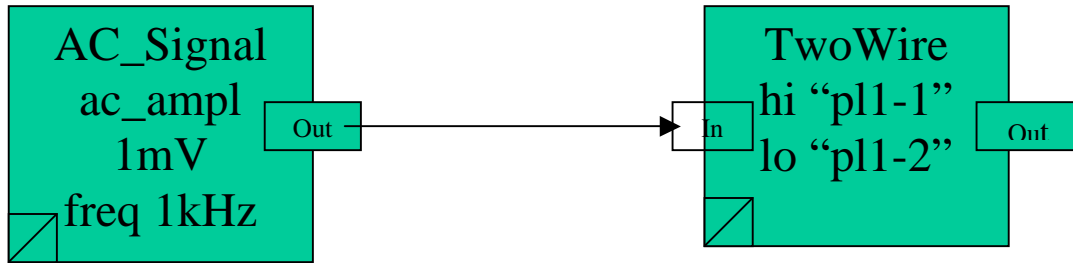


Figure 2. Simple Block Diagram Design

2.1.1 Pseudo Test Procedure Language layer.

```
apply AC_SIGNAL ac_ampl 1mV cnx hi "pl1-1" lo "pl1-2"
```

Figure 3. Pseudo TPL Example

Note we don't need to mention frequency because the AC_Signal TSF template defines the default value for frequency to be 1 kHz.

2.1.2 VB Program (using IDL).

We can also express our test requirement directly in VB using the SDTD standard IDL definitions.

```
'define UUT connection
Set acSig = A2k.Require("TwoWire") 'UUT Connection
  acSig.hi="pl1-1"
  acSig.lo="pl1-2"
'define AC_SIGNAL as its input
Set acSig.In = A2k.Require("AC_SIGNAL") 'create an AC as input
  acSig.In.ac_ampl = "1mV"
'Turn the signal On
acSig.Out.Run
```

Figure 4. VB Program Example

In the above example:

Require is a method of the resource (A2k), which returns a **SignalFunction** object of the type specified i.e. "TwoWire" or "AC_SIGNAL".

In and **Out** represents the signal input and the signal output to/from a **SignalFunction**.

Run is a signal control method that turns on the signal. **Run** has the meaning of **apply** because **apply** is both signal setup and connection switching. A signal can be explicitly turned off, using the **Stop** signal control method.

2.1.3 XML

We can even use the definitions in the SDTD standard and define our signal in an XML format.

```
<Signal>
  <TwoWire>
    <hi>p11-1</hi>
    <lo>p11-2</lo>
    <In>
      <AC_SIGNAL>
        <ac_ampl>1mV</ac_ampl>
      </AC_SIGNAL>
    </In>
  </TwoWire>
</Signal>
```

Figure 5. XML Signal Definition

2.2 Building custom signals

At some point a TSF or Library component that matches the required signal will not be available. If a noisy **AC_SIGNAL** was required, because there is a need to measure the UUT tolerance to distorted signals, the **AC_SIGNAL** description does not have a notion of noise. To overcome this problem we need only define our definition of "what a noisy signal is?" and then use it within the test requirement.

For example we might want 'noisy' to be random or white noise, rather than harmonic distortion or non-harmonic distortion, such as a 50 Hz hum.

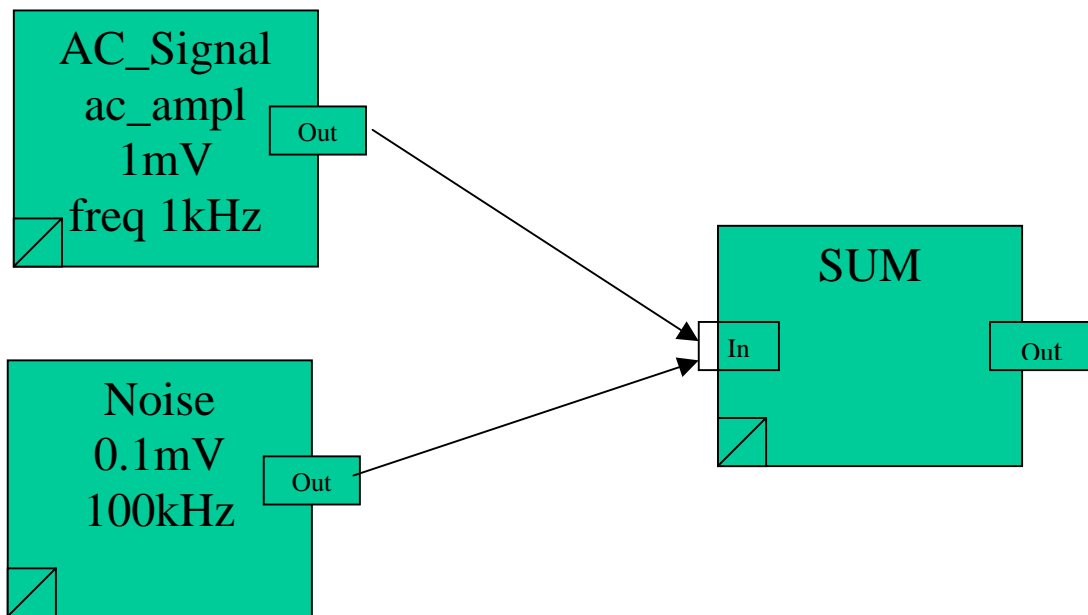


Figure 6. Noisy AC Signal Block Diagram Design

Having defined such a signal we can either use them directly in our test requirement, or package them up into our own TSF Library components for reuse by others. There is never any ambiguity of what the signal definition is, because the SDTD mathematical definitions provide a formal definition of the new signal.

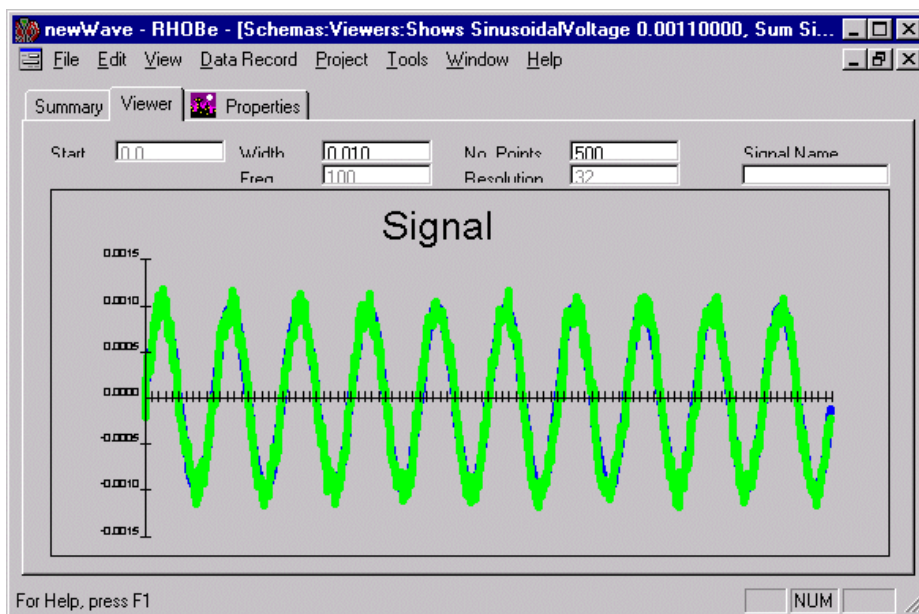


Figure 7. Noisy AC Signal mathematical definition

2.2.1 Pseudo Test Procedure layer.

```
apply AC_SIGNAL ac_ampl 1mV with Noise amp 0.1mV freq 100kHz
cnx hi "pl1-1" lo "pl1-2"
```

Figure 8. Noisy AC Signal TPL definition

2.2.2 VB Program (using IDL).

```
'define UUT connection
Set acSig = A2k.Require("TwoWire") 'UUT Connection
acSig.hi="pl1-1"
acSig.lo="pl1-2"
'define Noisy AC_SIGNAL as sum of two components
Set noisyAcSig = A2k.Require("SUM")
Set noisyAcSig.In(1) = A2k.Require("AC_SIGNAL")
noisyAcSig.In(1).ac_ampl = "1mV"
Set noisyAcSig.In(2) = A2k.Require("Noise")
noisyAcSig.In(2).amp = "0.1mV"
noisyAcSig.In(2).freq = "100kHz"
'and connect to UUT input
Set acSig.In = noisyAcSig.Out
'Turn the signal On
acSig.Out.Run
```

Figure 9. Noisy AC Signal VB definition

This example shows the parameter form of **In** and **Out**, e.g. **In(1)**. The parameter value is used only as a unique identifier, to distinguish one input or output from another. In the above example **In(1)** or **In("AcSignal")** would have been equally acceptable.

2.2.3 XML

We can also use XML to define our new Signal

```
<Signal>
  <TwoWire>
    <hi>pl1-1</hi>
    <lo>pl1-2</lo>
    <In>
      <Sum>
        <In>
          <AC_SIGNAL>
            <ac_ampl>1mV</ac_ampl>
          </AC_SIGNAL>
          <Noise>
            <amp>0.1mV</amp>
            <freq>100kHz</freq>
          </Noise>
        </In>
      </Sum>
    </In>
  </TwoWire>
</Signal>
```

Figure10. XML Noisy Signal Definition

3 STANDARDIZED APPROACH

The TPL provides verbs giving users the ability to manipulate test objects that is signal definitions contained within the TSFs, in to actual test procedures or signal sequences.

Users can define test requirements in terms of these test objects embedded upon COTS programming languages. In this way different languages can use their own syntax and language semantics and still define the same test requirement because they both reference the same external test objects.

The Test Procedure Language [TPL] Layer provides a mechanism for those users who want to document test requirements in a textual format. The use of the TPL to write test requirements is analogous to using ATLAS inasmuch as the TPL uses stylized English signal statements to describe tests and to manipulate signals. It differs from using ATLAS in that it does not provide a fully defined programming language. Instead A2K allows users to adopt their own preferred programming language in which the signal statements and the underlying semantics of tests can be written.

The goals of the TPL are:

1. its keywords have meanings that are normally accepted by the world-wide testing community
2. it is an effective means for communicating test information relating to the testing of Unit(s) Under Test[UUT] between an originator of an UUT Test Requirement[TR] and an implementer of a TR
3. test requirements written according to the TPL rules shall be portable to implementations on different designs of test equipment that have the same testing capability no matter how it is controlled.

Any TPL description comprises of the two elements:

1. signal statements that are used to configure, manipulate, control and measure signals
2. carrier language that is a programming language in which the signals statements can be written, sequenced, observed and generally supported.

To produce test requirements using the TPL, users must embed the test in their preferred carrier language.

Within a TPL environment there will be some translation mechanism to convert from this neutral format of the signal statements into their preferred carrier language format before the test statements can be compiled and executed. The examples provide a background of how the TPL maps into a specific carrier language.

Use of a translator to convert the neutral representation of the test statements into the carrier language format offers certain benefits in that parameter type checking and semantics checks can be conducted prior to test execution.

4 EXAMPLES

The following examples show different test requirements expressed in TPL, VB - a COTS programming environment, and an IEEE Std 716 –1995 C/ATLAS. The examples serve to illustrate how the SDTD standard can be implemented to define UUT test requirements using a variety of platforms and languages.

In the examples the following conventions are used to aid understanding:

Items **highlighted** are optional

Items **Bold** are A2k Basic or TSF components

Items *Italics* are Native Basic Language Words, the carrier language.

Items **gray** are comments

4.1 DC Amplifier

In this example the UUT test requirements are defined as apply power 28Vdc across PL1-1/2, 20Vdc across PL1-3/4, Supply 10Vdc across J1-1/2, and measure voltage across J2-1/2, calculate the voltage gain between J1-1 and J2-1.

The following diagram represents a block diagram of the requirement.

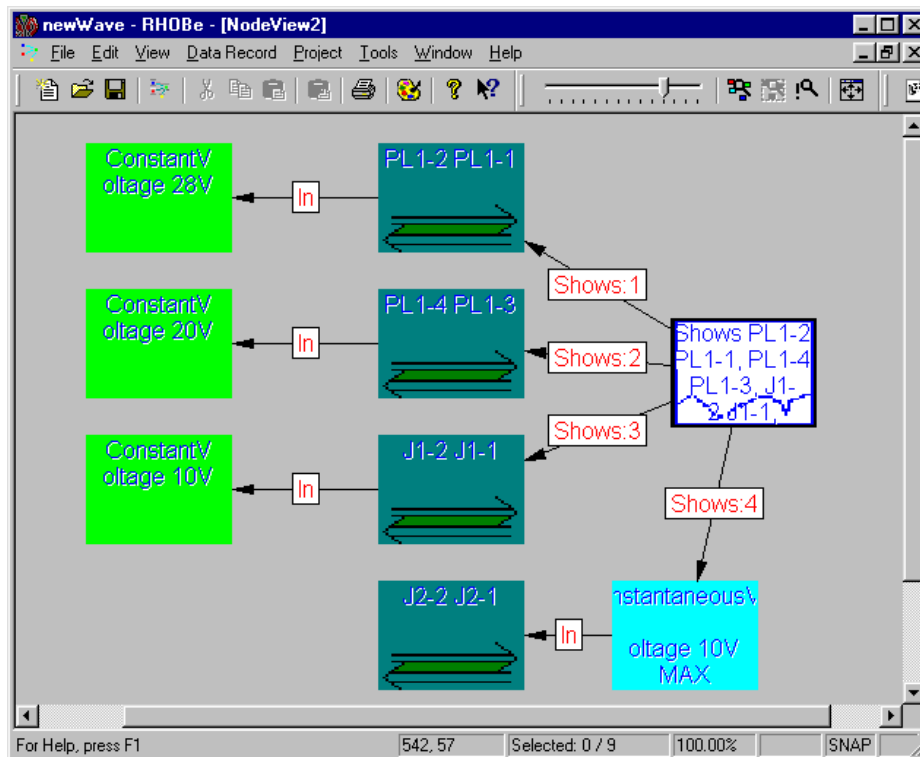


Figure 11. DC Amplifier Block Diagram

4.1.1 TPL

This example shows how to express the DC Amplifier requirement using a pseudo carrier language. The **highlighted** items would be expressed in a user selected carrier language. The test requirements show how to mix signal definitions with procedural and output statements using the carrier language.

```
*****
*   Test Methodology       :   TPL           *
*   UUT Type              :   DC Amplifier   *
*   Model                 :   DC-01         *
*   Serial number        :   DcAmp 001     *
*****
//
*****
* Begin the preamble part *
*****
//
test PowerUp(single Vin-1, Vin-2) {
    apply DC_SIGNAL dc_ampl Vin-1 range 16:32V CNX HI PL1-1, LO PL1-2 As DC1;
    apply DC_SIGNAL dc_ampl Vin-2 range 16:32V CNX HI PL1-3, LO PL1-4 As DC2;
    return();
}
//
test GainTest(single Vin) {
    single Vout;
    apply DC_SIGNAL dc_ampl Vin Range 0 : 20V CNX HI J1-1, LO J1-2 As DCin;
    measure InstantaneousVoltage into Vout Nominal 1000V Max CNX HI J2-1, LO J2-2 As SigName;
    return(Vout/Vin);
    remove DCin, Signame;
}
//
*****
* Begin the procedural part *
*****
//
*****
*   Start testing the UUT *
*****
//
output("Start of TPL Program 'PRO_EXM' ");
remove all;
powerUp(28.0V, 24.0V);
gain=GainTest(10.0V);
output("Gain is",gain);
remove all;
output("End of TPL Program 'PRO_EXM' ");
//
*****
*   End of procedural part *
*****
```

Figure12. DC Amplifier TPL Test Requirement

4.1.2 Embedded Test Procedure Layer in Native Language (VB)

This example shows an optimized DC Amplifier requirement using VB as the carrier language and illustrates the redundancy of the "Remove" or "Disconnect" statements. Because of the behavior defined in the SDTD standard the TPL Verbs create **SignalFunctions** and control them through their Signal control interface. Within the TPL, signals are turned on with an internal statement like '<object>.Out.Run'. To remove or disconnect the signal we might expect to need an equivalent '<Object>.Out.Stop', however these operations are implied when the **SignalFunction** object is deleted. In addition all **SignalFunction** objects are deleted when they are no longer referenced or used. The effect is that as a program exits, it tidies up all variables and references to **SignalFunctions**, this in turn informs the **SignalFunction** (implicitly) that it is no longer used which then resets any signals, thus disconnecting or removing the signal or connection.

Note that the TPL statements are embedded using a carrier language comment, in this case the character (').

In the following example the **highlighted** items are optional.

```
' /*****\
' *      Test Methodology      :      ATLAS2K      *
' *      UUT Type              :      DC Amplifier  *
' *      Model                 :      DC-01         *
' *      Serial number        :      DcAmp 001      *
' \*****/
' //
' /*****\
' * Begin the preamble part that contains the declarative statements *
' \*****/
' //
Subroutine PowerUp Vin-1, Vin-2
    'apply DC_SIGNAL ac_amp1 Vin-1 Range 16V:32V CNX HI PL1-1 LO PL1-2
    'apply DC_SIGNAL ac_amp1 Vin-2 Range 16V:32V CNX HI PL1-3 LO PL1-4
End Subroutine
' //
Function GainTest(Vin) as Double
    'apply DC_SIGNAL ac_amp1 Vin Range 0V:20V CNX HI J1-1, LO J1-2]
    'measure InstantaneousVoltage into Vout Nominal 1000V Max CNX HI J2-1, LO J2-2]
    GainTest = Vout/Vin
End Function
' //
' /*****\
' *      End the preamble part      *
' \*****/
' //
' /*****\
' * Begin the procedural part that contains all the UUT test statements *
' * that call the declarative statements in the preamble part *
' \*****/
' //
' /*****\
' *      Start testing the UUT      *
' \*****/
' //
'/ Inform user:      Begin testing /
Output " Begin ATLAS2K Program PRO_EXM "

'/ Power up the UUT /
PowerUp Vin1:=28.0, Vin2:=24.0
```

```

'/ Start the amplifier test /
Vout = GainTest(Vin:=10.0);
'/ Output the result information to the test operator /
Output " Gain is val/Vin " & Vout
'/Good test practice to remove all connections and settings of the test instrumentation/
'/This is done automatically at the end of a program/
'/ Inform user:          End of UUT testing /
Output " End ATLAS2K Program 'PRO_EXM' "
'//
'/******\
' *      End of procedural part          *
' \*****/

```

Figure 13. Optimized DC Amplifier TPL/VB Test Requirement

4.1.3 VB Native Language and A2k Basic Components

This example builds a complete VB program that defines the DC Amplifier Test Requirement. The program defines a signal model and then controls the model using the Run methods. The style of the program is not optimal but follows the same layout of previous examples to allow direct comparison and mapping of TPL components. Additional non-signal statements such as user interaction are also included.

The code is written without the support of the TPL or any other runtime environment. It is a raw VB Test Specification portable to any platform that supports the language features identified in *italics*. It makes use of a **Resource** object called A2k. The reason for the need of a **Resource** object is basically any development environment that uses SDTD cannot restrict itself to the one source that can create building block objects. It would be like locking yourself into a single implementer and then never being able to change. Therefore, by having all building blocks created through some resource (factory) object allows the user to have multiple SDTD implementations on the same machine.

The **highlighted** items show optional items.

```

'/******\
' *      Test Methodology      :      ATLAS2K          *
' *      UUT Type              :      DC Amplifier     *
' *      Model                 :      DC-01           *
' *      Serial number         :      DcAmp 001        *
' \*****/
'//
'/******\
' * Begin the preamble part that contains the declarative statements *
' * Define the Signal Graph here                                     *
' \*****/
'//
Dim A2k
Set A2k = CreateObject("Atlas2k.Resource")
'define DC_SIGNAL, Voltage Vin-1 Range 16V-32V CNX HI PL1-1 LO PL1-2
Set dc1 = A2k.Require("DC_SIGNAL")
    dc1.Voltage = "28V Range 16V:32V"
Set dc1cnx1 = A2k.Require("TwoWire")
    Dc1cnx1.HI = "PL1-1"
    Dc1cnx1.LO = "PL1-2"
Dc1cnx1.In = dc1.Out

'define DC_SIGNAL, Voltage Vin-2 Range 16V-32V CNX HI PL1-3 LO PL1-4
Set dc2 = A2k.Require("DC_SIGNAL")
    Dc2.Voltage = "24V Range 16V:32V"
Set dc1cnx2 = A2k.Require("TwoWire")

```

```

    Dc1cnx2.HI = "PL1-3"
    Dc1cnx2.LO = "PL1-4"
Dc1cnx2.In = dc2.Out

'define DC_SIGNAL Voltage Vin Range 0V-20V CNX HI J1-1, LO J1-2
Set dc3 = A2k.Require("DC_SIGNAL")
    Dc3.Voltage.Range = "Range 0:20V"
Set dc1cnx3 = A2k.Require("TwoWire")
    Dc1cnx3.HI = "J1-1"
    Dc1cnx3.LO = "J1-2"
Dc1cnx3.In = dc3.Out

'define (InstantaneousVoltage) Nominal 100V MAX CNX HI J2-1, LO J2-2
Set mivcnx = A2k.Require("TwoWire")
    Mivcnx.HI = "J2-1"
    Mivcnx.LO = "J2-2"
Set miv = A2k.Require("InstantaneousVoltage")
    Miv.Nominal = "10V MAX"
Min.In = mivcnx.Out

'//
'/******\
' *      Signal Graph Complete          *
' *      End the preamble part          *
'\*****/
'//
'/******\
' *      Begin the procedural part that contains all the UUT test statements
' *      that call the declarative statements in the preamble part          *
'\*****/
'//
'/******\
' *      Start testing the UUT          *
'\*****/
'//
'/ Inform user:          Begin testing /
Output " Begin ATLAS2K Program PRO_EXM "

'/Apply Power /
dc1.Out.Run 'Apply dc1
dc2.Out.Run 'Apply dc2

'/Measure the gain into Vout
Vin = 10.0
Dc3.Voltage = Vin
dc3.Out.Run 'Apply dc3
miv.Out.Run 'Take measurement
Vout = miv.Measurement/Vin
'/ Output the result information to the test operator /
Output " Gain is val/Vin " & Vout

'/We don't need to tidy the signals because they will be turned off/
'/When their SignalFunction objects are no longer needed they get deleted /
'/As the program exits but just in case your interested/
dc3.Out.Stop 'Remove dc3
dc2.Out.Stop 'Remove dc2
dc1.Out.Stop 'Remove dc1

'/ Inform user:          End of UUT testing /
Output " End ATLAS2K Program 'PRO_EXM' "
'//
'/******\
' *      End of procedural part          *
'\*****/

```

Figure 14. DC Amplifier TPL Test Requirement

4.1.4 ATLAS 716-95

The following traditional ATLAS 716–95 example is provided as a common reference.

```
C*****\
*   Test Language       :   ATLAS95       *
*   UUT Type:          :   DC Amplifier   *
*   Model:             :   DC-01         *
*   Serial number:     :   DcAmp 001     *
\*****$
CS
C*****\
* Begin the preamble part that contains the declarative statements *
\*****$
CS
000010 BEGIN, ATLAS PROGRAM 'PRO_EXM' $
CS
C 000100 DECLARE, VARIABLE, 'STORE' IS DECIMAL $
CS
005000 DEFINE, 'POWER-UP', PROCEDURE ('VIN-1', 'VIN-2' IS DECIMAL) $
005005 APPLY, DC SIGNAL,
          VOLTAGE 'VIN-1' RANGE 16 V TO 32 V,
          CNX HI PL1-1 LO PL1-2 $
005010 APPLY, DC SIGNAL,
          VOLTAGE 'VIN-2' RANGE 16 V TO 32 V,
          CNX HI PL1-3 LO PL1-4 $
005015 END, 'POWER-UP' $
CS
010000 DEFINE, 'GAIN TEST', PROCEDURE ('VIN' IS DECIMAL)
          RESULT ('GAIN' IS DECIMAL) $
010010 DECLARE, VARIABLE, 'VOUT' IS DECIMAL $
010020 APPLY, DC SIGNAL,
          VOLTAGE 'VIN' RANGE 0 V TO 20 V,
          CNX HI J1-1 LO J1-2 $
010030 MEASURE, (VOLTAGE INTO 'VOUT'), DC SIGNAL,
          VOLTAGE MAX 1000 V,
          CNX HI J2-1 LO J2-2 $
010040 CALCULATE, 'GAIN' = 'VOUT' / 'VIN' $
010050 END, 'GAIN TEST' $
CS
C*****\
* Begin the procedural part that contains the UUT test statements *
* that call the declarative statements in the preamble part *
\*****$
CS
C*****\
*   Start testing the UUT *
\*****$
CS
E100000 REMOVE, ALL $
100005 PERFORM, 'POWER-UP' (28, 24) $
100010 PERFORM, 'GAIN TEST' (10) RESULT ('STORE') $
```

```

100020 OUTPUT, C' GAIN IS 'GAIN' '$
999980 REMOVE, ALL $
999990 FINISH $
99999 TERMINATE, ATLAS PROGRAM 'PRO_EXM' $
C$
C*****\
*      End of procedural part      *
\*****$

```

Figure 15. DC Amplifier TPL Test Requirement

4.2 AC Amplifier

The following example is a UUT test requirements for a simplified AC Amplifier. The original specification had been written using a *For/Next* Loop. This defines a dynamic test requirement. The term dynamic refers to the signal definitions are being dynamically changed over time by the test program, the speed or timing of each loop is undefined and therefore the requirement is ambiguous.

An alternative way to define this test requirement would have been through a Frequency Sweep description. This represents a static test requirement. The term static is used because even though the signal changes over time the signal model remains constant. In static test requirements there is no timing ambiguity, this is because both the frequency value and the time are defined.

Historically test requirements have been produced using a dynamic description, and as a consequence there are not many static test requirements available. This has lead to a situation where requirements are seen as linear and procedural in nature, rather than being parallel and functional². In the case of the AC Amplifier the test requirement has not been captured rather the implementation has been captured.

The choice of using static or dynamic descriptions for test requirements depends on the final user. Dynamic signal descriptions are easier to follow and have less interactions making diagnosis easier. Static signal descriptions are more portable, because they contain less ambiguity, and allow more complex testing with scenarios. What is important is that the SDTD standard supports a mix of both.

The following diagram shows the key block diagram design for a frequency sweep AC Amplifier. The remaining examples show different variations of the dynamic test requirement.

² The word functional here means operating in its functional environment, as the unit would be in service and not function test as opposed to diagnostic test.

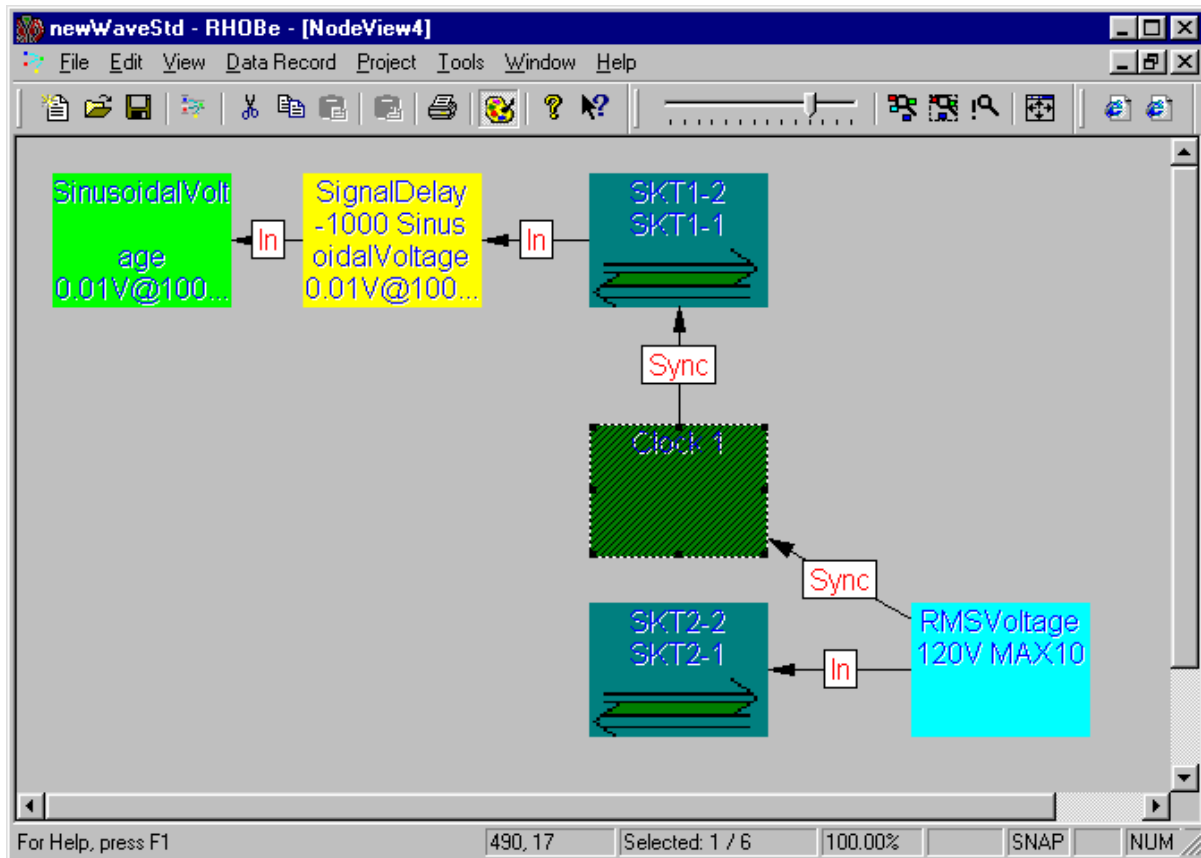


Figure 16. AC Amplifier Block Diagram Design

4.2.1 TPL

The following example shows the dynamic test requirement where the **highlighted** items mean implement in an appropriate carrier language

```

*****
*   Test Language       :   TPL
*   UUT Type           :   AC Amplifier
*   Model               :   AC-01
*   Serial number      :   AcAmp 001
*****
//
//
//*****
Begin the preamble part that contains the declarative statements
* Note: (GO, NOGO,) HI, LO are A2K Global Variables
*****
//
function Diagnosis-1()
output("Adjust Gain Control Lower");
//
function Diagnosis-2()
output("Adjust Gain Control Higher");
//
*****

```

```

*      End the preamble part      *
*****

output("Start Testing") ;
remove all;
Setup AC_SIGNAL ac_ampl 20.0mV errlmt +-0.5 %, freq 100.0Hz errlmt +-5.0% range 100:1000Hz
    As AC1;
Connect AC1 CNX HI SKT1-1, LO SKT1-2;
Measure RMSVoltage into Vout Nominal 120.0V Max
    CNX HI SKT2-2 CNX LO SKT2-4 As rmsvltg;

    for Freq:= 100.0 ; 100.0 < Freq > 1000.0; Freq ++100.0
    {
        Change AC1 Freq Freq;
        Fetch rmsvltg into Vout;
        Gain = Vout/20 ;
        Compare(Gain LL 195 UL 205);
        if HI = True Diagnosis-1();
        else if LO = True Diagnosis-2();
    }
Remove all;
output("End of AC Amplifier Gain Test") ;
//
*****
*      End of procedural part      *
*****

```

Figure 17. DC Amplifier TPL Test Requirement

4.2.2 Embedded Test Procedure Layer in Native Language (VB)

Any test requirement capture will need to compare measured values against expected values. The COMPARE statement is interesting because the IDL *Compare* method acts not on the measurement variable but on the measurement object. To overcome this in the example we re-assign the calculated value back to the measurement object.

This example also shows how the current ‘Verb’ definitions extend the meaning of the ATLAS 716 equivalents. In this case the signals are removed when they are no longer used or needed. The effect of this is that after each measurement the input signal is finished with, because there is nothing else to do. The signal gets released and therefore may open any connection relays through the *Connection* object. The result is that each measurement closes any relays, takes the measurement and then opens the relays. The solution is to define ARM and READ such that the signal is not released until an explicit DISCONNECT is performed.

Note the use of the ‘AS’ clause, to identify the signal object, remember the *SignalFunction* graphs remains regardless of whether a signal is present or not. The graph does not define the signal at that time, only at some time in the future when it is *Run* will the signal exist.

In the following example the highlighted items are optional.

```

/*****\
*      Test Language      :      ATLAS95
*      UUT Type          :      AC Amplifier      *

```

```

` *      Model          :      AC-01          *
` *      Serial number  :      AcAmp 001     *
\*****/
`//
`/*****\
` * Begin the preamble part that contains the declarative statements *
\*****/
`//
Subroutine Diagnosis_1
    Output "Adjust Gain Control Lower"
End Subroutine

Subroutine Diagnosis_2
    Output "Adjust Gain Control Higher"
End Subroutine
`//
\*****\
` *      End the preamble part          *
\*****/
`//
`//
\*****\
` * Begin the procedural part that contains all the UUT test statements *
` * that call the declarative statements in the preamble part          *
\*****/
`//
\*****\
` *      Start testing the UUT          *
\*****/
`//
Output "Begin AC Amplifier Gain Test"
`apply AC_SIGNAL ac_ampl 10mV errlmt 5%, Freq 100Hz errlmt 5% Range 100Hz:1kHz
` CNX HI SKT1-1, LO SKT1-2 AS freqSouce
`measure RMSVoltage INTO Measment, Nominal 120V MAX CNX HI SKT2-2, LO SKT2-4
` AS rmsRead

For freq = 100 To 1000 Step 100
    `Change freqSource Freq freq
    `Read rmsRead INTO Measment
    Gain = Measment / 20
    RmsRead.Measurement = Gain
    `Compare rmsRead UL 205mV LL 195mV]
    if rmsRead.NoGo then
        if rmsRead.Hi then
            Output "Gain High"
            Diagnosis-1
        else
            Output "Gain Low"
            Diagnosis-2
        end if
    Exit Next
End if

Next
\These are not necessary
\ [REMOVE (rmsRead)]
\ [REMOVE freqSource]

\ Inform user:      End of UUT testing /
Output "End of AC Amplifier Gain Test"
`//
\*****\
` *      End of procedural part          *
\*****/

```

Figure 18. DC Amplifier TPL Test Requirement

4.2.3 VB Native Language and A2k Basic Components

If our requirement was to match an original control language rather than define the test specification, we could still do this by making use of the Signal Function's **Conn** references. **Conn** references are similar to **In** references and allow connection statements to know where the signal flows. But it does not infer any signal control, so the programmer is fully responsible for making sure every thing is turned on and in the correct order. This enables us to map existing dynamic test requirement semantics.

The previous example and this one shows how when using dynamic test requirements test semantics can be subtly changed, in this case we can optimize our relay usage by changing our signal models.

Note the sense object *RMSVoltage* must use either **In** or **Conn** to know where to take the measurement.

In the following example the **highlighted** items are optional.

```
\*****\  
\ *   Test Language       :   ATLAS95  
\ *   UUT Type           :   AC Amplifier      *  
\ *   Model              :   AC-01            *  
\ *   Serial number     :   AcAmp 001        *  
\*****/  
\//  
\*****\  
\ * Begin the preamble part that contains the declarative statements *  
\*****/  
\//  
Subroutine Diagnosis_1  
    Output "Adjust Gain Control Lower"  
End Subroutine  
  
Subroutine Diagnosis_2  
    Output "Adjust Gain Control Higher"  
End Subroutine  
\*****\  
\ *   Define out Model   *  
\*****/  
Dim A2k  
Set A2k = CreateObject("Atlas2k.Resource")  
'define AC_SIGNAL Voltage 10mV errlmt 5%, Freq 100Hz errlmt 5% Range 100Hz-1kHz  
\    CNX HI SKT1-1 LO SKT1-2 AS freqSouce]  
Set ac1 = A2k.Require("AC_SIGNAL")  
    ac1.ac_ampl = "10mV errlmt 5%"  
    ac1.freq = "100Hz errlmt 5% Range 100Hz-1kHz"  
Set ac1cnx1 = A2k.Require("TwoWire")  
    ac1cnx1.HI = "SKT1-1"  
    ac1cnx1.LO = "SKT1-2"  
ac1cnx1.In = ac1.Out  
Set freqSouce = ac1cnx1  
  
'define (RMSVoltage) Nominal 120V MAX CNX HI SKT2-2, LO SKT2-4  
\    AS rmsREad  
Set mivcnx = A2k.Require("TwoWire")  
    mivcnx.HI = "SKT2-2"  
    mivcnx.LO = "SKT2-4"  
Set miv = A2k.Require("RMSVoltage")  
    miv.Nominal = "120V MAX"  
    miv.UL = "205mV"  
    miv.LL = "195mV"  
miv.Conn = mivcnx.Out 'Identifies where to make the measurement  
Set rmsRead = miv  
\//
```

```

\*****\
\ *      End the preamble part          *
\ \*****/
\//
\//
\*****\
\ *      Begin the procedural part that contains all the UUT test statements    *
\ *      that call the declarative statements in the preamble part            *
\ \*****/
\//
\*****\
\ *      Start testing the UUT          *
\ \*****/
\//
Output "Begin AC Amplifier Gain Test"
freqSouce.Out.Run 'Apply Frequency Source
mivcnx.Out.Run 'Connect SKT2-2 SKT2-4 for measurement

For freq = 100 To 1000 Step 100
    freqSouce.Freq = freq
    freqSouce.Out.Change 'Change output signal of Freq Source

    rmsRead.Run 'Take measurement
    measment = remRead.measurement 'fetch measurement into variable
    Gain = Measment / 20
    rmsRead.Measurement = Gain
    rmsRead.UL = "205mV"
    rmsRead.LL = "195mV"
    if rmsRead.NoGo then
        if rmsRead.Hi then
            Output "Gain High"
            Diagnosis-1
        else
            Output "Gain Low"
            Diagnosis-2
        end if
    End Next
End if

Next
\// Inform user:      End of UUT testing /
Output "End of AC Amplifier Gain Test"
\//
\*****\
\ *      End of procedural part          *
\ \*****/

```

Figure 19. DC Amplifier TPL Test Requirement

4.2.4 ATLAS 716-95

This ATLAS 716-95 test requirement has no DECLARE and DEFINE statements. But its flow control of testing is determined by the order of the ATLAS 716-95 statements and the first-order predicate logic IF-THEN-ELSE.

The ATLAS 716-95 Test Requirement TR-2 is in Clause 8.20.2.2 on page 64 of the IEEE Std.771-1998.

```

C*****\
*      Test Language          :      ATLAS95      *
*      UUT Type              :      AC Amplifier  *
*      Model                 :      AC-01         *
*      Serial number         :      AcAmp 001     *
\*****$
C$

```

```

C*****\
* Begin the preamble part that contains the declarative statements *
\*****$
C$
000010 REQUIRE, GLOBAL, 'STIM-F', SOURCE,
        CONTROL, AC SIGNAL, FREQ RANGE 100 HZ TO 1000 HZ BY 100 ERRLMT +-5 PC,
        CAPABILITY, AC SIGNAL, VOLTAGE 20 MV ERRLMT +-0.5 PC $
C$
C$
000050 DEFINE, 'DIAGNOSIS-1', GLOBAL, PROCEDURE $
000053 OUTPUT, C'ADJUST GAIN CONTROL LOWER' $
000056 END, 'DIAGNOSIS-1' $
C$
000060 DEFINE, 'DIAGNOSIS-2', GLOBAL, PROCEDURE $
000063 OUTPUT, C'ADJUST GAIN CONTROL HIGHER' $
000066 END, 'DIAGNOSIS-2' $
C$
C*****\
*      End the preamble part      *
\*****$
C$
C*****\
* Begin the procedural part that contains all the UUT test statements *
* that call the declarative statements in the preamble part *
\*****$
C*****\
*      Start testing the UUT      *
\*****$
003000 APPLY, AC SIGNAL, VOLTAGE 20 MV ERRLMT +-0.5 PC,
        FREQ 100 HZ ERRLMT +-5 PC,
        CNX HI SKT1-1 LO SKT1-2 $
10 MEASURE, (VOLTAGE INTO 'MEASMENT'), AC SIGNAL,
        VOLTAGE MAX 120000 MV,
        FREQ MAX 1000 HZ,
        CNX HI SKT2-2 LO SKT2-4 $
C   Prepare measurement resource for subsequent READs $
20 SETUP, (VOLTAGE), AC SIGNAL,
        VOLTAGE MAX 120000 MV,
        FREQ MAX 1000 HZ,
        CNX HI SKT2-2 LO SKT2-4 $
30 CONNECT, (VOLTAGE), AC SIGNAL,
        VOLTAGE MAX 120000 MV,
        FREQ MAX 1000 HZ,
        CNX HI SKT2-2 LO SKT2-4 $
C$
003100 FOR, 'STIM-F' = 100 THRU 1000 BY 100, THEN $
05  CHANGE, AC SIGNAL,
        VOLTAGE 20 MV ERRLMT +-0.5 PC,
        FREQ 'STIM-F' RANGE 100 HZ TO 1000 HZ ERRLMT +- 5 PC,
        CNX HI SKT1-1 LO SKT1-2 $
10  READ, (VOLTAGE INTO 'MEASMENT'), AC SIGNAL,
        VOLTAGE MAX 120000 MV,
        FREQ MAX 1000 HZ,
        CNX HI SKT2-2 LO SKT2-4 $

```

```

C$
15  CALCULATE, 'GAIN' = 'MEASMENT'/ 20 $
20  COMPARE, 'GAIN', UL 205 LL 195 $
25  IF, NOGO, THEN $
30    IF, HI, THEN $
35      OUTPUT, 'GAIN HIGH' $
40      PERFORM, 'DIAGNOSIS-1' $
45  FINISH $
50  ELSE $
55      OUTPUT, 'GAIN LOW' $
60      PERFORM, 'DIAGNOSIS-2' $
65  FINISH $
70  END, IF $
75  END, IF $
80  END, FOR $

C$
003200 REMOVE, (VOLTAGE), AC SIGNAL,
          VOLTAGE MAX 120000 MV,
          FREQ MAX 1000 HZ,
          CNX HI SKT2-2 LO SKT2-4 $

C$
10 REMOVE, AC SIGNAL,
          VOLTAGE 20 MV ERRLMT +-0.5 PC,
          FREQ 'STIM-F' RANGE 100 HZ TO 1000 HZ ERRLMT +- 5 PC,
          CNX HI SKT1-1 LO SKT1-2 $

C*****\
*      End of procedural part      *
\*****$

```

Figure 20. DC Amplifier TPL Test Requirement

5 CONCLUSIONS

The Test Procedure Language [TPL] layer complements the robust signal definitions of the SDTD standard to create a complete and comprehensive strategy for the capture of test requirements. The SDTD defines a rich set of signal definition capability allowing the subtleties of existing test languages to be mapped onto SDTD components.

Test requirement capture is of paramount importance for UUT maintenance and support. The ability to capture test requirements in COTS languages and for them to have identical meanings across automatic test systems (ATSs) is a major advantage to the industry.

The SDTD provides not only the language for capturing test requirements but also the semantic and mathematical meaning for simulating electrical signals, without constraining the user to predefined and fixed signal noun and verbs, but still providing a common language framework.

6 REFERENCES

- [1] *IEEE Std 716-2000 Draft H*, June 2002:IEEE TP Working Group.
- [2] *IEEE Std 771-1998 ATLAS 716 User Guide*.