Implementing IEEE 1641 - Compilation Techniques (to IVI Driver Code)

Matt Cornish
EADS Test Engineering Services (UK) Ltd
matt.cornish@eads-ts.com

Abstract – **Drawing on a recent study, sponsored by the UK MoD, this paper provides details of a compile-time approach to the implementation of an IEEE Std. 1641™ [1] test program (in contrast to previous implementations, which have adopted a run-time approach). Consideration is given to methods for capability description of test resources, through IEEE ATML [2]. In addition, comparison is made with the general run-time approach [3], in terms of portability and validation.**

As a detailed view of this particular aspect of a 1641 implementation, this paper incorporates the example tests; Gain and 1 dB Compression Point, for a mobile communications device. These tests are defined using the Standard's Test Signal Framework (TSF); test programs are produced using the TSFs in the C# carrier language; IEEE ATML Test Station and Instrument Description are created and used to determine suitable test resources; and, an XML document is created and used to create a translation from the IEEE 1641 & TSF defined test instructions to the test resources' IVI driver calls; this process effectively 'compiling' the C#, IEEE 1641 test program into an IVI test program (in this case, the native driver framework for the test platform).

Keywords – IEEE; 1641; Signal; ATML; Implementation; IVI

I. INTRODUCTION

IEEE Std. 1641 Signal & Test Definition [1] brings portability and resistance to obsolescence through signal-based test definition. Not a test language per se; IEEE 1641 is intended to be used within COTS (Commercial Off-The-Shelf) programming languages & development environments to reach as many platforms as possible, through IDL (Interface Definition Language), XML (Extendable Mark-up Language) and TPL (Test Procedure Language).

Up until only recently, implementations have been primarily run-time systems, either providing signal-based instrument drivers or creating a live mapping to some internal driver architecture.

Now, there exist a number of IEEE 1641 implementations in development that perform translation or compilation of signals into a further layer of driver code (such as IVI or ATLAS), with the purpose of bringing the benefits of signal-based test definition to existing run-time systems. Such examples of these can be found in the ATML demonstration, seen at AutoTestCon 08, and TYX SigBase, producing ATLAS-based code; each having benefits in their own right.

One of the major objectives of IEEE 1641 has always been that of 'openness'. Indeed, IEEE 1641 forms a key element of the UK MoD's Open System Architecture [5]; a document and policy intended to give the maximum choice of TPS development tools, through interoperability, while the TPSs themselves remain optimally portable and resistant to obsolescence. This paper is concerned primarily with the use of IEEE 1641 defined IDL (Interface Definition Language) and XML, since these are open to the greatest & most familiar choice of COTS test languages, compilers and development environments.

II. REVIEW OF IEEE 1641 TEST PROGRAM INTERFACING

IEEE 1641 defines a Require (<SignalDescriptor>, [UniqueID]) method; where the SignalDescriptor is any BSC (Basic Signal Component), TSF (Test Signal Framework – signal library component) or XML signal definition (i.e. an 'anonymous TSF') and a UniqueID, allowing specific resource information to be specified (typically as a result of some platform-specific allocation activity). The Require method must return an interface that supports manipulation of the signal's attributes, in addition to the IEEE 1641 defined Run(), Change() & Stop() methods.

Thus:

```
// Using my TSF library
FM_SIGNAL myFMSig
myFMSig = Require(FM_SIGNAL)
myFMSig.modIndex = 8%
myFMSig.Out.Run
```

Figure 1 Example of a Test Program Using IEEE 1641 IDL

Figure 1 shows an example of the IEEE 1641 defined interface being used to construct and execute a test signal, where the test signal (FM_SIGNAL) is defined in a TSF library, loaded into the test system. The TSF library format is defined by 1641 and may be expressed using XML, XSD (XML Schema Document) or IDL.

IDL is an industry standard interface definition language, in this case used to define the object type, FM_SIGNAL.

XML is used to describe the complete TSF behaviour in terms of 1641 signals, enabling the test system to determine a resource capable of implementing that TSF, where the test resources are also described in terms of 1641 signals (e.g. using ATML Test Station Description).

```
<!-- ATML Test Description →
<Operation xsi:type="OperationSetup" … >
  <Sensor>
    <LocalSensorSignalReference localSignalID="ls1"/>
      <std:Signal>
        <myTSFLib:FM_SIGNAL name="myFMSig" modIdx="8 %"/>
      </std:Signal>
      </Sensor>
  …

<Operation xsi:type="OperationConnect" ... >
  <Signal>
    <LocalSignalReference localSignalID="ls1"/>
  </Signal>
  …
```

Figure 2 ATML Test Description Example of an IEEE 1641 Signal

Figure 2 shows the same 1641 signal as part of an ATML Test Description. Test Description's 'Operations' are based on 1641's Test Procedure Language, in equivalent to 1641's IDL interface. Hence, 'OperationSetup' and 'OperationConnect' map onto the IDL Require and Run, respectively.

In this case, an XSD is used to define the interface (attributes) of elements with namespace prefix myTSFLib.

Note that Test Description is not a test program, but describes a test that is to be implemented by a test program. The process of generating test programs from ATML Test Description will hinge on the ability to translate its signal parts into executable code.

```
//<TPL>
//Setup FM_SIGNAL modIdx 8 %
// as source myFMSig;
//Connect myFMSig …
// as connection toUUT;
//</TPL>
```

Figure 3 IEEE 1641 TPL Example Test Program

Finally, Figure 3 shows a TPL example, again, of the same test program.

TPL is targeted at test programming languages that do not support IDL interfacing technologies (typically legacy), such as C, and is intentionally familiar to ATLAS users. In use, TPL is formed in the comment fields of the test program and must be translated to the same carrier language before the carrier's own compiler can process it.

Note that ATML Test Description's 'Operations' and 1641's TPL & IDL methods are equivalent, so the tests described in Figure 1 , Figure 2 and Figure 3 are identical.

III. IEEE 1641 Demonstration Phase II.V

Phase II.V of the study programme sponsored by the UK MoD investigated the use of 1641 signal definition for amplifier characterisation, implemented through a translation approach to generate an IVI test program [4].

The approach of this study to generating IVI code was to take a 1641 test program, written in C#, using a 1641 IDL defined COM (Component Object Model) interface, and use that same COM interface to translate the 1641 method calls into IVI method calls.

The significance of this approach is that the translation process used the existing compiler and run-time tools of the language. This same translator would work equally with C++, VB, Pascal, etc. – thus, supporting the Open System Architecture intent to maximise the choice of TPS development tools.

Additionally, as a by-product of supporting the IDL defined programming interface, productivity enhancements in many COTS development environments, such as auto-complete, interface browsing, keyword help and chroma-coding, are also supported.

IV. TSF Programming Interface

As seen in section II, the interface for running a test program is already defined by 1641. Additionally, the Basic Signal Components of the Standard also have defined interfaces.

However, it is expected (and is the case) that, in general use signals more complex than BSCs will be required and that these will be built within the 1641 defined Test Signal Framework. These custom signal components (called TSFs – in reference to the framework) consist of a model and an interface, which may be expressed in a variety of formats, to suit the application.

EADS newWaveX provides a 1641 TSF design environment that links the 1641 Standard's data formats together [3], such that a TSF library may be exported as XML, XSD or IDL files.

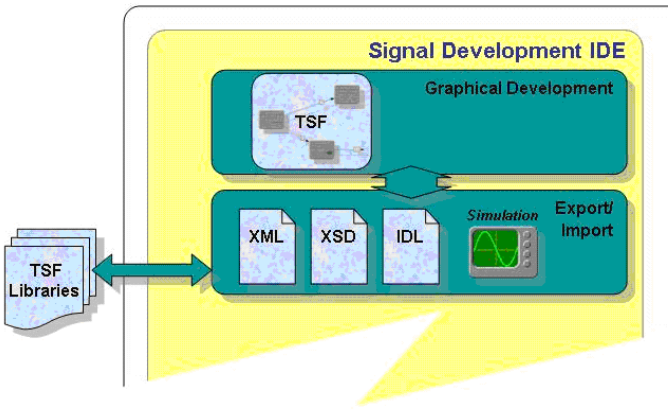

Figure 4 newWaveX TSF Signal Modelling

Figure 4 shows how the newWaveX Signal Development environment links a graphical development and simulation environment with a variety of different file format exports. Behind the scenes, the graphical development environment is rule-based and ensures the validation of the signals that it is used to develop.

The exported IDL may be compiled using a COTS (Commercial Off The Shelf) compiler to make a type library supporting the use of the TSF in test programs. The same type

library is used by newWaveX to provide a TSF wrapper interface to its translation framework, as shown in Figure 5 .

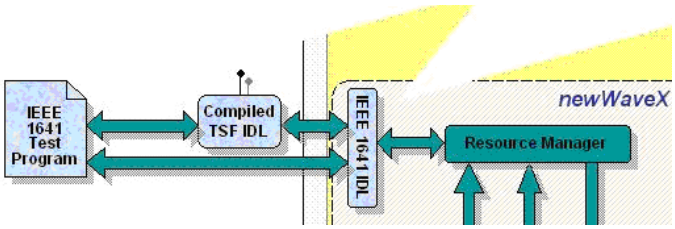

Figure 5 IEEE 1641 Core Interface Supplemented by Exported TSF Interface

V. ATML Capability Description

```
<hc:Capability name="FMSignal">
  <hc:Interface/>
  <hc:SignalDescription>
    <std:Signal Out="odcp1">
      <tsf:FM_SIGNAL name="fmSig" modIdx="range MAX 30 %"/>
    </std:Signal>
  </hc:SignalDescription>
</hc:Capability>
```

Figure 6 ATML Capability Description Example

IEEE ATML provides an XML format with which to describe a resource's capabilities in terms of signals, using IEEE 1641, a basic example of which is shown in Figure 6 This capability description may be written in terms of the TSFs expected to be used on a test station or more generically, using Basic Signal Components.

The first stage of implementing a 1641 test signal requirement is a mapping of that signal onto a signal found in the capability description. This applies whether at run-time, some pre-allocation step or for translation/compilation.

IEEE 1641 Signal Modelling Language provides a means to validate that these capability descriptions against the performance of the resources they describe.

VI. Mapping 1641 onto Driver Functions

A. *1641 Interface*

Referring back to section II, IEEE 1641's defined core programming interface provides a number of functions. These include:

```
Require <Signal>, [Allocation Information]
Run [timeout]
Change [timeout]
Stop [timeout]
```

Further, each Basic Signal Component has its property interface defined by the Standard. For Example:

```
RMS:      nominal
          condition
          gateTime
          samples
          UL
          LL
          measuredVariable
```

In the case where the required signal is specified using XML (known as an 'Anonymous TSF'), an additional function retrieves individual elements of that signal:

```
Item <Signal Name>
```

The returned signal element may then be referred to through its BSC interface.

These core functions and properties are all that is needed to interpret any 1641 signal, since any TSF interface wrapper (see section IV) can be mapped through these functions.

B. *Driver Interface*

Through 1641 Signal Modelling and the ATML Capability Description, it is possible to determine the driver functions that will implement their associated signal attributes.

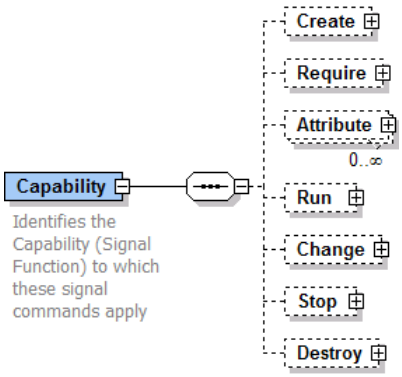

Figure 7 Actions Required to Implement 1641 Capabilities

Figure 7 illustrates the structure of the actions required to implement a given resource capability in terms of its 1641 signals. Two additional actions, beyond those of 1641, are required to 'Create' and 'Destroy' instance data, in the case of unmanaged code.

Each action then breaks down into two types of driver command:

1) *Command*

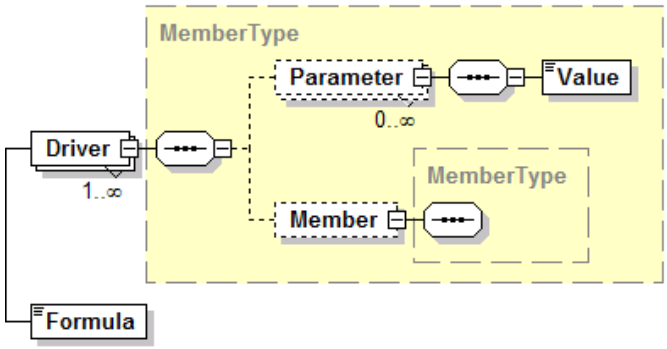

Figure 8 Driver Command

Figure 8 illustrates a possible structure of the driver commands required to implement an action, such as switch on an instrument's output or read back a value. Additionally, a formula may be required.

A formula may be specified using the same terms found in IEEE 1641, e.g.:

```
"{{MeasOutputPower.measurement} - {input_pwr}}"
```

Figure 9 Example Formula

Since each of these 1641 attributes are, themselves, mapped to actions within the same instance document, their driver commands can be determined.

2) Assignment

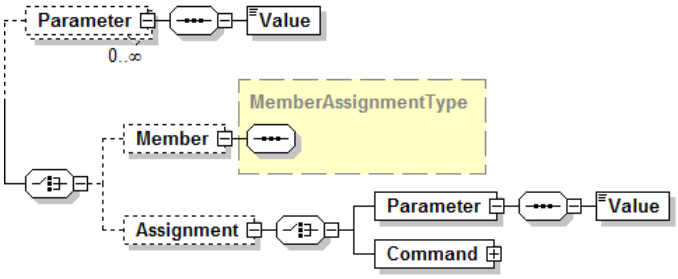

Figure 10 Driver Assignment

Figure 10 illustrates a possible structure of the driver commands required to implement an assignment action. In this structure, assignment may either be a value or a value obtained through a further driver command.

C. *In use*

The structure described above is implemented in newWaveX Platform Development and has been trialled in IEEE 1641 Demonstration Phase II.V [4], for the UK MoD; successfully describing all the IVI driver commands required to implement 1 dB Compression Point and Gain characterisation of a communications amplifier module using Rohde & Schwarz' Vector Signal Generator, Spectrum Analyser and Vector Network Analyser.

So, for example:

```
myODCP.input_pwr.magnitude = dPower;
```

Would return the driver string:

```
RsRFSigGen.RF.Level = <-20>
```

And, the calling program could equally be an application parsing an ATML Test Description document or an application interpreting TPL statements. I.e. the processes described here work equally for 1641 IDL COTS test programs, ATML Test Descriptions and 1641 TPL test programs.

Note that, for the translator, it is not possible to determine whether the assignment value, '-20', is a literal or a string, so it is returned as `<-20>`, denoting a possible variable.

VII. TRANSLATION FRAMEWORK

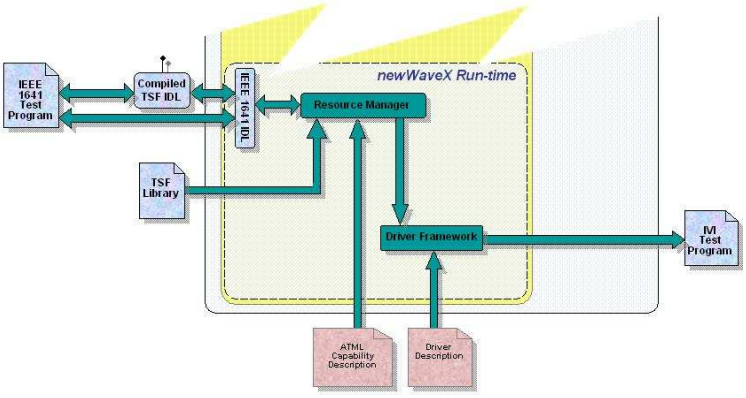

Figure 11 newWaveX Signal to Driver Translation Framework

The application framework for the implementation of the translation process, using the driver command structure described in section VI, is shown in Figure 11 .

Translation is achieved by matching the signal requirement, supplied through the 1641 IDL interface, to the signal capability in the ATML Capability Description document. The resource identified is located in the in the Driver Description document and the IVI driver commands are assembled.

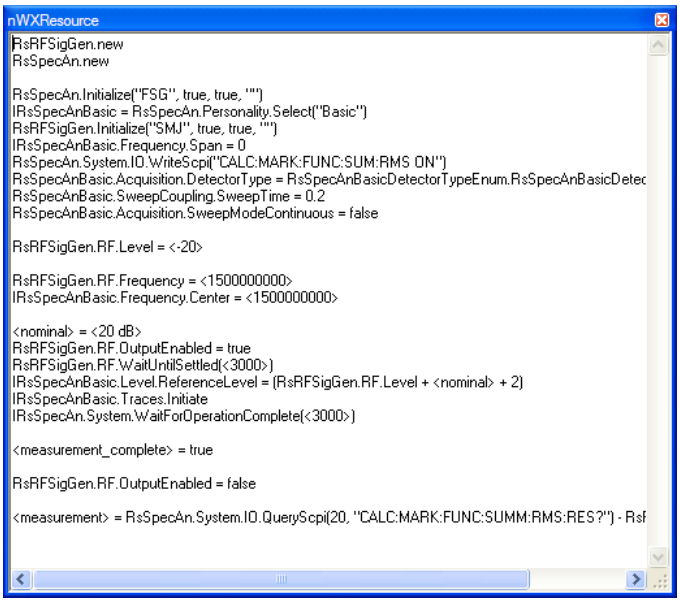

Figure 12 Example newWaveX Driver Code Generator Output

The C# IEEE 1641 test programs, written for the 1641 Demo II.V [4] were executed against ATML Capability Descriptions and Driver Descriptions for a Rohde & Schwarz' signal generator, spectrum analyser & vector network analyzer. Figure 12 shows a sample of the IVI driver code output from this process. This driver code may then be substituted into the test program to produce the final driver-based test program.

VIII. Test Program Generation

Clearly, the output generated in section VII is not a syntactically correct test program and, of course, syntax varies across programming languages. What remains is for the full test program to be generated, applying the appropriate syntax to the output driver code.

What the approach presented in this paper shows is an implementation that can be applied across a broad range of test programming languages. Through the IEEE 1641 defined interfaces, this same translation module may be embedded in a wide variety of test program development environments, to add a 1641 to native driver translation service.

IX. Conclusions

A method has been shown that translates IEEE 1641 test signal definitions (including ATML Test Description) into resource driver code, for used when compiling test programs.

ATML Capability Description has been used to provide for automatic resource allocation, also using 1641 signal definitions. This approach brings the benefit of the Standard's Signal Modelling Language, which may be used to validate these capability descriptions against the performance of the resources they describe.

A further XML document structure has been described, supplementing the capability description, to provide driver descriptions that implement those capabilities on the described resources.

An architecture has been shown, making use of the above description documents, to translate an IEEE 1641 test definition into driver code (in this case, IVI) required to implement that test definition on a given test platform.

This process has been shown to use the IEEE 1641 defined test programming interfaces, to provide an implementation that can be applied across a broad range of test programming languages, through interoperability and in support of UK MoD's Open System Architecture [5].

Additionally, such an implementation, through the 1641 interfaces, is open to embedding in development environments using ATML Test Description and 1641 TPL.

In all, a compile-time approach (in contrast to previous studies, which have adopted a run-time approach) to the implementation of IEEE 1641 and ATML on test equipment, yielding a more native test program, while retaining the validation benefits of the Standards.

X. References

[1] IEEE Std. 1641™-2004, IEEE Standard for Signal and Test Definition. IEEE 2005

[2] IEEE Std. 1671, Automatic Test Mark-up Language. IEEE 2008

[3] Implementing IEEE 1641 – Resource Drivers & COTS Languages. IEEE AutoTestCon 2008

[4] IEEE Std. 1641™ Demo - Gain & 1 dB Compression Point, with a Translation to an IVI Implementation, through Signal Description of Test Resources

[5] Open Systems Architecture For Automatic Test Systems, SR/DLO/0501 Issue 2