Implementing IEEE 1641 – Resource Drivers & COTS Languages

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

Abstract – IEEE Std. 1641 Signal & Test Definition [1] has been shown to bring portability and resistance to obsolescence through signal-based test definition in conjunction with signal-based resource description; with the UK MoD providing for a phased demonstration programme to show the Standard used in various test languages and test resource environments [5][6][7].

Intended to show complete systems, from test requirement to UUT, the UK MoD demonstration phases also provided an interface to control real test resources, but how was this achieved and how did IEEE 1641 support its implementation?

Since the first publishing of IEEE 1641, the IEEE 1671 ATML (Automatic Test Mark-up Language) [8] group of standards has begun trial use. ATML makes significant use of IEEE 1641 for signal modelling; and, some its concepts, namely Test Description & Instrument Description, have been explored here.

Using IEEE 1641's signals, attribute types, connections & events, this paper proposes a method of automatically generating & maintaining interface definition libraries (IDL) from TSF (Test Signal Framework) libraries, for import/include into resource driver source code projects. Additionally, this is shown to ensure conformance to the TF library; support early binding test languages (e.g. C#), thus compile-time error detection; and, productivity features like auto-complete & keyword help in test development environments.

Keywords – 1641, ATML, Signal Modelling, Signal Modeling, Driver, Software Validation

I. INTRODUCTION

IEEE Std. 1641 Signal & Test Definition [1] brings portability and resistance to obsolescence through signal-based test definition that is purely in terms of the UUT. 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 (Extensible Mark-up Language) and TPL (Test Procedure Language). This paper is concerned primarily with the use of IDL and XML, since these are open to the greatest & most familiar choice of COTS test languages, compilers and development environments.

Many papers have been written to show how IEEE 1641 might be implemented, showing improvements over traditional methods for both test portability and resistance to obsolescence [2][3][4]. Indeed, the UK MoD has provided for a phased demonstration programme to show the Standard used in various test languages and test resource environments [5][6][7]. Additionally, the US DoD is pursuing the use of the IEEE 1671 ATML [8] group of standards, which make significant use of IEEE 1641 for signal modelling.

IEEE 1641 'Signal & Test Definition' is just that; a method of defining signals (to be applied or measured) and the structure of a test which encapsulates these signals to determine a pass/fail result. Though these signal definitions may be modelled using the Standard's Signal Modelling Language (e.g. to prove their defined behaviour versus that of the resources used to implement the test), the run-time system is treated as something of a 'black box'. An implementation of an IEEE 1641 run-time system must accept signal definitions through the methods defined in the Standard; such as that defined by the IDL informative annex.

However, this is not as complicated as it might first seem, as the Standard provides a number of information exchange formats which can be leveraged to support a software validation managed path from signal definition to a run-time system's source code project.

II. IEEE 1641 RESOURCE INTERFACE

IEEE 1641 defines a Require (<SignalDescriptor>, [UniqueID]) method that a run-time system must support; where the SignalDescriptor is any BSC (Basic Signal Component), TSF (Test Signal Framework) or XML signal definition (i.e. an 'anonymous TSF') and the UniqueID allows specific resource information to be specified (typically as a result of some platform-specific pre-compilation 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.

A. IEEE 1641 Basic Signal Components

In the case where BSCs are used to provide the SignalDescriptor, the Standard explicitly defines the interface that must be returned by Require():

```c
// Define MySignal
FM fm
    Sinusoid sin
    fm.Require(AM)
    fm.modIndex = 8
    sin.Require(Sinusoid)
    fm.In = sin.Out
    fm.Out.Run
```

Figure 1 - Example of a Test Language Using IEEE 1641 IDL

Figure 1 shows an example of IEEE 1641 defined IDL calls used to construct and execute a test signal directly from BSCs. The key elements, which are explicitly defined by IEEE 1641...
& therefore may be checked at compile time, are highlighted in *emboldened italics*.

**B. IEEE 1641 XML Signal Definitions**

To Require() an XML defined signal, it is possible either to specify the XML in-line with the code or as a URL.

Figure 2 shows an example of IEEE 1641 defined IDL calls used to construct and execute a test signal using an XML signal definition. The key elements, which are explicitly defined by IEEE 1641 & therefore may be checked at compile time, are highlighted in *emboldened italics*. It can be seen that considerably fewer elements may be checked at compile time. Though, that is not to say that a compiler could not be developed which could provide this additional checking or that the XML could not be validated as part of a separate process.

```c
// Define MyPMSignal
SignalFunction mySignal
mySignal = Require;
<Signal name="**" Out="**2" >
<Sinusoid name="Sin32"/>
<FM name="AM2" In="Sin32"/>
</Signal>
(FM)(mySignal.Item("FM2")).modIndex = 8
mySignal.Out.Run
```

Figure 2 - Test Language Example of an IEEE 1641 'Anonymous TSF'

**C. IEEE 1641 Test Signal Framework**

IEEE 1641's Test Signal Framework provides for the definition of re-useable signals and measurements (e.g. TACAN, AM SIGNAL, RADAR_RX) from more Basic Signal Components, including their programming interfaces. Such TSFs typically make up a library to be used with a particular group of tests or test platform; but the signals, events, attribute types & connections are all defined within IEEE 1641 and may be similarly represented in a variety of formats (e.g. XML or IDL). In use, TSFs are identical to BSCs, but functionally more complex.

Since the use of TSFs allows the interface to test resources to be extended and customised (a feature lacking in IEEE 1641’s predecessor, ATLAS), it is expected that TSFs are to be the typical building blocks for tests called by a test program.

Hence, the generation of resource interfaces that support the manipulation of TSF attributes is a key activity in IEEE 1641 test system development.

```c
MY_FM_SIGNAL myPMSignal
myPMSignal = Require(MY_FM_SIGNAL)
myPMSignal.modIndex = 8$
myPMSignal.Out.Run
```

Figure 3 - Test Language Example of an IEEE 1641 defined TSF

**D. Benefits of TSFs in Resource Interfaces**

By comparison of the examples given in this section, it is proposed that the use of TSFs presents two key benefits in writing test code for test resources:
- Maximised & earlier support for checking & debugging
- Concise, maintainable code

However, a potential future benefit that might arise out of in-line XML checkers is the ability to validate attribute values at compile time.

**III. REVIEW OF LATE VS EARLY BINDING**

Consider the two fundamental ways to use software interfaces to programmatically control another application, such as a test program controlling an resource driver in an ATS run-time system.

*Note*: Though the general terms 'late binding' & 'early binding' are sometimes associated with Basic language variants, neither this discussion nor IEEE 1641 are targeted or restricted to any particular language. For example, in COM 'late binding' and 'early binding' would refer to compiling without and with Dispatch IDs, respectively.

**Late binding**: a client application attempts to create a connection to a server application at run-time and to call its methods and attributes. It was not previously necessary to confirm the existence, methods or attributes of the server application.

**Early binding**: the compilation and usually writing process of a client application makes reference to a server application's defined interface.

*Note*: That is not to say that, in either case, the client application should not perform run-time checking to confirm that the server application does exist and can provide the required methods and attributes.

**A. Advantages of Late Binding**

Because a late binding client does not incorporate information about a specific version of a server interface, a late binding client/server relationship may be seen as more version-independent than an early bound client/server relationship. However, this may be seen as simply poor software configuration management.

Some software development environments don't actually support early binding, making late binding code projects more portable than early binding code projects.

Extra server interface information held in early binding code projects, which is not held in late binding code projects, may yield larger file sizes and longer compilation times.
B. Advantages of Early Binding

Faster execution speed for the client application, since the client already knows the location of the server application and its methods & attributes. In late binding, each of the server application's methods & attributes must be located before they can be called.

Because the server application's interface is known at the time the client application is being written, debugging and error checking is greatly simplified & quicker since many COTS compilers can check the code automatically. The late binding alternative is manually checking the code, though some specialist compilers may perform this checking.

Productivity enhancements in many COTS development environments, such as auto-complete, interface browsing, keyword help & chroma-coding, are supported by creating an early binding to a server application's interface.

C. Client/Server Binding in Test

Both late and early binding techniques are adopted in many test development environments.

IV. RUN-TIME VS. DESIGN-TIME RESOURCE ALLOCATION

Consider the two fundamental methods of allocating test resources on an ATS, as a result of a test requirement:

Run-time; test resources are automatically allocated while the test program is running, on the basis of a capability match between a test requirement and test resource. This may include compensation for path losses, according to the eventual match.

Design-time; test resources are matched to test requirements during an integration phase (this may include some automated process) and fixed for the life of the test.

IEEE 1641 TSF orientated test programs will support both these scenarios, since either; a required TSF is available on the ATS and can be allocated; or, a required TSF is not available on the ATS, but the TSF library can be opened to retrieve the signal model, from which a match can be found.

V. EXCHANGING IEEE 1641 SIGNAL MODELLING FORMATS

Figure 4 presents the general model of IEEE 1641 in terms of four layers. The Standard provides definition for each of the layers such that a verifiable signal model can be obtained from any layer and through a choice of languages.

There are refinements to this model, some of which include, from the bottom up:

SML (Signal Modelling Language) would typically be used only in signal modelling applications; possibly even only to validate the correct behaviour of signal modelling applications, which might use quicker and more efficient synthesis frameworks.

The BSC (Basic Signal Component) layer defines BSC interfaces in terms of IDL (Interface Definition Language), XML and TPL (Test Procedure Language), whereas their functional behaviour is defined in the terms of the SML layer. For example, a signal modelling application would typically present the BSC layer to support a user in developing signals, but would need to link to the SML layer to generate simulations of the signal models, as described above.

The TSF (Test Signal Framework) layer defines both the signal model (in terms of BSCs) and interface definition. Both signal model and interface can be defined together, using XML, in a TSF library. Furthermore, an XML schema description (XSD) of a TSF interface can be used to validate an XML description that uses that TSF (for example, ATML Test Description [8]), but this does not define the behaviour of the TSF signal model. IDL can be used to define a TSF interface for compilation against early binding test code, but, again, this does not define the behaviour of the TSF signal model. For example, an integrator might want to validate a signal model of a TSF against a particular test resource; whereas, a test programmer might want to validate test code through an early binding compilation with a TSF IDL interface definition.

At the very top layer is the carrier language, used to sequence the test signals and measurements defined through the layers below. At this layer a mix of the above formats may be used. For example, the IDL Require() method may accept a URL to an XML file, defining a <signal>.

Considering the possibilities listed, it may be seen that it is not simply the case that one format may be translated to another, either within the same layer or between layers.
VI. CREATING TSF PROGRAMMING INTERFACES WITH IEEE 1641 - A WORKED EXAMPLE

Through a hierarchical database architecture, all the Standard's informative data formats have been encapsulated and fronted by a graphical user interface, to provide a visual signal modelling environment for TPS.

A. Signal Modelling

Figure 5 shows a TSF that might be used to measure noise rejection in a radio receiver, modelled using IEEE 1641. The graphical representation is obtained from a drag-and-drop signal modelling environment supporting IEEE 1641 BSCs, but fronting a database architecture which captures the IEEE 1641 four layer model & interchange formats described above.

B. Signal Verification (Simulation)

Through the layered model, represented in the database architecture, it is possible to map the BSCs within the TSF model to the IEEE 1641 SML, to produce a functional model of the TSF.

Figure 6 shows a simulation created from the functional model described above. For signal modelling purposes, this simulation might be used only to confirm that the behaviour of the TSF is as intended by the designer; however, such a simulation might also be used to validate the correct operation of a test resource in an integration or even obsolescence management activity.

C. TSF Interface Definition

Though an IEEE 1641 TSF signal model refers to the TSF's interface attributes, these are defined separately, along with their types, ranges, default values and annotations. Figure 7 shows the interface attributes for this example TSF, defined in the database.
D. TSF Library Generation

The complete TSF is shown using IEEE 1641 XML notation in Figure 8, for completeness. It can be seen, however, that even this does not carry all the information required by the design process, such as the graphical layout information that would be needed to maintain the visual signal modelling environment. Thus, the database is still needed to bring all the information together.

Note: A brief inspection of the XML reveals the reftype of the BSCs in the TSF signal to be frequency; thus, the explanation as to how, for example, SignalDelay can be used to define the start frequency of a frequency sweep.

E. TSF Reuse in the ATML Group of Standards

The TSF Library code generated above is a complete definition of the TSF: It can be used to derive the TSF behaviour, by extrapolation back to the IEEE 1641 SML for both the TPS designer and ATS integrator. It can also be used to generate a programming interface to support early binding concepts, such as compile-time debugging and development environment productivity enhancements.

With the addition of an XML schema description, the TSF library can be reused in the ATML group of standards, for Test Description and Instrument Description. Such an XSD is shown in figure 9.

F. Resource Driver programming Interface

Consider now that the test definition and/or instrument description exist for the test and test system with which we are concerned. These are in terms of IEEE 1641 and align with the
examples provided thus far. Processes by which the layers and informative data formats may be traversed have been described. It is now possible to obtain the software interface definition that is compliant with the Standard, even though this TSF did not exist until we defined it. That interface is shown in figure 10, in terms of the IDL open standard.

```idl
library MyTSFLib
// Lines removed for brevity.
//RADIO_PWR

object,
    uuid("5CDA514B-EC0A-40AF-8283-A360A02F410B"),
    dual,
    helpstring("IRADIO_PWR Interface"),
    pointer_default(unique)
)
interface IRADIO_PWR : IMyTSFLib

enum [RADIO_PWR_BASE=(MYTSFLIB_BASE+256)]
{;
    [propget, id(RADIO_PWR_BASE+2), helpstring("Start frequency")]
    HRESULT start_freq[out, retval; Frequency pVal];
    [propsetref, id(RADIO_PWR_BASE+2), helpstring("Start frequency")]
    HRESULT start_freq[in; Frequency newVal];

    [propget, id(RADIO_PWR_BASE+3), helpstring("Stop frequency")]
    HRESULT stop_freq[out, retval; Frequency pVal];
    [propsetref, id(RADIO_PWR_BASE+3), helpstring("Stop frequency")]
    HRESULT stop_freq[in; Frequency newVal];
}

interface IRADIO_PWR

[noncreatable]
 coclass RADIO_PWR
[
default] interface IRADIO_PWR;

Figure 10 - TSF Interface defined in terms of IDL

This IDL definition of the TSF’s interface may usefully be used even before a driver is coded, since it can be compiled into a type library. A type library being useable to support test code compilation, even before the drivers are written, thus streamlining the test writing process. In the test development environment, this type library (in some cases this will need to be compiled as an empty driver stub) will support all the productivity features of modern COTS applications; auto-complete, interface browsing, keyword help & chroma-coding, etc.

Sufficient information also exists in the TSF library to create a simulation of the eventual run-time system, via the mapping to SML; a further tool to aid debugging.

Furthermore, the other informative formats, provided by the Standard, enable the source signal model to be available to maintainable code. Some specialist compilers may perform this checking using other techniques.

IDL (Interface Definition Language) provide the greatest support for design time checking & debugging, in addition to which, coding with TSFs yields the most concise & maintainable code. Some specialist compilers may perform this checking using other techniques.

Compilation of IDL can be achieved on a variety of platforms, using COTS tools, to create a type library, with which a driver code stub can be created.

Implementation of the driver is, of course, dependent upon the specific resources available on a given test system, but the correct behaviour of that implementation can be confirmed through the links to and use of synthesis and simulation provided for by the Standard’s Signal Modelling Language.

VII. CONCLUSIONS

Of the various methods for programmatic control of test resources, IEEE 1641 TSFs (Test Signal Framework), through IDL (Interface Definition Language) provide the greatest support for design time checking & debugging, in addition to which, coding with TSFs yields the most concise & maintainable code. Some specialist compilers may perform this checking using other techniques.

Productivity enhancements in many COTS (Commercial Off The Shelf) development environments, such as auto-complete, interface browsing, keyword help & chroma-coding, are supported by TSF IDL.

IEEE 1641 TSF orientated test programs will support both run-time and design time resource allocation, through the XML defined signal models; though, both allocation & coding may be greatly simplified where an ATS has a particular purpose or test domain for which a fixed set of TSFs can be provided.

IEEE 1641 TSF defined ATS may support both instrument obsolescence management and test interoperability through these TSFs and their rigorous definition using SML (Signal Modelling Language), which may be obtained through the Standard’s layered structure.

Furthermore, the IEEE 1641 layered & multi-lingual structure allows a link between test descriptions, test code, instrument descriptions & resource drivers which supports validation between any of these deliverable items.

These activities are strongly supported by the IEEE 1671 ATML (Automatic Test Mark-up Language) group of standards in particular, Test Description and Instrument Description.

REFERENCES

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