Logging

Gaining Access to the Inner Workings of Your TPS

Michael McGoldrick
Defense and Aerospace Division
Teradyne
North Reading, MA

***Abstract*—This paper describes an enhanced test system software architecture that includes a general use facility for passing test program data from an executing test program set (TPS) to one or more support applications running in parallel with the TPS. These support applications may include tools to assist the TPS developer in debugging the TPS, characterizing the unit under test, archiving test results data, and other applications that a test organization may find relevant and helpful. The same test system software architecture can also be used by instrument vendors in the design and implementation of the debugging tools that accompany their instrument drivers and other software.**

The paper also includes a description of a possible implementation of such an architecture, and demonstrates how it can be used to simplify the development of multiple and diverse types of support applications with little to no impact on the development of the TPS itself.

Keywords—TPS, Debugging, Tools, UUT, Test System, Architecture;

I. INTRODUCTION

A. TPS Development Challenges

The conversion of test requirements into actual TPSs can be a difficult task. It is the TPS developer's job to translate those test requirements into a working functional test program that generates an appropriate sequence of commands to a suite of instruments in the test system to properly generate stimulus and collect and analyze UUT responses to that stimulus to fulfill the UUT's test requirements.

Once that functional test program has been created, the TPS developer needs to debug the program, since inevitably the initial version of the program will not be perfect. To do this, he or she could use the debugging tools in the test application development environment to gain insight into the cause of the failure. While those tools may be able to report the values of variables and the contents of memory when stopped at a breakpoint, that information may be too low level to provide useful insight into the cause of a test failure. Instrument vendor software panels are another debugging option. However, those tools may not be able to sync to the current hardware state of an instrument, limiting their ability to help isolate test failures during TPS execution.

Part of the TPS debugging effort involves the qualification of the test program's test limits. How stable are the various collected UUT responses, and how close to the test limits do they lie, are questions the TPS developer needs to/should answer before the TPS is released. If the test results lie close to the test limits, then the TPS may exhibit marginal failures on good UUTs once deployed.

One way to address this is by adding dedicated code to the TPS to deal with test result analysis. However, adding this and other non-test related code (for example, code to log test results to a file) to the TPS increases its size and complexity, thereby lengthening its development time and increasing its cost. It can also lead to inconsistent or specialized implementations of those non-test-related tasks, as shared code for those tasks evolves over time or because each test developer provides his/her own custom implementations.

After a TPS is released, it is possible there are still some latent defects in the TPS due to timing issues, marginal tests, or differences in test instrumentation performance, and some further debug may be required. The resolution of such defects is more difficult at this stage because the development tools are often not present on a deployed test system:

- There is no access to the internal state of the test program at the point of failure.
- It may not be possible to control the execution of the released TPS to be able to bring instrument vendor debugging tools or soft front panels on line at the moment of failure.

B. Beneficial Test System Software Architecture Enhancements

Despite the seemingly varied nature of the issues described in the previous section, this paper demonstrates how all can be addressed through a common software architecture enhancement. This enhancement:

- Provides a means for migrating non-test related code out of the TPS to separate software utilities.
- Facilitates the creation of dedicated utilities that handle support tasks like logging test results to a file, analyzing and producing reports of test stability and margins, among other tasks.
- Assists in the debugging of released TPSs by providing a means to access internal test data while a TPS is executing.

Some advantages of using dedicated utilities for support tasks are:

- Enhancements may be made to the utilities without affecting the TPS.
- Eliminates customizations that might result in TPS-specific behaviors and maintenance issues.
- Provides a means for a test organization to expand the toolset provided by instrument or test system vendors to assist in debugging TPSs.

II. Proposed System Architecture

A. Publish/Subscribe Model

The basis for the enhanced system architecture described in the previous section is the Publish/Subscribe design pattern [1], which is related to the Observer design pattern [2].

The intent of the Observer design pattern is to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" [3].

The Publish/Subscribe design pattern builds on the Observer pattern by eliminating the tight coupling that otherwise exists among the objects: objects that publish data have no knowledge of the objects that subscribe to changes in that data and vice versa.

The system software architecture proposed in this paper follows the Publish/Subscribe design pattern. The architecture is depicted in the diagram below:

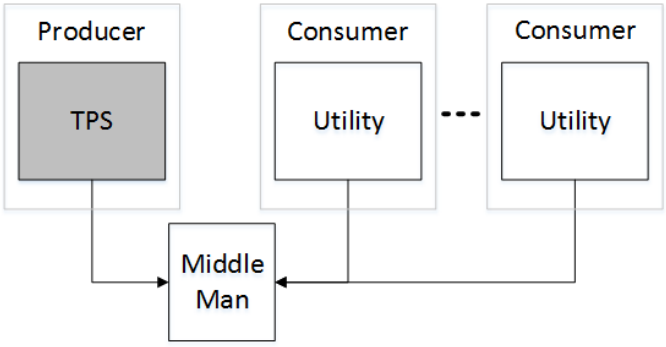

Fig. 1. Proposed Test System Software Architecture

The architecture components are described in more detail in the following sections.

B. Producers

Producers are programs that "publish", or generate, data to be logged. Typically, these programs are the TPSs themselves, but tools or other software components that control or monitor instruments or generate test data could also be included in this category.

C. Consumers

Consumers are programs that "subscribe to", or want to receive and process, data generated by producers. What these programs do with that data is up to them. Possibilities include archiving the data to a file or external database, displaying the data 'live' in a graphical user interface (GUI), or anything else that a test organization might deem useful. A TPS may communicate with multiple such consumers.

D. The Middle Man

The Middle Man is a software component that is responsible for passing producer generated data to be "logged" to interested consumers. In this context, "logged" is synonymous with "processed"; it does not imply archival of the data, although that is certainly something that a consumer could do with it.

Producer and consumer programs communicate with the Middle Man when they start up to let it know what kind of data they generate and respond to, respectively. The Middle Man maintains a list of subscriptions so that it knows what data to route from which producer(s) to which consumer(s).

The Middle Man component needs to already be running to allow producers and consumers to register with it upon their own start up. A practical way of achieving this is to implement it in the form of a Windows service that starts automatically upon system PC boot.

E. Data Passing and Subscriptions

One approach for implementing the Middle Man would be to simply pass all data generated by producers to all consumers, and let the consumers filter out the unwanted information. However, this could lead to degraded system performance, depending on the numbers of active producers and consumers, the amount of data being sent, and the frequency of that data transmission. At best, this degraded system performance would lead to longer test execution times. In some scenarios, it is possible that test timing would be impacted, possibly resulting in intermittent UUT performance and marginal test results.

A better approach would be to make the Middle Man more intelligent so that it only passes data to those consumers that need it. Toward this goal, the Middle Man maintains a subscription list to efficiently manage the distribution of data to be logged. Assisting the Middle Man in this effort requires the cooperation of producers and consumers:

- A producer announces itself to the Middle Man upon start-up. In response, it receives a list of the types of data (the Relevant Data Type list) to which consumers currently want access, and only sends data to the Middle Man if the type of that data matches one of the entries in the Relevant Data Type list.
- A consumer announces itself to the Middle Man upon start-up as well, telling it the type(s) of data it wants to receive.

- A consumer also tells the Middle Man when it is going away or otherwise no longer wishes to receive data. In response, the Middle Man updates its list of subscriptions. If there is no longer any interest in previously subscribed data types, the Middle Man sends an updated Relevant Data Type list to all active producers.
- A producer should also tell the Middle Man when it is going away to help it manage its subscription list and avoid error conditions when it sends out updated Relevant Data Type lists.

F. Identifying Types of Data

Data type identification should not be done using any sort of C++ or .NET class or struct names since that would mean that the Middle Man would need to have explicit knowledge of these types, and thus would have build and/or run-time dependencies on consumer and producer applications. This would make it impossible to add future producer or consumer applications referencing different types of data without modifying the Middle Man component. To make it easier to extend the system, some sort of key value should be used to identify the type of data being published.

It is important that key values for different types of data be unique in order for consumer applications to run properly. While simple integer values could be used for keys in a private implementation, the use of identifiers that are guaranteed to be unique across organizations would be preferred as that would allow a test organization to make use of consumer or producer applications from instrument, test system, or application development environment vendors, in addition to their own, without conflict. As the name implies, the GUID (*G*lobally *U*nique *ID*entifier) or UUID (*U*niversally *U*nique *ID*entifier) would be a suitable candidate [4].

In addition to the type of data, data notifications from producers should include some sort of context information to sub-class or characterize the data being produced. Consumers can key off this context information to determine whether the data being provided is relevant.

G. Adapters

When a producer sends data to the Middle Man, it does so in a generic manner as an array of bytes, and identifies the type of data by including a key. The conversion of the data to be logged into an array of bytes is termed "serialization".

Similarly, when a consumer receives data, it does so in the form of an array of bytes delivered from a particular source via a callback function or .NET event invocation. Using the supplied key, the consumer can then "deserialize", or regenerate, the original data.

As a result, it is necessary that there be some sort of agreement between the producer and consumer on what the key value represents and how the data is to be serialized/deserialized. A typical way to create such an agreement is to introduce a software layer to handle the serialization/deserialization processes. The layer would define the key value, and functions to serialize and deserialize the data. This layer is referred to in this paper as a data type *adapter*.

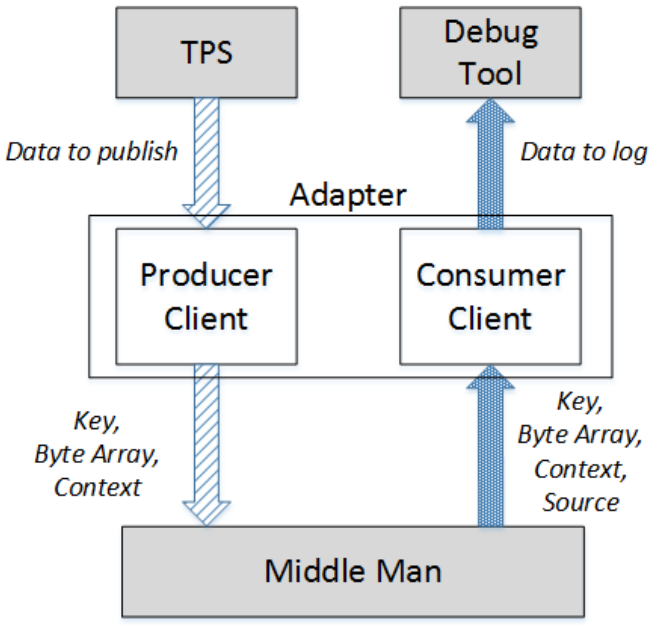

Fig. 2. Conceptual View of Adapters

An adapter can be regarded as an abstract entity that defines a contract between producers and consumers. Adapters could be realized in a single software library shared between producers and consumers alike, but more likely they will be implemented across a set of software libraries in a native language like C or C++, in .NET, or using a mixture of both, with producers and consumers using the libraries separately.

A description of the meaning of the context information being provided with the data is included in the adapter's functionality.

III. Example Applications

A. Function Call Logging

A common debugging tool of interest to TPS developers and integrators is one that traces a series of calls to instrument driver functions. Such a tool, usually developed by an instrument vendor as part of the software suite included with their instrument drivers, particularly one that lists the values of function parameters and return values, can be used to quickly identify tasks that are not configured correctly or that occur out of order. Unfortunately, these tools are typically proprietary, and do not provide any information on function calls made to software components other than the vendor-provided drivers.

The architecture proposed in this paper yields a method for logging function calls that is common across multiple instrument vendors, providing a way for a TPS developer to identify issues resulting from the use of multiple instruments simultaneously. At the very least, it would mean that a TPS developer would only need a single tool to generate these logs, rather than multiple, proprietary ones that operate differently.

In many cases today, test system software provides wrapper layers to remove explicit driver calls from TPSs. In those cases, test organizations have had to develop their own tools for logging calls made to these wrapper functions. Another advantage of the approach described here is that such wrapper function calls could also be logged using the same techniques employed by the instrument vendors. The idea could even be extended to the TPS code itself, resulting in a complete call tree for all of the activity during the execution of the TPS. This capability would only be realized, of course, if the Function Call Logging adapter were publicly available and used both within a test organization and by test instrument/system vendors.

A proposed Function Call Logging adapter would:

- Define a key for the Function Call Logging data type.
- Expose functions for producers to log function calls and returns from functions, including parameter names, types, and values.
- Expose functions for consumers to convey (or remove) interest in receiving function call/return data. These functions would include a way to target specific APIs and producer programs.
- Provide a mechanism by which a consumer receives notifications of new function calls.
- Define how the context information provided by a producer with the data notifications would be used to identify the API to which a function belongs.

The code to add function call logging support to an API would tend to be boilerplate in nature, and straightforward to include in the API. Consider the hypothetical C API function shown below:

```
ViStatus _VI_FUNC Foo (
    ViSession vi,
    ViInt32 param1,
    ViInt32 size,
    ViInt16 data[])
{
    BEGIN_LOG_FUNCTION(“Foo”,
           P_ViSession, "vi", vi,
           P_ViInt32, "param1", param1,
           P_ViInt32, "size", size,
           P_LAST);

    // Actual implementation elided…

    END_LOG_FUNCTION("Foo",
           P_A_ViInt16, "data[]",size, data,
           P_ViStatus, "status", status,
           P_LAST);

    return status;
}
```

The extra logging related code essentially wraps the actual function implementation. In this example C function implementation, the symbols BEGIN_LOG_FUNCTION and END_LOG_FUNCTION are C macros that resolve to calls exposed by the adapter to send data to the Middle Man. The P_* symbols are other C macros or C enumeration values that identify the datatypes of the parameters and return values. Implemented within the adapter, and so hidden from the API developer, are the key for the “Function Call” type of data and details of the serialization process and the interface with the Middle Man.

When an API function call is published and distributed to an interested consumer, the adapter on the consumer side invokes a previously registered callback function (or event if the consumer is a .NET application), with parameters populated with the specifics of the producer function call. In response, the consumer application can display information about the call in a window, record the information in a file, or whatever else it might deem appropriate.

Should there need to be a change in how the function call data is processed (for example, converting from a file-based to a database-based archival method, or modifying how the data is presented on the screen), such changes can be made solely in the consumer applications with no impact on the TPS code.

B. Test Result Data Logging

Other common tasks relevant to TPS qualification include collecting test results, analyzing the margin of test limits, and measuring test stability. The creation of a Test Results adapter would facilitate all three of these tasks.

This adapter would have characteristics similar to those of the Function Call Logging adapter. It would:

- Define a key for the Test Result Data Logging data type.
- Expose functions for producers to log test statements, including measured values, test types, test limits, and an ID to uniquely identify the test.
- Expose further functions for producers to convey information that identifies the name and, if applicable, versioning data for the TPS, UUT, and the test system.
- Expose functions for consumers to convey (or remove) interest in receiving test results data. These functions would include a way to target specific producer programs.
- Provide a mechanism by which a consumer receives notifications of new test statements.
- Define how the context information provided by a producer with the data notifications would be used to identify the type of test being performed.

Unlike with function call logging, which may not be apparent in the TPS source code if the logging is only supported in wrapper or instrument driver functions, test result data logging would need to be coded into the TPS directly, since the TPS contains the actual test code. In this case, the TPS makes direct calls to the functions exposed by the Test Results adapter. The example below demonstrates how this

might be accomplished in a C-based TPS making a voltage test using a digital multimeter:

```
ViReal64 measurement;
ViReal64 minLimit = …;
ViReal64 maxLimit = …;
ViBoolean testPassed;
ViStatus status;
ViInt32 testID = …;

// Setup elided…

// Read the voltage reported by the DMM
status = terAI7_DMM_Read(
     vi,
     1,
     TERAI7_VAL_SS_NOT_TRIGGERED,
     1.0,
     &measurement);

// Record the test result
testPassed = LOG_TEST_RESULT(
     measurement,
     minLimit,
     maxLimit,
     LOG_TEST_TYPE_ANALOG_VOLTAGE,
     testID);
```

The symbol LOG_TEST_RESULT is another C macro that resolves to a call to an adapter function that performs a simple floating point comparison against min and max test limits. The symbol LOG_TEST_TYPE_ANALOG_VOLTAGE identifies the type of the test, and the unique test identifier is represented here by a variable.

Here, the extra logging related code is not really extra at all, as it contains the comparison of the measurement value against test limits, which is code that the TPS needs to execute anyhow.

A consumer application receives test results data using the same callback function/event invocation method that was described for the Function Call Logging adapter.

One of the advantages of the logging system described in this paper is that you can associate multiple consumers for the same published data. For example, we could create a consumer application that logs test result data to a database (for manufacturing process control purposes for example), while a separate application could be used to graphically depict test measurements against limits (for the benefit of the TPS developer). These applications can also be running simultaneously. Again, changes to the existing consumer applications, or the creation of new ones, would not impact the TPS, assuming that the data already provided in the TPS code is sufficient for the new applications.

C. *Creating Custom Adapters*

The previous adapters are general purpose in nature, and would preferably be provided by the test system vendor in support of TPS or tool development. Tools development for more specialized or in-house test instruments, or the need to carefully deal with sensitive data while still supporting TPS debugging and qualification, may require the creation of adapters that are private to a particular organization.

Creating a custom adapter is relatively straightforward. An outline of the steps involved is show below:

1. Create a GUID to define the nature of the data to be published.
2. Decide on what data to publish.
3. Create a producer client adapter to handle the serialization of the data to be sent to the Middle Man, and expose a set of functions for the TPS or other producer application to call to publish data.
4. Create native or .NET consumer client adapters to handle deserializing the data delivered by the Middle Man, and notifying the consumer application(s) of the published data.

To facilitate steps 3 and 4, the provider of the Middle Man component should also provide general purpose software libraries, against which the adapter components would be built, that handle the direct communication between producer or consumer application processes and the Middle Man server.

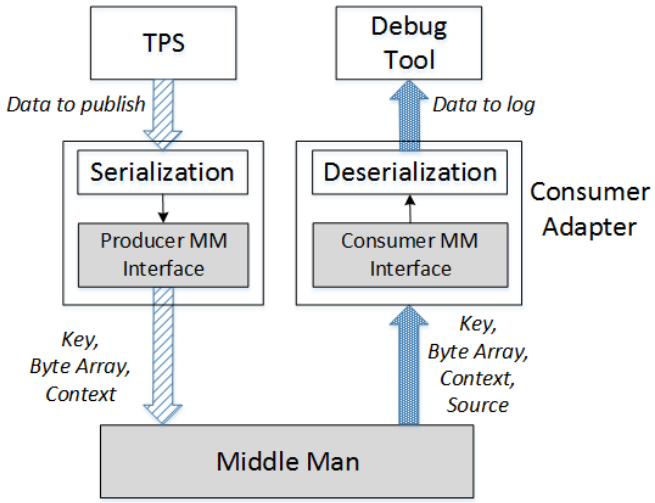

Fig. 3. Custom Adapter Design (Shaded components part of the Middle Man software suite)

IV. Conclusion

The test system software architecture enhancements described in this paper represent an attempt to simplify TPS development by off-loading certain common tasks that do not directly address the test requirements of the UUT.

Besides easing the burden of TPS development, the resulting software architecture provides a means to more easily incorporate new and enhanced debugging tools and capabilities, even on deployed test systems that lack installed application development environments.

The revised software architecture enables test organizations to complement the tools provided by instrument and test system vendors by creating their own set of dedicated tools.

These tools can be easily integrated across a suite of TPSs in a way that supports future enhancements of the tools without affecting the TPSs themselves.

V. References

[1] Publish–subscribe pattern. (2016, June 11). In Wikipedia, The Free Encyclopedia. Retrieved 18:26, July 22, 2016, from https://en.wikipedia.org/w/index.php?title=Publish%E2%80%93subscribe_pattern&oldid=724774008.

[2] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Reading, MA: Addison-Wesley.

[3] Gamma, E., Helm, R., Johnson, R., & Vlissides, J., 293.

[4] Leach, P., Mealling, M., & Salz, R. (2005, July), *A Universally Unique Identifier (UUID) URN Namespace*, Retrieved from http://www.ietf.org/rfc/rfc4122.txt.