INTEGRATING A GRAPHICAL INSTRUMENT DEVELOPMENT ENVIRONMENT WITH AN EXISTING APPLICATIONS DEVELOPMENT ENVIRONMENT AND TEST EXECUTIVE

Yönet A. Eracar, Ph.D.
Teradyne, Inc.
600 Riverpark Drive
N. Reading, Mass. 01864
978-370-1608
yonet.eracar@teradyne.com

Abstract – This paper describes a graphical instrument-specific test development environment that generates standard-language test code or a test data file that can be used by virtually any Applications Development Environment (ADE). Examples demonstrate where code is appropriate, where pure data is appropriate, and how each is used by the various available commercial ADE and test executive environments.

INTRODUCTION

A good graphical instrument-specific test development environment helps a test engineer go through the learning curve of programming a test instrument and shortens the time to develop a test. If there is a graphical test development environment for the test instrument, the test engineer starts test development by experimenting with the environment and creating a simple test case to become familiar with both the environment and the test instrument. Through multiple revisions, the test engineer extends the functionality of the test and debugs the test program on the test instrument--all within the test development environment.

In most test development environments, the test data is persisted in a proprietary format that can be processed only by the development environment that generated the test data. Some instrument vendors may provide a driver Application Programming Interface (API) to execute the generated test data and to allow the user to add more code to integrate the test with other programs. If the test instrument does not come with a development environment, then the test engineer develops the test program manually using the test instrument's driver API.

Next, the test engineer integrates the test data or the test program with the rest of the Test Program Set (TPS). Usually, the current test that the test engineer is working on is only one of the many tests that make up a complex TPS. These tests can be executed independently or collaboratively within the TPS.

If all the tests in a TPS are generated by the same development environment, integration of the tests within the same development environment is possible. But, if the test types (i.e. digital, analog, or mixed) are different or if the tests are targeting different instruments from different vendors, integration of these tests can be very challenging.

Each generated test program can be a standalone program, which can be executed independently from others in the TPS, or can be executed from a test executive that manages the overall TPS. Every test executive has its advantages and disadvantages. While a test executive needs to be flexible to integrate various test programs and tools, the flexibility of the test executive is inversely proportional to its simplicity of use.

This paper addresses two problems: the need to generate test programs efficiently and the challenge of integrating multiple test programs in a complex TPS. First, the paper lists the constraints for test program development in general. Next, the paper discusses the advantages and disadvantages of two test programming approaches--standard-language test code generation and test data generation. These two approaches are then compared, and a hybrid approach is proposed as the third alternative. Examples from an existing graphical test development environment are used to support the discussion above.

CONSTRAINTS FOR TEST PROGRAM DEVELOPMENT

The following is a list of constraints that a test engineer should consider when developing a test program. The cost of TPS development is the most important constraint for test engineers. However, this paper will focus only the technical constraints listed below. This is not a complete list and does not follow an order of importance.

- Execution time: If the test program requires operator interaction and batch execution is not possible, test programs with long execution times can be a problem in terms of the throughput in a factory environment.
- Load time: Time required to load the test program in the memory of the host processor for execution can be a concern, if multiple test programs are being executed back-to-back and the timing between individual tests is critical.
- File size: With the recent advances in the storage media technology, the size of the test program is less of an issue. On the other hand, if the system contains multiple test programs, large file sizes for the test programs can create problems in terms of available disk space on the target tester system.
- Flexibility for customization: It is seldom that an automatically generated test program satisfies all the test requirements. The test engineer may need to customize parts of the program and to do additional integration work, such as the synchronization of the current instrument with the rest of the instruments in the tester system.
- Required ADE / tool on the target system: If the test development environment generates test code, then the target tester system needs an ADE to build and debug the test program. This not only means additional cost, but also extra programming experience for the test engineer. If the test development environment generates test data, the target tester system needs a run-time engine to execute the test data.
- Portability: Usually the test development environment dictates the platform and the Operating System for test program development. However, the platform for the test execution can be different from the one for test development. If the test program needs to be maintained on multiple platforms, it needs to be portable to avoid extra development.
- Training: Power users of the test instruments perform intense customization of the automatically generated test programs and eventually prefer manual programming rather than automatic generation. For such users, the test development environment works as a valuable training tool for advance programming.

TWO APPROACHES TO TEST PROGRAM DEVELOPMENT

The following sections discuss the advantages and disadvantages of two test programming approaches -- generating standard-language test code and generating test data – in light of these constraints.

1) Generating Standard-language Test Code

The execution time of test code depends on the implementation of the test instrument driver API, the communication mechanism between the host processor and the test instrument, and the efficiency of the generated test code. Given today’s very fast processors, the real bottleneck is the data and message transfer between the host processor and the test instrument rather than the execution of the test program making calls to the driver API on the host. An instrument driver that is designed and implemented to eliminate excessive data transfers (i.e. transferring data in large blocks vs. individual register access) during execution improves the execution time.

The file size of the generated test code depends on the size of the test (i.e. stimulus, response, and any diagnostics actions) as well as on the abstraction layer of the driver API and the efficiency of the code generator. Many ADEs have undocumented internal limitations on the size of the code file or the size of individual functions in a code file such that large test code files or functions may fail to compile and build under these ADEs. A robust code generator should detect such occurrences and generate modular (i.e. smaller function size, smaller file size) test code, even it can’t reduce the overall size of the test program.

The generated test code is flexible in that the test engineer can insert code pieces that are not supported by the graphical test development environment or the target test instrument. These code pieces are usually necessary for various setup tasks and synchronization with other test instruments. However, if the test program needs to be supported at multiple platforms, a standard programming language (i.e. C, C++) must be used as the generated code. The use of a standard programming language also lets the test engineer pick his favorite ADE. With a non-standard programming language, the test engineer’s options for ADEs are limited.

If the test program needs to be portable, then the use of a standard-language test code (i.e. C, C++, Java) supported by multiple platforms and operating systems is necessary for test programming.

A major benefit of automatic code generation is that it serves as a training tool for the test engineer to learn how to manually program in the driver API of the tester instrument.

2) Generating Test Data

Once the test data is loaded in the memory, the critical factor for the execution time of the test program is the data and message transfer between the host processor and the test instrument; this is similar to the situation with test programs using automatically generated test code.

However, reading the test data from a file and representing the data in the memory can increase the overall execution time and the used memory significantly. For example, Extensible Markup Language (XML) [2] is a widely used format for storing and sharing test data among test development environments. Document Object Model (DOM) and Simple API for XML (SAX) are the primary paradigms used for processing data in XML format. DOM parsers generally require the entire file to be loaded into the memory and constructed as a tree of objects before access is allowed. A SAX parser accepts XML information as a single stream of data. The data stream is unidirectional and the SAX paradigm results in a faster XML processing and smaller memory usage than DOM. However, with SAX parsers, it is difficult to retrieve information at random from the XML file.

Data file size can be optimized through various binary encoding if disk space and the media for TPS transportation is an issue.

When test data is generated, the tester PC does not require an ADE to exist on the tester system to execute the test data as long as the instrument provides a run-time execution engine for the tester system. Similarly, the portability of test data depends on the availability of a run-time execution engine for the platform and the operating system of the tester system.

Reviewing test data file formats is not a good place to start for training. Data file formats provided by the vendors tend to change in time unless the format is restricted by an industry standard.

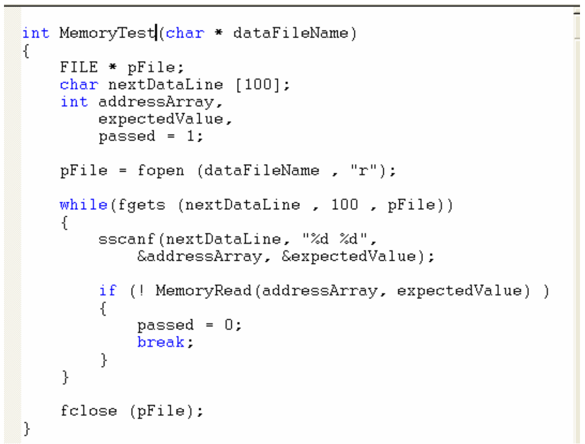
Generating Standard-language Test Code vs. Generating Test Data

Table 1 summarizes the comparison between generating standard-language test code and generating test data in terms of the test programming constraints. As can be seen in the table, not one of the approaches is a clear winner. The constraints of the TPS and the target test system dictate the decision for the test programming approach.

Table 1. Comparison of generating standard-language test code vs. generating test data

Constraint	Test Code	Test Data
Execution time	depends	
Load time	depends	
File size		√
Flexibility for customization	√	
Required ADE / tool on the target system		√
Portability	depends	
Training	√	

In a case where a single approach does not satisfy all the constraints, the test engineer may desire to use a hybrid approach to utilize the flexibility of test code and the compactness of test data. Figure 1 shows a simple memory test function written in C programming language. The function accepts a test data file and compares the test data stored in the data file against the actual data read from the memory.

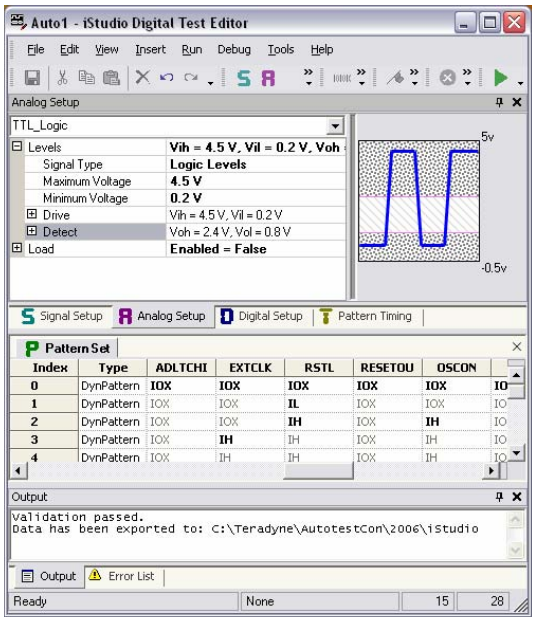


```
int MemoryTest(char * dataFileName)
{
    FILE * pFile;
    char nextDataLine [100];
    int addressArray,
        expectedValue,
        passed = 1;

    pFile = fopen (dataFileName , "r");

    while(fgets (nextDataLine , 100 , pFile))
    {
        sscanf(nextDataLine, "%d %d",
            &addressArray, &expectedValue);

        if (! MemoryRead(addressArray, expectedValue) )
        {
            passed = 0;
            break;
        }
    }

    fclose (pFile);
}
```

Figure 1. Hybrid approach: reading test data from test code

TEST CASE: DIGITAL TEST EDITOR

The Digital Test Editor [1] associated with Teradyne's new generation of digital test instruments is designed to address the test development constraints described above. The Digital Test Editor, shown in Figure 2, is an integrated graphical toolset that uses a UUT-centric approach for test development, debugging, and code generation of digital functional test. It hides the instrument-specific details from the test engineer through a UUT-centric abstraction, minimizes programming through automatic code and test data generation, promotes test data re-use through built-in data sharing support, and simplifies TPS integration effort through integrated tools for development, validation, and debugging.



TTL_Logic

Levels	Vih = 4.5 V, Vil = 0.2 V, Voh
Signal Type	Logic Levels
Maximum Voltage	4.5 V
Minimum Voltage	0.2 V
Drive	Vih = 4.5 V, Vil = 0.2 V
Detect	Voh = 2.4 V, Vol = 0.8 V
Load	Enabled = False

Index	Type	ADLTCHI	EXTCLK	RSTL	RESETOU	OSCON	
0	DynPattern	IOX	IOX	IOX	IOX	IOX	IO
1	DynPattern	IOX	IOX	IL	IOX	IOX	IO
2	DynPattern	IOX	IOX	IH	IOX	IH	IO
3	DynPattern	IOX	IH	IH	IOX	IH	IO
4	DynPattern	IOX	IH	IH	IOX	IH	IO

```
Validation passed.
Data has been exported to: C:\Teradyne\AutotestCon\2006\iStudio
```

Ready | None | 15 | 28

Figure 2. Generating test data using the Digital Test Editor

The Digital Test Editor persists the test data in XML format internally. To execute the test, the Digital Test Editor first validates the test data. If the validation passes, the editor generates test code in C and then builds it in the memory to be a Dynamic Link Library (DLL). Finally, the editor executes the DLL by calling a predefined entry function to the DLL.

When generating test data, the Digital Test Editor follows the same steps as the execution. Only at the last step, instead of executing the DLL, it compresses through a zip utility and saves the compressed DLL in the memory to a target directory. This ensures that the results obtained from executing the digital test from the Digital Test Editor are the same as the ones from executing the test data through a combination of an external ADE and a test executive. On the downside, the execution time gets a small hit for decompressing the test data before execution. Figure 3 shows the data flow for code generation and execution in the Digital Test Editor.

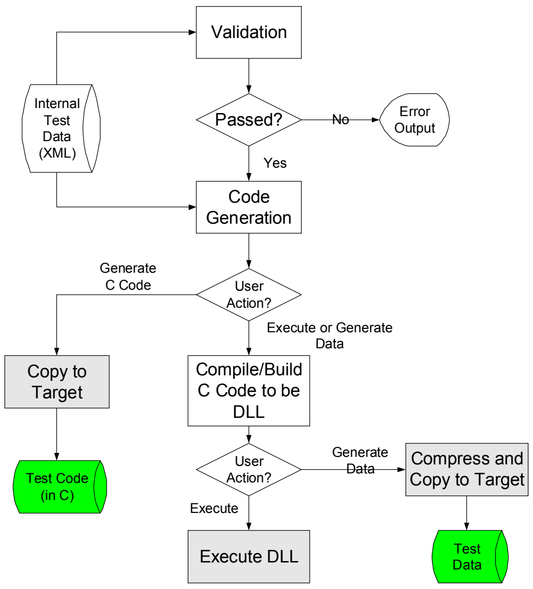
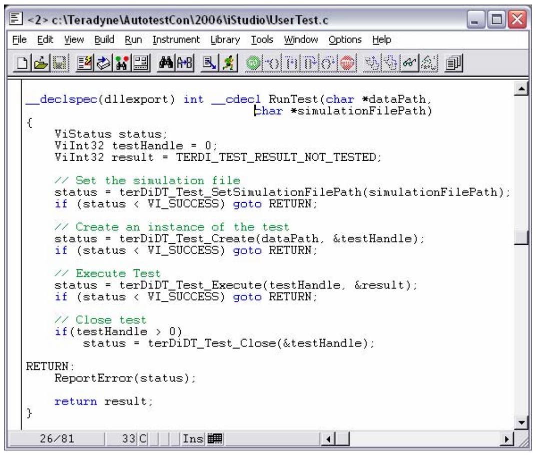


Figure 3. Data flow for code generation and execution in the Digital Test Editor

In addition to the test data, the Digital Test Editor generates a template C file and a header file (see Figure 4) that contain the necessary functions to load and execute the generated digital test data file. The functions in the template C file include the necessary modifiers to be exported from a DLL. DLLs are widely accepted mechanisms for integrating test programs with commercial test executives (i.e. TestStand, TestStudio).


<2> c:\Teradyne\AutotestCon\2006\iStudio\UserTest.c
File Edit View Build Run Instrument Library Tools Window Options Help
26/81 33 C Ins

```
__declspec(dllexport) int __cdecl RunTest(char *dataPath,
                                          char *simulationFilePath)
{
    ViStatus status;
    ViInt32 testHandle = 0;
    ViInt32 result = TERDI_TEST_RESULT_NOT_TESTED;

    // Set the simulation file
    status = terDiDT_Test_SetSimulationFilePath(simulationFilePath);
    if (status < VI_SUCCESS) goto RETURN;

    // Create an instance of the test
    status = terDiDT_Test_Create(dataPath, &testHandle);
    if (status < VI_SUCCESS) goto RETURN;

    // Execute Test
    status = terDiDT_Test_Execute(testHandle, &result);
    if (status < VI_SUCCESS) goto RETURN;

    // Close test
    if(testHandle > 0)
        status = terDiDT_Test_Close(&testHandle);

RETURN:
    ReportError(status);

    return result;
}
```

Figure 4. Execution of the generated test data

The Digital Test Editor generates C code as well as C# code to talk to the driver API of the Teradyne digital test instruments. This is the same generated code used at the execution of the digital test in the Digital Test Editor.

As a test case, a simple memory (ROM) test of fifteen thousand test vectors was developed using the Digital Test Editor. There were on the average 10 pin changes per test vector. The size of the internal XML test vector data file was 11.8 MB. The size of the generated test code in C was 11.4 MB (241 KLOC), and the test vectors were grouped in four separate functions by the code generator to reduce the size (number of function calls) of the individual functions under a critical value that causes commercial compilers, like Microsoft Visual C and NI LabWindows/CVI, to fail to build.

The size of the DLL that was built from the generated test code using Microsoft Visual C version 6.0 was 2.56 MB. The size of the generated test data was 800 KB. The results obtained from the execution of the generated test code and the test data were identical, since the same underlying code was running. The test data had significantly smaller footprint as expected, but took longer to execute due to the decompressing took place at the beginning of the execution.

CONCLUSIONS

In this paper, first, challenges in test program generation were introduced. Next, a set of user and environment constraints for test program development was described. Two test development approaches (automatic standard-language test code generation and automatic test data generation) were compared in terms of these constraints. Advantages and disadvantage of both approaches were discussed in detail, and a hybrid approach was proposed. Finally, examples from the Digital Test Editor, a graphical instrument-specific test development environment, were used to support the above discussion.

REFERENCES

[1] Eracar, Y.A. and Lopes, T., "A UUT-centric test specification simplifies digital functional test development", IEEE AUTOTESTCON'2005, Orlando, FL, USA.

[2] XML (Extensible Markup Language) 1.0. World Wide Web Consortium Recommendation 4 February 2004 [cited 2000-05-05]. Available from World Wide Web: <http://www.w3.org/TR/2004/REC-xml-20040204/>.