AUTOMATED TESTING OF OPC SERVERS

B. Farnham, CERN, Geneva, Switzerland

Abstract

CERN relies on OPC Server implementations from 3rd party device vendors to provide a software interface to their respective hardware. Each time a vendor releases a new OPC Server version it is regression tested internally to verify that existing functionality has not been inadvertently broken during the process of adding new features. In addition bugs and problems must be communicated to the vendors in a reliable and portable way. This presentation covers the automated test approach used at CERN to cover both cases: Scripts are written in a domain specific language specifically created for describing OPC tests and executed by a custom software engine driving the OPC Server implementation.

INTRODUCTION

The architecture of many of CERN's Detector Control Systems (DCS) is, to a large degree, stable. Users of the DCSs are currently able to sufficiently control and monitor their systems to successfully create and gather physics data. Nothing is perfect however. Controllers and administrators have working systems, but there is scope for improvement: fixing bugs, adding features, improving response times to commands, reducing feedback latency etc.

The OPC Test Script Runner is a tool, developed at CERN, for writing, running and recording the results of scripted tests which engage DCS components at the level of OPC [1] (a standard providing a uniform means of controlling and monitoring devices). The tool aims to address two concerns regarding the stable DCS environment described above:

- Changes carry an inherent risk of regression: Changing X might accidentally break Y (and Z). A DCS user may request a new feature to a device, exposed via its OPC Server. The user wants assurance that, after making changes, the critical features on which the DCS relies function at least as well as they did before. Additionally the user wants assurance that the new feature functions as per specification and will continue to do so over time as subsequent modifications are made.
- It can be hard to accurately convey a bug to a vendor, and hard for a vendor to recreate a bug for diagnosis. Consider a subtle bug in an OPC Server, occurring only under some long and complicated sequence of client operations: Communicating the detail of this bug to the vendor involves the user translating this sequence into a textual description, transmitting it to the vendor who de-translates that description back into OPC client actions to replay to an OPC Server in an attempt to reconstruct the original bug. Written language can be an awkward medium for unambiguously recording a series of complex actions. Misinterpretation of one action in the sequence may result in the vendor being unable to recreate the problem for diagnosis.

OPC test scripts describe functionality and perform runtime verification. Test scripts, designed to capture behaviour of critical features, are run against new releases to check whether the modifications damaged existing functionality. Furthermore, as new features are released, their functionality is described and verified through test scripts in order to test the feature at point of release. These test scripts are added to the catalogue of regression tests to verify that functionality subsequent releases.

A similar approach applies to bug reporting - in effect a test script is written which fails, the failure highlighting the ill effects of the bug. The script is passed to the manufacturer who runs it to observe the bug first hand. The bug is fixed once the script passes.

THE TEST SCRIPT RUNNER STRUCTURE

The OPC Script Runner contains an OPC client. At their most basic level, OPC test scripts describe sequences of OPC interactions which the client carries out with the server at runtime. A test script which simply specifies a sequence of client interactions is not sufficient however, in order to ascertain whether the interactions had the desired effect, i.e. whether the test script passes or fails. There is an additional tie between the test script and script runner, namely assertions, a notion familiar to unit testing practitioners. Assertions in the script instruct the assertion handling module of the script runner to set watchpoints with specific criteria for success, failure to meet these criteria results in the assertion failing, and thus the test script failing. The test script runner records and displays assertions results. Assertions are described later in more detail.

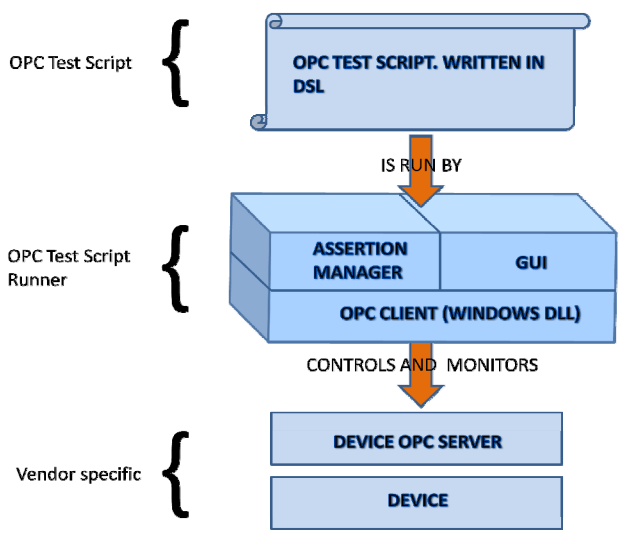

Figure 1: OPC Test Script Runner Components.

The OPC Client part of the test script runner is a MSWindows dll, written in C++, the upper part of the test script runner, handling script execution, assertion management and the graphical user interface is written in Groovy, a dynamic language for the Java Virtual machine. Interactions between the higher level processing on the JVM and the lower level OPC client are handled by open source Java Native Access (JNA).

A dynamic language such as Groovy was an important choice for the upper portion of the test script runner in order to aid defining the Domain Specific Language [2] (DSL) in which the test scripts are written. DSLs are 'small languages', designed for a particular field and consisting of nouns and verbs specific to that field. The field in this case is OPC testing, OPC nouns from the DSL include familiar OPC terms such as 'group' and 'item' and verbs pertaining to OPC nouns like an item's 'asyncWrite' or a group's 'destroy' plus unit test style verbs such as 'assertTrue' or 'assertNotEqual'. The DSL for the test scripts has been designed to promote readability, the idea is that 3rd party users such as an OPC Server vendor should be able to read a testscript and understand it in terms of the OPC interactions.

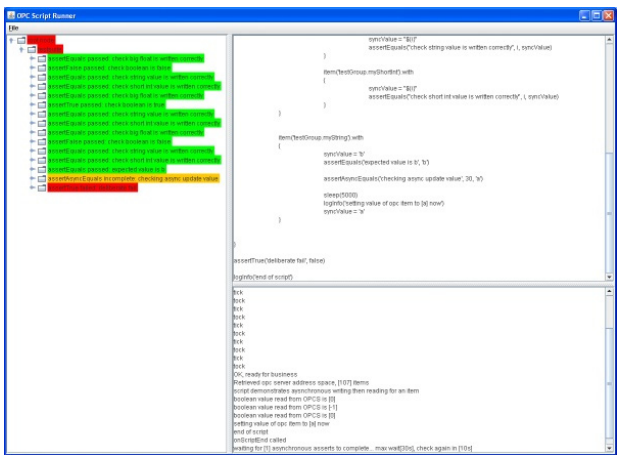

Figure 2: OPC Test Script Runner GUI layout.

The left panel displays a tree of the script's assertions colour coded on their pass/pending/fail states, the tree is dynamically built at runtime as the assertions are made and the assertion criteria met or otherwise. The top right panel displays the script text (read only) and the bottom right panel displays a log window which updates with system messages and user defined log messages written in to the script. From the menu users can open and run scripts and export script assertion results in Junit style XML format.

ASSERTIONS

Assertions are the mechanism for defining pass/fail criteria in test scripts. Two types of assertions are available:

Synchronous Assertions

These are the simplest case. A script can make assertions about an OPC item's immediate value - that it is equal to x, or not equal to y for example. Synchronous assertions pass or fail immediately. Script excerpts detailing synchronous assertions include:

- *item.assertEquals("Verifying the value of an integer item", '50')*
- *item.assertTrue("Verifying the value of a boolean item")*
- *item.assertNotEquals("Verifying the value of a string item", 'ERROR')*

Asynchronous Assertions

OPC interactions can be asynchronous in nature. For example OPC has the concept of groups: A client requests that a server builds a group and adds items to it and that the server informs the client of changes to group items at a rate not greater than some client specified frequency. From the perspective of the test script, commands can be delivered through the client but the effects only seen some time later. This use-case is handled by the asynchronous class of assertions. These assertions provide a required criteria for success (as with their sycnhronous siblings) however they also specify a time limit within which the criteria must be met. Asynchronous assertion excerpts from scripts include:

- *item.assertAsyncEquals("Waiting a maximum of 2.5 seconds for channel status to be stable:on", 2500, 1)*

Asynchronous assertions have 3 states: Pass, fail and pending (while the criteria is not met but the time limit not yet reached). This asynchronous class of assertions allow for performance test scripts. For example an asynchronous assert could be scripted immediately after multiple commands have been sent to verify whether a system remains responsive (to within the timeframe specified by the assertion) following a command flood.

A SIMPLE EXAMPLE SCRIPT AND ITS ANATOMY

The following is an OPC test script used for verifying the functionality and response time for turning channels on for a CAEN industrial power supply. Note some parenthesis and user messages have been omitted for brevity. As previously stated, the DSL in which test scripts are written is intended to be readable by 3rd parties such that they can understand what the test does and the assertions the tests make. The DSL includes a regular expression like syntax (*) to denote collections of OPC items with matching addresses.

```
init('', 'CAEN.HVOPCServer')

group('setup.software.and.hardware.chain').with
   item('**.ConnStatus').assertEquals('Ok')
   item('**.OPCServerEventMode').assertFalse()

group('set.initial.device.state').with
  items('**.Board*.Chan*.Pw').each
```

```
        it.syncValue = 'false'
items(**.Board*.Chan*.Status').each
        it.assertEquals('0')

group('main.body').with
   // set up asynchronous assertions
   items(**.Board*.Chan*.Status').each
        it.assertAsyncEquals(10000, '1')
   // turn the channels on
   items(**.Board*.Chan*.Pw').each
        it.syncValue = 'true'

// pausing for 11s to allow asynchronous criteria
sleep(11000)

group('wrap.up').with
   items(**.Board*.Chan*.Pw').each
        it.syncValue='false'
logInfo('end of script')
```

The test script runner dictates no special structure for the test scripts it executes, however experience has shown the structure followed above to be an effective skeleton:

1. Verify the software and hardware chain: Assert that the OPC Server and underlying device are present and the basic state is suitable. For example is there an OPC item which can be used to assert that a device is connected to the OPC Server and is powered on.
2. Set and verify the pre-test device state: Send commands to the underlying device to set the initial state and assert that the required initial state has been achieved.
3. The main body: Commands are issued and assertions made about the effects of those commands.
4. Wrap up: Send commands to set any post test state and assert that the post test state is achieved

USE CASES

OPC test scripts are simplest when a repeatable sequence of events and their corresponding effects can be defined. For regression test type scripts this is almost always the case: A known sequence of client/server interactions is expected to have a known set of required outcomes. So long as these outcomes are visible to the script runner via OPC then they can be verified in the form of assertions. A slightly more complicated type of test involves a known end effect (observable via OPC) but with an undefined set of events leading up to it - often this is the situation regarding subtle bugs: For example after many hours of continuous operation an OPC item suddenly ceases to update. Again, so long as the effect is observable via OPC the behaviour ought to be able to be captured using an OPC test script. Test scripts can request random numbers to introduce a stochastic element into interactions.

Some examples cases where the OPC Test Script Runner has already been employed:

1. A device vendor must switch their OPC Server implementation from an outdated OPC toolkit library supplier to another (in order to provide Windows 7 support). This change is not insignificant. A catalogue of test scripts has been written against the previous OPC Server version on XP. These scripts are being run against the new OPC Server version on windows 7 to check for bugs and deterioration in functionality.
2. The OPC test script runner has been used to successfully narrow down operations causing a memory leak in an OPC Server from an industrial power supply vendor. Memory usage is not available via OPC so this was monitored using an external tool: Windows XP Perfmon. Multiple scripts were written, each script focusing on a different type of client/server interaction (different types of reads and writes for example), and each script run for some set period whilst monitoring the memory usage. Certain scripts appear to cause the OPC Server memory usage to grow more quickly than others, providing the vendor with empirical information as to the problematic client/server interactions. The scripts (plus runner) have been passed to the vendor.

FURTHER WORK

Currently scripts are a single continuous sequence. The ability to define methods (parameterized repeat blocks of script) would allow for more concise and intuitively readable scripts.

The current implementation of the OPC Test Script Runner supports only the OPC Classic specification released in 1996 and based on deprecated Microsoft COM/DCOM technology. The OPC Foundation released a next generation OPC specification called OPC-UA [3] which looks set to gradually replace OPC Classic over time. The OPC Test Script Runner client will be updated to be OPC-UA compliant and the test script DSL extended to include OPC-UA operations.

The test scripts runs without human intervention, however, each script must be loaded and started by hand. It would be more efficient to instruct the script runner to run a catalogue of tests sequentially. For example run all scripts in a given directory.

CONCLUSION

Automated testing is an established practice in software development, providing increased assurance against the regression of existing functionality as current OPC based DCS technology evolves. Furthermore, the ability to provide vendors with runnable instances of bugs and problems can provide significant efficiency gains over writing traditional bug reports. The OPC test script runner provides a means to bring both of these benefits to bear on the field of OPC components.

REFERENCES

[1] The OPC-DA Specification.
www.opcfoundation.org

[2] M. Fowler and R. Parsons “Domain-Specific Languages”. Addison Wesley 2011.

[3] The OPC-UA Specification
www.opc-foundation.org/ua