Universal Functional Test (UFT):
a Cost-Effective Manufacturing Test Development Software Platform.

Chris Plumlee, MSEE
CAE Integration, LLC
President

Contents

High Cost of Functional Test in Electronics Manufacturing 1
Reuse and Scalability Before the First Line of Code. 3
UFT Product Test State Machine 4
Scaling up for High Volume 5
Underlying Mechanics 6
Compactness: 6
Modularity: 7
Auto-Recognition of UUT: 8
Disconnect Detection: 8
Logging: 8
For More Information 9

High Cost of Functional Test in Electronics Manufacturing

Functional test in the manufacturing process requires custom development due to the unique nature of each product. The manufacturing mindset is to minimize or eliminate custom work, as it is neither inexpensive nor easy to scale. There are three general approaches for development of manufacturing tests, each with its own pressure against reusability:

1. In-house test engineers at the end of the New Product Introduction (NPI) cycle are often under pressure to win back delays that have accumulated all along the development process.
2. Outside test resources are generally constrained by the project bid, which is optimized for the product in question rather than reusability.
3. Off-the-shelf functional test solutions are quick to deploy but generally lack key features which much be added later by returning to approach #1 or #2..

Universal Functional Test (UFT):

a Cost-Effective Manufacturing Test Development Software Platform.

UFT is designed from the ground up to automate the functional test processes to the greatest *reasonable* extent in order to:

1. Optimize use of the test operator's time (increasing productivity and decreasing cost).
2. Reduce operator error (increasing productivity and avoiding false-passes, which can result in Out-Of-Box Failure or OOBF)

While it is possible through complicated fixturing to "fully" automate a test process, many automated steps diverge into corner-cases which become increasingly expensive to automate for decreasing benefit. While clamshell fixtures necessitate full automation due to their restriction of technician access, open-frame fixtures allow access to the product. This fixture design approach opens up choices for cost cutting that are supported by UFT. The goal is to pick all the low-hanging fruit and cover critical features, allowing the operator to make some judgment calls at higher levels of complexity to save time and money. UFT achieves this by providing easily reconfigurable dialogs for interacting with the operator.

Any step deemed too expensive to automate is offloaded to the operator with simple, well-written instructions and/or pictures. These value decisions occur at various points in the development process against a test plan and can be handled with a minimal amount of interaction between the test engineer and the product team. Since the manual steps are easy to set up, there is little effort wasted if the decision to automate comes later on in the product's lifecycle.

Built for scalability and reusability, UFT's library of test routines has been growing for well over a decade. While it already has subroutines to perform many tests needed, each overall test plan is customized and the library expanded to ensure a given product meets its specifications. When a product fails a test, diagnostics are shown to the user, including details from the test and optional diagnostics relating specific components. UFT ensures each manufactured product is correctly…

- Moving itself
- Measuring
- Analyzing data
- Doing whatever the product does

… and is ready to be included into the next assembly or shipped to a customer.

Reuse and Scalability Before the First Line of Code.

After a survey of programming languages, Tcl/Tk was selected as the base language of UFT for the following reasons:

1. Tcl/Tk is a cross-platform language purpose-built to write middleware programs. The language's author, John Ousterhout, named it Tool Command Language (Tcl) when he designed it to automate many CAD design operations in different applications. It has since become an integral part of Xilinx Vivado design suite and other FPGA tools.
2. Tcl is an interpreted language, allowing the developer **LIVE DEBUG** capabilities. There is no need to stop the test, recompile, or get the product back to a specific state to debug the test. The test engineer can interactively query, debug, alter, and rerun the code from the Tcl command shell while the program is running and the product in the required state.
3. The Tk graphical extension to Tcl is a high-level language for generation of GUIs with minimal coding.
4. Tk is open-source and extensible, providing simple methods for inclusion of tools developed by a community of open-source contributors.
5. Expect – the number one Tcl/Tk extension for manufacturing. Originally developed for taking scripted control of a command-line program to the next level with interaction (Don Libes 1990), this language is ideal for interacting with *any hardware that presents a command line interface* (CLI) whether it be over serial or network connection. UFT uses Expect to communicate with:
    a. Unit Under Test (UUT) supporting command-line interface
    b. Programmable test devices such as
        i. Standard GPIB test equipment
        ii. Bulk call generators
        iii. Network Switchgear
        iv. Test equipment with non-standard interfaces

UFT Product Test State Machine

Production is a continuous operation. Each unit must endure a series of tests, but it will also be followed by another unit. A state machine is used to usher a single product through test in a controlled manner, resetting for the next product.

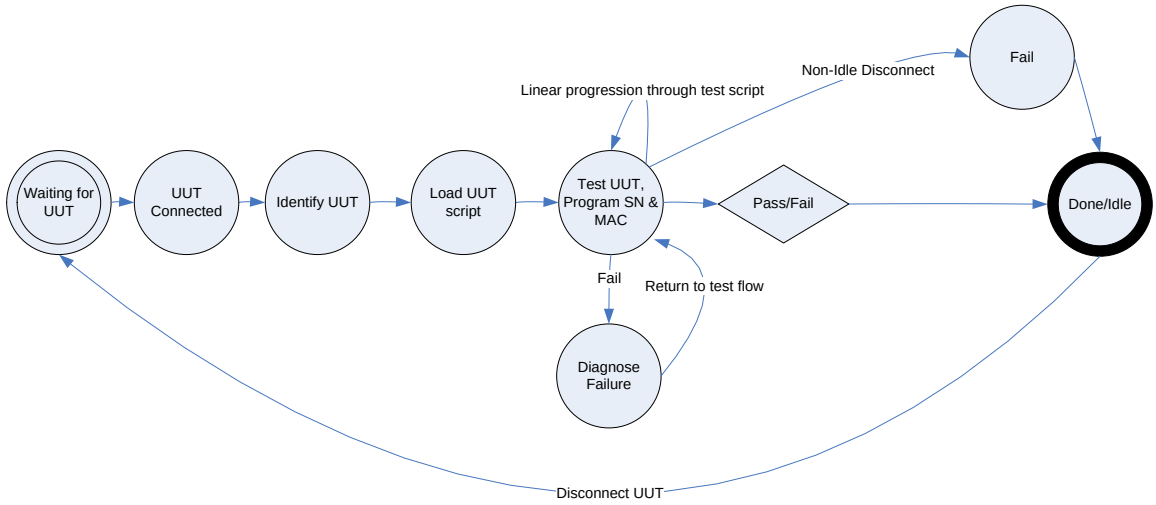

Figure 1: UUT State Machine

- The program initializes the designated port for communicating with the Unit Under Test (UUT), then waits to hear from it.
- As the UUT boots up, it announces various pieces of information which are used to identify the UUT and load the appropriate test program.
- When immediate start is not desired or for products that don't identify themselves, the operator scans a barcode or selects from a drop-down to select the product. A an optional "start" button allows the operator to initiate the test.
- The test script is generally a list of steps with attributes like diagnostics and critical failures (end test without proceeding to the next step). Procedural or decision-tree code can be added for optimal performance. For instance a top-level test that confirms that many things are correct can be followed *only on failure* by sub-tests that provide further diagnosis.
- Upon completion of the script, UFT waits in an idle state until the UUT is removed, then resets for the next UUT.

Scaling up for High Volume

The general principle of operation for this tester is that the operator will connect a UUT to any port test station, and it will automatically determine and start the appropriate test. Regardless of the setup procedure for the given UUT, the last step is to power up the UUT. This allows the tester to see the UUT boot into a known state, giving the tester an opportunity to identify and interact with the UUT.

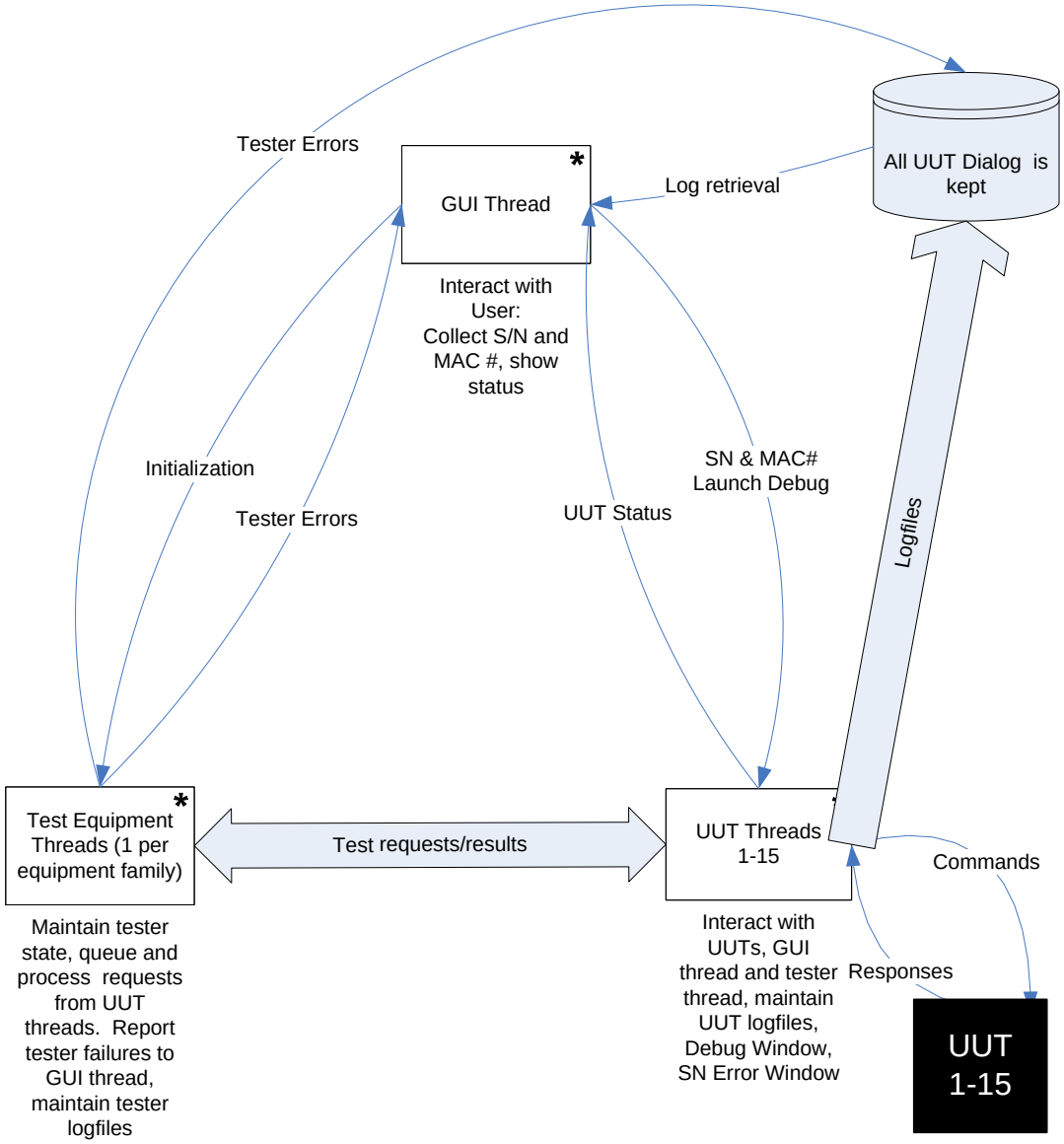

Figure 2: UFT Parallel Asynchronous Test

The parallelism of the tester can be scaled up to so the operator is never waiting, as follows:

1. Operator plugs each of a UUT's ports into tester, then connects power.
2. Operator scans UUT barcode into UAT to track and program unique identity. The test on the connected UUT begins. Meanwhile, the operator returns to step 1 until all ports are full or production queue is empty.
3. If equipped with enough test ports, the time required to load and unload all test ports will exceed the test time, allowing this tester to keep the operator busy on a continuous basis.
4. Operator disconnects passing UUTs and places them in the "good" bin, returning to step 1 until production queue is empty.
5. Failing UUTs: To keep production flowing the operator generally sets these aside in "bad" bin for later debug and continue running fresh product.
6. All logs are saved and cross-referenced in a database for retrieval using GSR. A barcode scan pulls up all records for a product for debug and diagnosis.
7. Troubleshooting and repair run at a different pace than production. To support both, UAT always displays an estimate of how soon each test slot will be ready for operator's attention, based on duration of previous test runs. This is handy for long-running tests on debug boards as the operator can walk away and do other work until the stated time has elapsed.

Underlying Mechanics

Compactness:

When writing a functional test a common step is to look for the appearance of a specific phrase, then continue to the next step OR come to an error/default condition after a preset timeout. The following 8-line tcl/expect code snippet is built on to make a number of the basic procedure calls that make up UFT. :

```
package require Expect

catch {open "com1:" r+} s

catch [exp_spawn -noecho -open $s]

set timeout 10

expect {

        "Boot Menu" {set productType UbootMenu}

        default {set productType generic}
```

```
}
```

Figure 3: Tcl Code 8-line Snippet

This small amount of code performs the following functions:

1. open a serial port and passes the port it to expect
2. sets a variable “productType” to “UbootMenu” if the string “Boot Menu” appears anywhere in the text produced by the port within 10 seconds. Program execution continues immediately upon this condition being met.
3. All other conditions including errors result in productType being set to “generic” execution to continue after 10 seconds.

For comparison, the following table shows the lengths of similar sample code in several other languages. Since these are publicly posted samples they do not have the exact behavior as described in the above Tcl/Expect snippet. The deviations are noted in red and would take more code to achieve the desired result.

Language	Volume of Code	Notes	Source of sample code
Visual Basic	25 Lines	Reads a line of text or times out in 10 seconds. No comparison performed.	Microsoft Learn https://learn.microsoft.com/en-us/dotnet/visual-basic/developing-apps/programming/computer-resources/how-to-receive-strings-from-serial-ports
C	162 Lines (201 Lines – 39 Comments)	The serial port must produce a predefined format of output.	Pololu Robotics & Electronics https://www.pololu.com/docs/0J73/15.6
LabView	15 nodes	An extra step -- the serial port is written first “IDN?”. After a fixed time delay the entire response from the serial port is read; no comparison performed.	LabView Simple Serial.vi example code.

Modularity:

Reliable use of test software with different hardware is a basic building block of this system. Each UUT/Tester operation is abstracted to a generic procedure call. For each UUT type and each Tester component, there is a “personality” script file containing these procedure calls. Included in the same file are any parameters required to perform each necessary operation on the given device.

Devices that are out-of-rev can be upgraded automatically to current-rev prior to test. This simplifies the burden on the test engineer by minimizing the number of test varieties that must be maintained.

Custom content such as test machine configurations, test steps and limits are stored in configuration files.

Auto-Recognition of UUT:

The general-purpose UUT-thread is provided with specific strings output by the various product families early on in their boot processes. These are used in the "Waiting for UUT" state to determine which "personality" script to load for test. Further uniqueness broken down in the "personality" script defines the use of any combination of product characteristics such as model, HW revision, FW revision, SW revision. Each UUTs has a unique connection port, and is addressed from 1 to N by logical port number (whether local, terminal server, or chassis port number).

Ports may be reserved for special functions such as firmware upgrades, so your operators don't need to press any buttons on the computer to perform these steps.

Disconnect Detection:

Non-idle disconnect detection or "Crash" of the product's CLI is always a possibility. This expectation is built into the low-level routines for interacting with the product as a "Timeout" error, generally a failure. Alternatively, there is an accepted issue in your product that occasionally causes negative results like a crash, UFT can employ fault-tolerance to recover and continue the test.

Idle disconnects are disconnects detected during the idle state, when the Operator is expected to disconnect the product. UUT thread loops through simple commands such as LED diagnostics, receiving the appropriate command prompt back from the product. When a command prompt is not received within a UUT-appropriate timeout, the UUT is assumed to be disconnected. The UUT thread displays final results and prepares itself to detect the next new UUT. The results are cleared from the screen upon the detection of the next UUT.

Logging:

A unique logfile is generated for each UUT session. Taking advantage of the unique nature of physical interfaces, there can only be one UUT at a time per Terminal Server port. Adding a timestamp to those details makes a unique filename as follows:

[TestName]_UUT_[StationName]_[PortNumber]_[Time_Datestamp].log

At the end of each test, logs are immediately sent to the server via fault-tolerant mechanism so that *production does not stop* for network/server issues. This provides an *indelible* record of the test while permitting production to continue

even while your network is down. If the same UUT returns to the tester several times, the logs are NOT appended nor overwritten – there is a unique log for each session.

All communication with the UUT or test equipment is in the form of a command-response dialog, which at times results in multiple dialogs going on simultaneously. Fortunately, there is a standardized, plain English way to track a dialog involving multiple parties. Logfiles are in playbill script format as follows:

[Timestamp][Speaker:Port]:[Message]

This allows technicians to view the events chronologically, or find specific events with a search. UUT logs do NOT key in on the keywords in the “expect” statements, instead recording comprehensive records of everything said by all parties.

Some tests produce graphical, directory structures, or other digital collateral. This is captured along with the log file and submitted to the central repository in zip format.

For More Information

Call: 571-601-0744
Email: support@CAEIntegration.com
Schedule a video call: CAEIntegration.com/contact-us