



TAC Automation Models

Overview

One of the main reasons the Record / Playback (R/P) methodology of test automation usually fails to achieve set testing goals is that the paradigm does not sufficiently model real world testing. In order to understand this, we need to investigate several different but interrelated models. This document will discuss these models and then advance the **Test & Automation Consulting LLC (TAC) Abstract Function** model for test automation.

This document should be read in conjunction with the **TAC Automation Architecture Theory.doc** document.

Record / Playback (R/P) Model

Why doesn't R/P ever work (or work so very rarely that it is hardly worth talking about?) Ask just about any automator why the basic R/P methodology does not work well, and you are likely to hear a number of different things: hard coded values, static synchronization values, test cases can't handle changes, etc. While all of these are true to some extent, perhaps we can get to the heart of the matter by basing the discussion on the model that R/P represents.

A recorded script is based on a model that the tool vendor envisioned when creating the tool. That model appears to include a world where:

- Each test case is completely repeatable, with only very minor variations.
- The script recorder does not really need any special skills – often the vendor recommends that a business analyst is sufficient to record the script.
- The environment is always stable enough that a recorded script is repeatable.

What does this model say about manual testers? Do manual testers really do exactly the same thing each time through a test case? Do they add no extra value such that a business analyst can do the same testing by operating the system as if using it to execute business logic? This particular model of testing implies that a tester is not much more than a monkey, simply someone seated at a console, banging on the keys. If we had enough monkeys (say an infinite number) then they could simply bang on enough keyboards and all of the software errors would be found.

The time-tested fact is that manual testing works and that professional testers do add great value to software testing. This testing is not random and seldom accidental; there are some very good reasons that manual testing methodology and manual test cases have



Innovative solutions to maximize your testing investment.

worked so well for at least half a century. So what part of the manual testing model is lost when ported to the R/P automation tool? More importantly, what must we add to the basic playback script to make it match the manual test case model? To answer these questions, let's look at the testing model that manual testers use.

Manual Testing Model

There is a model that can be derived from observing how professional testers do their testing. Look at a typical manual test case. With few exceptions, the basic test case script consists of a couple columns of text captured in a Word or Excel file, as shown below.

1	Add a record to the database	First name: Gerry Last name: Franklin SSN: 234-34-5678	Record created, Record # returned
2	Search for the name	Franklin, Gerry	Expect to find it
3	Edit the record	Occupation: Lawyer Income: \$125,000	Expect dialog verifying change
4	Check record ordering	Record # from step 1	Expect valid ordering
5	Etc.		

The first column identifies a task to be performed; a business task that the test creator assumes is clearly understood by the tester, business analyst, or user of the system. This task is usually an action that is a single logical step in using the system: changing the application state from one to another. Note that there is usually no direct, one to one correspondence with the graphical user interface; the text does not say anything about moving the mouse, clicking on a key or mouse button. The assumption is made that performing the task will consist of the user performing those kind of physical actions based on their knowledge of the system and of the OS. Rarely does the test step explicitly explain out the action so fully that someone who does not understand the application can run the step. Why? Well, for one thing, the test case would become so long that it would take almost forever to write it. This would reduce the number of test cases that a tester could write. It would also make the test case brittle; any change would invalidate the test case to the point it would no longer be correct. Instead, the usual test case assumes that *the tester will add some system knowledge* to make up for the lack of specificity.

Contrast this with an automated test case where every single action that the user performs is tied intimately to the interface. Clearly, the two models diverge on the amount and kind of details included. The question is whether we can do anything about the need of the computer for all of the detail. Read on.

The second column of the test case consists of the data to use in performing the action; again, no direct mapping is made to the GUI. The final column, in general terms describes what reaction is expected from the system under test.



Note the beauty of this model. When there are minor changes to the system or the environment, the test case rarely needs to be modified. This is because it is just an outline of a test, an incomplete recipe or algorithm that is combined with something else in order to create an executable test case.

Now, obviously, this test case is not directly executable through an automation tool. It must be taken and executed by a manual tester to have value. The tester adds *something* to the manual test case to execute the test. I believe that the intangibles that are added by the tester during execution can be narrowed down to two qualities: **context and reasonableness**. The manual tester reads a step from the manual test case: *perform this task*. Several tasks are given sequentially to perform a complete test. The tester must understand the context of each action within the confines of the system and the environment and performs the physical actions on the workstation, through the GUI, to achieve the task using the data in the following column of the test case.

Consider how important the context of the action is. If the tester is told to open a file, but the system is not in a state such that a file can be opened directly, the tester changes the state of the system to move to a state where the file can be opened. The test case did not direct them to do it, but the tester understands the context of the step text and acts to make it happen. Let's take it a step further. As the tester waits for the file to open, there certainly are several possible outcomes that can occur whereby the file does not open successfully. The manual tester does not get lost or confused when the file does not open exactly as expected. If it takes longer to open one time than another time, the tester does not panic and fail the test case; they apply reasoning to the problem. Perhaps the network is slow. Perhaps the file is huge. If the file does not open immediately, but pops up an intermediate error or message, the tester applies rational thought to the window's contents and determines, based on experience and rationality, whether the reaction of the system constitutes an actual defect or merely an annoyance.

The point to be made is that the tester applies a standard of **reasonableness** to the reactions they see, and a sense of **context** to the tasks that they are to perform. The very two features not built into the test tool! Our task will be to find an automation model where we can simulate these two properties of the manual tester through programming.

Event-driven Architecture model

Let's look at another model which has a direct bearing on automated testing; that of the event driven or state model that all GUI architectures exhibit. In the above explanation of the manual testing model, I used the term state several times. Just what does state mean when using it in this context?

Consider how Windows works. It is usually in a steady state, just sitting idle, waiting for the user to do something. Certainly, in the background, Windows is busy, executing clean up tasks, updating the screen, etc. This background processing is certainly



Innovative solutions to maximize your testing investment.

necessary stuff, but nothing of much direct interest to the user or the application at hand. Let's assume that the user wants to write a letter to someone. First step is to start the word processor. It does not start automatically - we cannot "think it on." The user must physically instruct Windows to start the application.

The user must perform a positive physical action using the mouse and/or keyboard, perhaps finding the icon in the start menu and clicking on it. The application, in conjunction with the OS, does the rest; finding the application image on disk, loading it, initializing data structures, drawing the screen, etc. More on this in a moment. The point right now is that, after the word processor has been started, it does nothing. It does not start writing the letter. It is in a steady state waiting for the user to initiate a new action. Every step of a test case, or a user working on a business task for that matter follows this same model: direct the application to do something, and then wait for it to finish. Each time the computer finishes a step, the application is very patient; it waits for the user to order it to do something else.

When a programmer speaks about an event-driven architecture, this is not strictly what they are speaking about. It is an apt description, however, of how a GUI application actually works; set up an action: tell it to go. Set up an action: tell it to go. Repeat as necessary. There is no actual smooth flow through a GUI architecture; instead there is this jerky, stop and go action that consists of an order from a user and a follow through by the system.

99.9 percent of applications work this way in Windows. Now, let's get a bit more granular. What does it mean to set up an action and then tell it to go?

With Windows (and other GUI operating systems) we have a set of metaphors that we use to interact with an application; we call the *physical manifestation* of these metaphors controls. Each control can either help set up an action or give us an indication of the reaction to an action; some do both. Edit or list box, radio or check button, push button or label, menu item, etc. Each of these metaphors is well understood by an experienced user so that little training is necessary for most applications. This was the advantage of Windows that ended up killing DOS; every application has a common (!?) interface so that if one knows how to execute one Windows application, they likely could figure out any other application they had to use.

These controls are usually how we instruct Windows to move from one state to another. We interact with the controls to input data, we view data in the controls to understand the current state we are in. If you want to know what state your application is in right now, simply look at the monitor? What is the top window? What are the available menus? What are in the edit boxes, radio and check boxes? We understand the current state visually. Experienced users learn to determine the current state almost unconsciously.



Innovative solutions to maximize your testing investment.

Once we understand what the current state of the application is, we are ready to direct the system to move to another state. This directly corresponds to the first model illustrated here; the tester interacts with the application under test by identifying the current state, evaluating what the next task is to be done, and interacting with the system (via keyboard and mouse) to change the current state to the next state in the test.

The change of state for each step of the execution can be split up into four distinct parts. Each of the four steps will be explained below.

- 1. Ensure that the system under test (SUT) is in the correct state to perform the task.**
- 2. Fill in the interface elements necessary to instruct the system to perform the next task.**
- 3. Trigger the task.**
- 4. Wait for completion or some state that indicates the task is not going to complete.**

First the user must make the determination that the application is sitting in the correct state to perform the desired task and change to the next state. If we are to open a file, we had better be in the main window with no other dialogs open. If we are not sitting in the main window, we must either get there (by driving the application using the same four steps) or we must fail the test case. We cannot advance to opening the file from any other state.

Secondly, we must indicate to the system what it is, exactly, that we want it to do. We do this by interfacing with the controls of the application. Continuing with the file open example, we will use the menu structure to bring up an open file dialog box, then use the controls of the dialog to specify a single file to open; either by manipulating the tree view and tree list controls, or by typing in the file name and path to the edit box.

We eventually get to the point where we have supplied the application with sufficient data to clearly explain what we want. At this point we move to the third part of the action: triggering the task. There is always some point at which we declare we have finished with the set up and now the system should begin its work. Often the trigger consists of pushing a button, clicking a menu, or pressing the return key.

At this point, the application takes over. No longer is the user in control; the application in conjunction with the OS is going to start cranking in an attempt to fulfill the wishes of the user. Whatever the task, whatever the context: at this point the application is in full control and cannot be stopped. There are three possibilities at this point:

1. The action completes as expected (i.e. the file opens)
2. We get an intermediate action other than the file opening (error or message box which will convey information to the user as to why the file is not yet or may never open)



Innovative solutions to maximize your testing investment.

3. The action never does finish (in technical terms, the system goes to never never land.)

What does the automation tool do in a R/P script? It generally feeds in commands blindly (or almost blindly within the limits of the tool's basic capability.) There is no feedback loop as in the manual model where anomalies can be handled. And there are almost always anomalies. Applications change from build to build, release to release. Environments change (Win 95 to Win 98 to Win NT to Win 2K to Win ME to Win XP.) Is it any wonder that R/P fails?

Context and Reasonableness

The manual testing model has two unique characteristics that the R/P model is almost completely lacking; **context** and **reasonableness**: the manual tester's contributions to the test run. The manual test case is often left somewhat vague as to the implementation details (more on this later.) As noted above, the tester reads an instruction from the manual test case consisting of a task to be performed, the data to use for the task, and the expected reaction of the system. The tester supplies domain knowledge suitable for the requested task. Based on that knowledge, the tester performs the task, interprets the actual result and decides whether it matches the expected result.

In this section, I would like to present an automation model that compares well with the manual testing model and supply a template that can be directly translated into successful automation code. In this model we will leverage the peculiarities of the event-driven architecture model that all GUI operating systems (OS) exhibit.

A manual test case rarely specifies any details about the interface used in completing the task. For example, if we are testing a word processor package, the step is much more likely to read, "Open a File" than "Move the mouse to the file menu, click on the left mouse button, etc."

On its own, the task "Open a File" is an abstract idea. No specifics are given. An assumption is made that the tester running the test will understand the task and know what discrete actions must be performed to complete the task. By abstracting the task, we ensure that when the interface changes, the test case will still be valid and executable ***as long as the tester knows how to perform the task within the new context***. Think about that. In each release of our theoretical word processor SUT we have had to open a file. The specific physical steps have changed from version to version. Certainly the actions performed by the applications in executing the task have changed. However, the logical, abstract idea of opening a file has not changed, nor have the test case instructions as long as the abstract ideas in the test case are still valid.

This is not by accident. A software package tends to allow the same abstract tasks to be performed from version to version. It is not as if the essential tasks that our word processor performs become database tasks when we upgrade to version N+1. We buy



Innovative solutions to maximize your testing investment.

software with the expectation that it will help us perform certain tasks. If the new version did not perform those tasks, we would probably not purchase the upgrade.

Our manual test cases are more valuable due to their abstract nature; they have very little direct correspondence with the physical GUI. Consider, however, an R/P automated test case. Here, there is a direct correspondence between the statements in the script and the physical actions that must occur in the interface. Any change in the interface or behavior of the system will break the R/P test script.¹ Ideally, we would like to be able to abstract our automation test cases so they will work from release to release with minimal changes.

Let us list this as a goal of our automation program:

Each automated test case should be built such that changes in the physical interface of the system do not require changes in the test scripts to allow them to run.

That brings us full circle to the main topic of this document: How do we model our automated tests to remain static even when the interface of the SUT changes? The answer is in an effective **functional decomposition** of the testing domain into abstract tasks, each task captured into a programmed, callable function. The one to one correspondence of “line of code” to GUI element action does not then exist in the test script; instead it exists in the functions called by the automated test case. A change in AUT interface will still break the test cases; we cannot prevent that as the test tool must work with each GUI element. However, broken test cases are fixed by repairing the code in the functions and not the scripts. One single point of repair for any one fix. The maintenance on 1000 scripts should not be appreciably more than that on 50 scripts.

This functional decomposition of tasks is the essential feature of the Framework-based architecture, as described in the document, **TAC Automation Architecture Theory.doc**.

We want to model our automated test cases on manual test cases using the abstract task model. The better we do that, the closer we’ll be to achieving successful and effective automation. A manual test case has very little overhead when compared to an automated test case; it pays back in positive return on investment (ROI) every time it’s run. An automated test project must strive for that very outcome: reduced cost per test case through lower maintenance costs. Without a positive ROI over the course of the project, there is no valid reason to automate.

If we can, we would like to construct our automated test case by filling in a “document” using a similar model as that used for a manual test case:

¹ An excellent treatment of the problems of direct correspondence of script statements to physical GUI elements may be found in the June 2001 issue of The Journal of Software Testing Professionals, written by John Kent.



Innovative solutions to maximize your testing investment.

task | data to use for the task | expected action

Note the similarity between the manual test case step and the function call:

RC = Function_Name(Data1, Data2, ...)

Note that it does not matter whether or not a physical script is used. Many automators believe that scripts should never be used, and that a table-driven architecture should replace individual scripts when creating test cases. However, the structure of the test case does not really depend on the run-time architecture used. If we abstract the task correctly, we should be able to use the created function efficiently in any architecture.

Having decided to try to emulate a manual test case, it is imperative that we understand how the manual tester adds context and reasonableness to the test case in order to create an effective test. That was discussed above: the four step process. Since we will be using functional decomposition to create executable tasks for our test cases, we must figure out how to programmatically implement the needed context and reasonableness inside each function.

The Four Step Process Expanded

As noted above, each state change in the system can be broken down to four discrete steps:

- 1. Ensure that the SUT is in the correct state to perform the task.**
- 2. Fill in the interface elements necessary to instruct the system how to perform the next task.**
- 3. Trigger the task.**
- 4. Wait for completion or some state that indicates the task is not going to complete.**

Step 1 essentially deals with context: can we physically perform the action from the SUT's current state? Step 4 deals mainly with reasonableness: Did we get what we expected? Did we get a steady state other than what we expected? Did we even finish?

In the following sections, we will deal with how we can make our automated test functions follow these same four steps so that we might add both context and reasonableness to them. If we can develop a template for dealing with these four steps, we should be able to automate just about any task. If we can automate tasks, then we can stack them (much like a manual script) and create full test cases.

Step 1: Where the Heck Are We?

The first step simply ensures that we can actually start to perform the task. For example, if we are suppose to try to open a file, but we are not sitting in the main window with



Innovative solutions to maximize your testing investment.

access to the menu item, “Open File...”, we have to change the state of the system to get to the main window.

This action is simple for manual testers. They determine the state of the system as reflected by the visual cues on the screen. If the main window is not active, they apply keystrokes and/or mouse actions to move the state of the SUT where it needs to be.

Unfortunately, for our automation needs, we do not have this luxury. Conceivably, we could interpret the state of the SUT and apply an algorithm to it forcing a state change by moving to the place where we need to be. Consider, however, the complexity this would add to each function! Since there could be in innumerable states other than the required one, we must be able to test for where we actually are, and then calculate how we can move to the state where we need to go.

Complexity is the enemy of all programming tasks; it increases maintenance and reduces ROI.

Fortunately, when taken in the context of our task-based automation, this is a solvable problem. If the test case is correctly designed, the “step 4” of each task will always leave the state of the SUT in exactly the correct position for the next task’s “step 1.” Therefore, if the SUT is not in the correct state, it shows either a logic error in the script design, or an execution error. Either way, we want to let the user know that there was an error. If we are not in the correct state: we will log the problem, throw an error, and immediately fail the test case. Period. Do not try to figure out what went wrong, nor try to move to the correct state programmatically. Keep each function as simple as possible!

So how do we determine that we are in the correct state to continue? If the SUT is well designed, there should always be a visual cue to the system’s current state. Usually we are looking for the correct window to exist and be enabled for input.

Each automation tool has a way of determining the existence and state of a window. To ensure commonality throughout all tools supported by T-API, the TAC T-API uses a function called **WaitForWin_TAC()** to supply this needed functionality. In pseudo code, the function is shown below:

```
WaitForWindow(Window, TimeToWait, IsEnabled)  
  If (IsEnabled = TRUE) Then State = “Enabled”  
  Else State = “Exists”  
  Do  
    If (WindowName is in State) then Return 0  
    Else Wait 1 second  
  While (--TimeToWait > 0)  
  Return 1
```



End Function

Note that the function will return a 0 if the window is found in the expected state, else it will continue to cycle around waiting for the window to move into that state. Each time the window is not found in the correct state, we wait 1 second and then try again. The scripter calls this function with the name of the window, the time they are willing to wait, and the state required: either the window exists, or it both exists and is enabled. If WaitForWin times out, it returns a 1. The scripter forces an error and returns failure from the function which then ripples up and forces a failure of the script. An invocation of the function follows:

```
If (WaitForWin(MainWin, 10, TRUE) = 1) Then  
    Error Step1Synch, "Expected the " + MainWin + "but did not get it.  
EndIf
```

Again, this assumes that the tool has an error handler (such as QARun™ and Rational Robot™ have.) If the tool does not contain an error handler (such as Mercury WinRunner™) then other code must be written to simulate the behavior.

As stated before, this code may not strictly be necessary for step 1, as the previous task's "step 4" should have left the SUT in the correct state, or else thrown its own failure.

We can "WaitFor()" many different objects other than a window. We could be waiting for a particular object to exist, to become enabled, for a status bar message to appear, etc. TAC T-API actually has many different "WaitFor()" functions available.

Assuming we are in the correct place, the function execution now moves to step 2.

Step 2: Fill in Stuff

This step is very similar regardless of whether it is taken by a manual tester or an automated test. The interface interactions necessary to set up the state change are performed sequentially. Edit boxes get typed in, buttons get clicked, list items get selected, etc.

For automation, the scripting method may be simple brute force where each control is listed and data is passed in according to the tool's standard methods. In fact, the individual steps may simply be recorded. Or, there are a number of clever programming tricks which can be taken to minimize the amount of code in the function, such as using data-driven techniques where the data being read in from the data base / spreadsheet are fed to the correct controls automatically.

A critical element to this step is to ensure that the mechanism used to deal with interface elements actually monitors for control failures. For example, if an attempt is made to



Innovative solutions to maximize your testing investment.

feed keystrokes into an edit box that is disabled, the automator must make sure that the function is failed, a run-time error is generated, and consequently the test case failed.

Each automation tool allows different control feeding possibilities; T-API has a number of functions available in different tools for performing this task.

Once all of the interface changes have been completed, we are ready for step 3.

Step 3: Go

This step is identical whether performed by an automated script or by a manual tester. Whether the trigger action is pushing a button, clicking on a menu item, or some other kind of action, it represents the last point where the task invocation may be aborted. After the trigger action, the operating system working with the AUT will take over and complete the action. Step 4 starts immediately upon triggering the action.

Step 4: Are We There Yet (Repeat as Necessary)

This step is by far the most problematic in all of automation. If every test case worked exactly the way we expected, this step would not even be necessary. Unfortunately, far too often we do not get what we expect, either through intentional change in interface or actual defects.

For manual testers, the waiting time after triggering the action is often not noticeable. They await one of four possible outcomes:

1. If the action completes correctly, it is almost a seamless slide into the next action.
2. If the action does not complete and the system goes into never-never land, the tester will tend to wait an appropriate time, reset the application and/or the workstation, and move onto the next test case after recording the error.
3. A third possibility is an unexpected reaction, such as an error dialog or message dialog signifying a fatal condition for the test.
4. The fourth possibility is a message box signifying a condition that may be rectified by a positive action by the tester / automated test case. For example, if we are opening a file, we might get a message that the file cannot be opened because the drive is not mapped. If the user maps the drive to the correct server, the file could yet open. Of course, having mapped the drive, the file open may yet cause an error or go to never-never land.

It is obvious that the current generation of automation tools is not designed to compete with the manual tester in handling all of the possible outcomes to an action. The unenlightened automator and the recorded test case will assume that actions will always complete correctly. Fortunately for them, the task will often result in the expected reaction in the short term. Unfortunately, an optimistic automator and a recording script will usually create automated tests that will eventually languish in the bit bin because



they fail to handle the unexpected in the long term and require too much maintenance over the lifetime of the project.

The successful automator must be a pessimist – or perhaps a realist. They understand that if there are 20 lines of code in the function there are 21 different places it could fail. When running on a half dozen different platforms with varying levels of operating system patches, with SUT changes from build to build, with unproven test environments and uncertain networks, the possibilities of the function always running successfully are extremely slim. The difference between successful automation and shelf-ware is often the error handling in the step 4 phase of each function.

Automation tools do not have built into them a good way of dealing with step 4 problems; they are built using the assumption that we are always waiting for a single, discernable outcome. Luckily, the tools do give automators the facilities needed to create very robust solutions. The heart of the solution to multiple possible outcomes is in the “**WaitFor()**” function defined earlier.

Consider the problem. We have triggered an action and the application and operating system have taken over performing the task. The test case must wait for the action to complete before continuing. If the expected reaction to the task occurs, we can go on to the next task. If, however, something else happens, or no known reaction occurs within a reasonable time, the function must be able to decipher the event, log the discrepancy, and inform the script that the test case has failed.

For a concrete example, we will look at the file open task that we have been discussing through out this document. We have walked through the interface (the Open File Dialog box) and triggered the open action. Now we are waiting for the file to open. But it might not open. We may get one of a dozen different error messages (e.g. File is corrupted.) We may get a warning message with a range of choices (e.g. file is from a previous release. Do you want to open it, convert it, or cancel?) We may get nothing right away; the file may be quite large and taking a long time to process. We may get nothing ever; perhaps the word processor has had a catastrophic failure and frozen the process. How will we handle all of the eventualities?

The answer is a logical structure called the **Single-Action While Loop (SAWL.)** This programming construct allows us to wait for a variety of different actions and deal with each eventuality. The SAWL will wait for a given number of seconds (determined by the automator) for a resolution to the task. As it cycles through the known possibilities, it acts much like the manual tester who can deal with any known eventuality as well as the unknown. To get a complete understanding of this construct, please refer to the document, **TAC Architecture and High Level Design.doc**. For our purposes here, it can be seen as a number of WaitForWin() functions in a while loop that executes once each second until either a pre-determined reaction occurs or the loop times out.



The [pseudo] code follows:

```
1  While (--Time > 0)
2    If (WaitForWin(MainWin, 0, TRUE) = 0) Then
3      Return File opened correctly
4    Else If (WaitForWin(ErrorWin, 0, FALSE) = 0) Then
5      Handle Error and Exit
6    Else If (WaitForWin(MessageWin, 0, FALSE) = 0) Then
7      Handle Message and Maybe Exit
8    Else
9      Wait 1 second
10   EndIf
11  End While
12  If Timed out, log error, take picture of screen
```

A couple of notes. Each of the WaitForWin() functions has a time argument of 0 seconds. This permits us to check for several specific conditions very quickly while expending minimum resources. Only if the condition exists will the clause handling the condition fire. This is why we prefer using the TAC T-API construct, WaitForWin(), rather than the built-in synchronization functions. We can explicitly control the timing and return value of the function and then stack them in any order we want.

In line 2, we check for the state of the main window (the File Open dialog has closed and there was no other modal dialog opened.) This would signify the file opening successfully.

In line 4, we check for the existence of an error box. If an error box has popped, we pick up the error message (and possibly compare it to the expected error message) [in line 5] and exit the function.

In line 6, we check for the existence of a message box. If there is a message box active, we may handle it in line 7 and continue on in the loop or we may exit the function if we were not prepared to handle the message.

By adding more “else if” statements, we can handle any eventuality we know about. The time out handler is in case the action never finishes (i.e. the system has frozen) or we get an unknown steady state reaction (i.e. a new kind of dialog box we never saw before.)

As stated before, this is not meant to be a complete explanation of the loop; instead it is meant to point out that we must have a way of traversing step 4 satisfactorily. We must add the reasonableness that the manual tester adds through their experience and senses.

The entire function structure will look like the following:



Function DoSomething(Arg1, Arg2, ...)

// Step 1

If (WaitForWin(MainWin, 10, TRUE) = 1) Then
 Log the error and exit
EndIf

// Step 2

Handle the interface elements

// Step 3

Press Go button

// Step 4

While (Not timed out)

If (WaitForWin(Expected_reaction_Win, 0, TRUE) = 0) Then

Exit Function with success code

Else If (WaitForWin(ErrorBox, 0, FALSE) = 0) Then

Handle and log error, exit with error code

Else If (WaitForWin(MessageBox, 0, FALSE) = 0) Then

Handle message- if recoverable, continue, else exit with error

Else

Wait for 1 second

Decrement counter

End If

While End

// If we get here, we timed out

Log error, take picture of screen, exit with error code

End Function

This template may be used for almost any functional task that we might have to create for our automation task. Note that it follows the logic of the manual tester very closely, by adding context checks and reasonableness to an automated test case through every step that is taken.