Inference Engine Logic in SilkTest

Prepared by: Jeff Nyman

Inference Engine Logic in SilkTest 1
Two Types of Logic 4
Assignment Logic 4
Conditional Logic 5
Explicit Logic vs. Implied Logic 8
The Inference Engine 9
Modifying defaults.inc 10
Creating Appstates 12
Creating Testcases 14
Results-Route Matrix 15
Testcase Driver 15
Recapping the Logic 18

SilkTest has a recovery system built into it and it is a powerful tool in its own right as a way of making sure that unattended testing can be as robust as is possible. That said, the recovery system is only as "smart" as you tell it to be: it has no way to "learn" or "make judgments" about what is happening as part of script execution and then respond accordingly. One way to make this system even more robust is to create a type of *inference engine* that is build upon the recovery system that is an inherent part of SilkTest.

What do I mean by an inference engine? Well, in computer science you often have systems that contain a sort of knowledge base and a set of algorithms or rules that infer new facts from that knowledge base and from data input to the system. If this knowledge base is built up of some form of codified human expertise then you have the basis of what is called an expert system. That, however, can get tricky. The idea of an expert system is that the codified human expertise is then used to solve problems in some relevant domain. Here the degree of problem solving "ability", if such it can be called, is based on the quality of the data and rules obtained from the person who built the knowledge base in the first place.

The point I want to get across here, however, is that the expert system derives its "answers" by running the knowledge base through an inference engine, which is really just a set of routines that interacts with the user and processes the results from the rules and data in the knowledge base.

Right now I see building a truly effective expert system in SilkTest (or any automated testing tool) as a bit of a pipe dream. However I see building an inference engine not only as very feasible but also as very practical with positive return on investment for an automated test effort. Specifically what I want to describe here is a backward-chaining, goal-seeking inference engine that is implemented natively in SilkTest. Even with that, my goal is not a full inference engine at this point but rather something that mimics the basic concept to some degree within a limited context of applicability. Here that context, put most simply, will be a generalized facility for navigating and verifying a conditionally branching logic tree relative to functionality of some application. The idea is to navigate and verify this tree in an automated fashion while maintaining efficiency for the general testing approach used in the automated test scripts.

The overall flow that the conditional logic controls, which is termed execution branching, is the main focus of complexity in any type of software, and that tends to include client/server applications – both traditional and Web-based - that have a great deal of functionality or traversal paths. I have found that the focus on execution branching holds for Web sites as well and certainly holds true for desktop applications. In other words, if what I say here makes sense then it has a wide area of applicability.

I will admit to being a bit simplistic in what I say next but it could certainly be argued that any software is really built upon only two fundamental building blocks: *conditional logic* and *assignment logic*. This is the basis behind event-driven programming and note

that can be the case whether you are dealing with a structural language or an object-oriented language.

By way of example for the conditional logic: consider that an input trigger, event, or state (such as a key press or mouse click) will assign a value to some internal register. This, in turn, will cause conditional logic in the application to switch down some branch (path). As execution proceeds conditionally down each branch, the application you are testing will unconditionally execute one set of assignments instead of another. The conditional logic is sort of like the route you will drive down a road.

There are two types of assignment logic: the *triggers* (like traffic lights with arrows at each intersection) and the *results* needing verification (like the landmarks down each street). The traffic lights could send you down many possible routes but a given street should always have the same landmarks.

So how does all of this apply to an inference system as part of an automated test framework? Well, one of the needs of the system is to make sure that general test execution is always such that the *tests travel the least number of routes to verify all landmarks*. Following the analogy laid out above, the main goal of the inference engine is to control the arrows on the traffic lights.

To do this requires having a set of rules that the inference engine follows. While the implementation can be more or less difficult, the basic concept is simple: first, trigger the conditional logic (like traffic light arrows) and then check the results (making sure the actual landmarks match those you expected on whatever street you go down). You will know that the traffic light arrows (triggers) caused you to go down the correct street because the landmarks (results) are what you expected to see. If the landmarks differ, you found a defect - either in the triggers or in the results. What is nice about this is that rule expression becomes incredibly simple, as it is just this:

(base state + trigger) = result

Any effective system of this type would have to allow for multiple triggers. Also, the term “base state” can be thought of as a starting point and, in a similar fashion, there must be allowance for multiple base states if such is necessary.

(In SilkTest, a “base state” is a type of “application state” or appstate.)

The first stage of doing this process is an analysis such that you decompose the conditional logic and assignment logic in the application into a set of trigger-result rules that can be applied to the inference engine. Clearly this is something that a test team and a development team should work on together. (At the very least the development team should review what the test team has come up with.) This analysis should give you information about the following aspects of the application:

- **Triggers**
  Verification causes for which application states will be defined.

- **Results**
  Verification points for which test cases will be applied.

- **Variation Sets**
  Conditional logic, decomposed into combinations of triggers and results called rules.

- **Routes**
  Combinations of rules for verifying unique paths, from a starting point to an ending point.

Two Types of Logic

I mentioned before the distinction of assignment and conditional logic and indicated that this can serve as the basis for thinking about how applications are structured. I want to now follow on with that discussion a bit more.

Assignment Logic

It is possible to eliminate complexity in the assignment logic portions of most applications by identifying and maintaining a simple one-to-one relationship between the assignments in the application's functionality and the assignments in the inference engine. As with my discussion in the previous section, it is necessary to manage two types of assignments: the triggers and the results. The triggers retrieve values for execution or entry while the results retrieve expected values for comparison to actual values.

You also have to have some means of indicating all value assignment transitions. These correspond to the operational transitions you see in the application when data is gathered in some fashion, entered into the application, and then displayed. Note that such transitions can also include how that data changes as part of the operations of the application. Finally, it is necessary to update value assignments after triggers or before verification, at whatever is the earliest point that makes sense to do so. Speaking somewhat generally, but with experience, the easiest way to manage the trigger and comparison data is to construct a data dictionary or database that holds each of the triggers and comparison data elements that are needed for entry and verification. (Here your "database" can be nothing more than a character delimited file.)

As an example of what I am talking about, if you were triggering a control, you will need to store the association of the mouse click, key combination, menu selection and/or any other navigation keystrokes with that control on that particular part of the application.

(The “part” may be a dialog box in a Win32 app or a particular Web page within a Web site.) How you do this is an entire topic in its own right. It really depends on the amount of overhead that your environment permits. If we can afford the memory, disk space, concurrent operation or network traffic overhead of a relational database, this is probably the best tool. That said, spreadsheets or delimited text data files can do the trick just as well. In any event, you need to create storage structures analogous to relational tables. You then need to populate those tables with data so that you can store and retrieve trigger and verification values that mirror the application’s functionality.

As an example, if you want to verify whether your application has actually enabled a control as expected, your database will need to contain that data. You can then retrieve that data at the appropriate point.

There can clearly be some complexity here. For example, a trigger may require one of several input device triggers. The actual trigger may be beyond the scope of what you are testing. So you might build a standard function to retrieve and dispense the appropriate platform-specific trigger, whatever it happens to be at a given time. In a verification example, perhaps the end result you get from the application perspective looks the same from a user’s point of view but, in fact, is different. An example might be where one trigger for a piece of functionality returns a value, say 20, as an integer. However, another trigger, for that same piece of functionality, returns the value as "20" (a string value of the integer). You can write a function that returns what is required depending on what your verification procedure needs for comparison to actual.

You will also need to establish and maintain default values for start points such as log in screens, dialogs that need to be opened, etc. In other words, some triggers will only happen from a certain window or when certain other conditions are met, such as a log-in screen having already been passed.

You can contain any and all of this complexity by creating standard classes that encapsulate various controls and methods. These defined controls and methods will allow you to manage the particulars of each side of the assignment.

Conditional Logic

The previous section was about attempting to mitigate some complexity in the assignment logic. In a similar fashion, you can eliminate complexity in the conditional logic parts of an application.

One thing you can do, as a first step, is extrapolate the variations, normalize any “OR” conditions into “AND” conditions, and then remove any redundancies. That is quite a first step, but it is actually not as daunting as it might sound.

Keep in mind that each variation set is a collection of rules that the inference engine can use. For example, if result C is true when trigger A is true “OR” when trigger B is true,

then you can identify three variations that, together, provide all the rules necessary to prove correct operation:

Variation 1	Result C = TRUE	(when) Trigger A = TRUE	(and) Trigger B = FALSE
Variation 2	Result C = TRUE	(when) Trigger A = FALSE	(and) Trigger B = TRUE
Variation 3	Result C = FALSE	(when) Trigger A = FALSE	(and) Trigger B = FALSE

Notice how, with this approach, you have eliminated the "OR" representation in this logical condition by converting (normalizing) each variation into an "AND" condition. You might also notice that this lets you eliminate the case where both A "AND" B are true. The reason for this is that achieving a true result in this "OR" condition only ever requires a single trigger be true. Therefore, you can eliminate the redundant variation where both A "AND" B are true. This example is one of four types of "OR" conditions. Each such condition requires a slightly different normalization, as follows:

- One-And-Only-One
  (One and only one trigger must be true. This is also called Exclusive OR or XOR.)

- Exclusive
  (At most, only one trigger can be true.)

- Inclusive
  (At least one trigger must be true.)

- Normal
  (Any combination of true and false for triggers is permitted.)

A key point here is that you only normalize "OR" conditions. While it is true that "AND" conditions require no normalization, they do still offer redundancy elimination opportunities. I say that because achieving a false result in any "AND" only ever requires a single trigger to be false. Therefore, you can eliminate all redundant "AND" variations where more than one trigger is false.

Keep in mind that what I just showed you was one set of logic based on a particular trigger + result set of entities. You can and will have many more in any moderately sized application of even minimal complexity. However, with that said, it might be tempting to think that you should just add all logic sets together to produce your full set. In other words, taking every bit of normalized logic is your full rule set. That, however, is not necessarily the best way to go and I will try to show you that here.

Consider another set of logic that is down-stream from the previous set of logic where, for example, result E is true when result C is true "OR" when trigger D is true. Notice

how you can now treat result C (which you got from the previous logic set) as if it were a trigger. First, process your variations and you end up with this:

Variation 4	Result E = TRUE	(when) Result C = TRUE	(and) Trigger D = FALSE
Variation 5	Result E = TRUE	(when) Result C = FALSE	(and) Trigger D = TRUE
Variation 6	Result E = FALSE	(when) Result C = FALSE	(and) Trigger D = FALSE

What you can now do instead of adding (or multiplying) your two logic sets is create routes. In other words, create a set of routes that branch through all of the conditional logic paths that exist in each variation set, as such:

Route 1: Variation 1 + Variation 4:

- trigger A “AND” do not trigger B [then]
- verify that the assignments in result C are true
- “AND”
- do not trigger D [then]
- verify that the assignments in result E are true

Route 2: Variation 2 + Variation 4:

- do not trigger A “AND” trigger B [then]
- verify that the assignments in result C are true
- “AND”
- do not trigger D [then]
- verify that the assignments in result E are true

Route 3: Variation 3 + Variation 5:

- do not trigger A “AND” do not trigger B [then]
- verify that the assignments in result C are false
- “AND”
- trigger D [then]
- verify that the assignments in result E are true

Route 4: Variation 3 + Variation 6:

- do not trigger A “AND” do not trigger B [then]
- verify that the assignments in result C are false
- “AND”
- do not trigger D [then]
- verify that the assignments in result E are false

You can see that you actually still have some redundancy and that makes a certain amount of sense. (Going back to the driving analogy, just as we must always go down your driveway, regardless of which direction we actually end up going down your street, there are certain parts of your application you may always have to hit.) Notice that the

two variations that you *were able* to eliminate in this example did let you avoid at least two redundant routes.

The point here is that when you normalize branching logic into "AND" variations, you can combine sets of variations with the use of addition, instead of multiplication. This eliminates that conditional logic monster called the *combinatorial explosion*.

Combinatorial explosions, in this case, happen because the formula is $2x$ where x is the number of triggers. So if your application has, say, 100 triggers, which is actually a fairly small number, then you have the explosive number of 2^{100} possible combinations: 1,267,650,600,228,230,000,000,000,000,000, to be exact. If instead, you add the triggers, the formula is $x + 1$ or 101 combinations. You can see that built into this procedure then is a form of all pairing.

Explicit Logic vs. Implied Logic

The remaining need at this point is for what is known as probe-point containment. This can be determined by asking an operational question: when must you absolutely know what the application is doing and when can you just stimulate whatever the application is doing externally?

You will need a probe-point when you *absolutely* need to determine if expected and actual results agree. In the example I just gave above with our two logic sets, you would need to probe C and E to make sure the conditional logic achieves the correct result from the given input triggers. Because of this, as long as you maintain reasonable control over your input trigger accuracy at A, B and D, you actually do not need to verify each trigger. By that I mean that you can safely infer that an accurate result followed from an accurate trigger. That sounds great, right? But there is a danger lurking there and that danger is that it is very possible to get the right answers but for the wrong reasons).

Without going into the detail of the many possible scenarios, you may find a situation where you must know, explicitly, if the specific triggering event took place. For example, if you could not trust a trigger because you had limited the downstream results you are verifying to the point where the result might be the same whether or not the trigger fired (for example, a different database entry is made and thus a different indicator is set but the returned Web page looks the same). You would then need to verify the trigger explicitly, as a separate task, before verifying the downstream results. In this example you would treat the trigger verification just like any other result - as a testcase.

In SilkTest, I found it best to use appstates to trigger any conditional logic and to use testcases to verify the results. Each result verifies that the application (or the relevant area of it) actually performs the triggers and all of the expected assignments associated with triggering each branch on the conditional logic tree. Using the rules, the inference engine will then traverse all routes, one route at a time, to verify that the combination of triggers and results in each route actually operate as expected.

So, going back to what I started with, this is the preliminary analysis and decomposition of the conditional and assignment logic. You essentially convert all such logic into appstate triggers, testcase results, variation sets and routing combinations. You must do this analysis if you want to build an inference engine as part of an automation framework. The nice thing is that you can tailor this analysis to any set of logic that you want to trigger and verify. You can exclude conditional logic for paths to which you do not want to branch. (For example, maybe development is not done on those portions of the application yet.) You can exclude assignment logic for triggers you do not want to press or results that you do not want to verify. The point is that you can simply exclude any logic you want from analysis.

The Inference Engine

Once you have done your analysis, like I covered in the last section, you are ready to apply the results of that analysis to your inference engine. Of course, that only works if you have an inference engine built in the first place. It is the "building" part that I want to talk about now. This is the part that shows the practical "how" to all the previous theoretical "what."

Speaking broadly but not inaccurately, an inference engine is a nice tool for verifying conditional logic and assignment logic because it is predicated upon rule sets that we can construe as "knowledge" of a given application. In terms of how this fits in with SilkTest, all you really need do is combine a few simple modifications to the built in recovery system in SilkTest with a (somewhat) unorthodox approach to constructing appstate and testcase procedures.

The inference engine I am describing has two important features. One feature is called goal-seeking and the other feature is called backward-chaining. In this context, goal-seeking simply means the ability to find a set of triggers that will achieve a given result. If, for example, you tell the application that you want to open a particular window so that you can verify results, then you want our goal-seeking facility to find those triggers that will open *that* particular window. Backward-chaining is the "how" part of the goal-seeking equation. The idea here is that your engine starts with the result that you want to achieve. Backward-chaining then goes back up the branches of the logic tree systematically, until it finds the first trigger that it must execute to achieve *that* result. The engine does this by going backward through each of the intermediate triggers that also have the possibility of achieving the result.

The key here is that backward-chaining must not break any of the rules you give it. Ideally, when constructing any inference engine, I want it to avoid useless, dead-ends that could never succeed. The inference engine I will be showing you can actually do this as a by-product of using the recovery system appstates to create a tree of triggers. While this sounds a little complicated (and it is, to a certain extent), what these appstates/trigger-trees do is actually very simple and their power is in that simplicity because, by its very

nature, the design of the engine matches the granularity of how you have decomposed your triggering and verification activities relative to the application you are testing.

But what about the recovery system? Here I refer to the built-in one that SilkTest ships with. Fortunately, goal-seeking and backward-chaining are basically what the recovery system already does. Unfortunately, the recovery system *always* starts from scratch. This means that if you have two or more goals (results), the recovery system will backward-chain and re-execute the entire tree for each goal. This is largely necessary because the recovery system has to handle failures that occur during its steps. The price is that all of this back-and-forth motion can be very time-consuming and, in some cases, resource-intensive. For an inference engine that is not really a tenable state of affairs.

In the case of building an inference engine, it is necessary to modify the recovery system so that it is possible to separate the execution control from the backward-chaining and goal-seeking control. Putting that in context, while you are within a particular route, you do not want it to keep skipping back and forth (akin to driving back the forth along same street, repeatedly, before moving to the correct street). So you need to make a modification that lets the recovery system record each leg of the journey already triggered for each particular route. This includes modifying the ancestor of all appstates, the DefaultBaseState in SilkTest.

The modification provides a way of conditionally executing the DefaultBaseState and all other appstates. The modification still lets the recovery system re-establish the DefaultBaseState for the first testcase in each route. What that means is that if the previous route fails, the next route can continue. Failure within each route may not cause total failure of that route. The idea is that with the inference engine recovery is at such a granular level that the particulars may depend on how exceptions should be handled. One possible approach, for example, is using do-except logic constructs.

Now, the modification of DefaultBaseState may not seem like a major concern to you. And, then again, it may. One thing I actually try to avoid, with any tool or library, is to change the inherent functionality of the tool or library. If I can override certain things, fine. If I can extend things, fine. I generally dislike outright changing things. And yet here I stand: by what I am proposing for the inference engine, changing things is what is absolutely necessary. In fact, the first modification that would have to be made is to the *defaults.inc* file that comes with SilkTest. (Obviously you should make a backup copy of the existing file.)

Modifying defaults.inc

What follows are the actual modifications to the defaults.inc file that I mention above. I am getting into the implementation stuff here. So, from this point forward, this document may not interest those who do not understand SilkTest code or have no desire to.

The steps to take: you will have to declare two new variables, a new function, and modify the DefaultBaseState procedure.

Here is the first set of code to put in defaults.inc:

```
[ ] LIST OF STRING lsExecutedAppStates = {...}
[ ] BOOLEAN bDefaultAppStateFlow = TRUE
```

Here the list of string lsExecutedAppStates is used to hold a list of the appstates that has already been executed for the current testcase chain. The boolean bDefaultAppStateFlow is a flag used to indicate whether default, SilkTest-derived DefaultBaseState execution should occur.

The next thing is create a function called ExecuteAppstate(). That function is shown here:

```
[-] BOOLEAN ExecuteAppstate(STRING sAppState, BOOLEAN
bDefaultAppStateFlow optional)
    [ ]
    [-] if (bDefaultAppStateFlow == NULL)
        [ ] bDefaultAppStateFlow = TRUE
    [ ]
    [-] if (bDefaultAppStateFlow)
        [ ] return TRUE
    [ ]
    [-] else
        [ ]
        [-] if (!ListFind(lsExecutedAppStates, sAppState))
            [ ]
            [ ] ListAppend(lsExecutedAppStates, sAppState)
            [ ]
            [ ] return TRUE
            [ ]
        [-] else
            [ ]
            [ ] return FALSE
```

What the code does:

Obviously this function returns a true or false. A true value is returned if a given appstate needs to be executed. A false value is returned if a given appstate does not need to be executed. The first part of this function checks if a default appstate flow applies, by checking the value of the bDefaultAppStateFlow boolean variable. If the else condition is reached that means a special (non-default) appstate flow applies.

When a special appstate is to be applied, a check is made to determine if that appstate is in the lsExecutedAppStates list. If the appstate is in the list that means it was already executed and the function will return false, because the appstate should not be re-executed. However, if the appstate is not in the list then that means it has not been executed yet. In this case, the appstate is now written to the lsExecutedAppStates list (so

that next time around it will not be re-executed) and the function returns true, indicating that the appstate in question should be executed.

What the code means:

The original appstate flow for the recovery system, as it comes built-in to SilkTest, is to re-execute the entire appstate chain for every testcase function that has an appstate applied to it. The ExecuteAppstate() function modifies this behavior so that SilkTest will only execute each appstate in a chain if that appstate has not already been executed for that entire testcase chain. The result is that the recovery system still traverses the appstate tree but it will only execute those appstates that are not yet executed in that chain.

It is now necessary to modify the DefaultBaseState appstate procedure in defaults.inc. The only modification here is to wrap the core elements of this procedure with a call to the ExecuteAppState() function just defined. In defaults.inc, under the DefaultBaseState() function, you will see a do...except construct. The only modification is as such:

```
[-] appstate DefaultBaseState ()
    [ ] ...
    [-] if(ExecuteAppState("DefaultBaseState", bDefaultAppStateFlow))
        [ ]
        [+] do
        [+] except
```

Creating Appstates

To put all of this in context, I will now show you how to create appstates in this new setup. Going along with the trigger idea that I mentioned earlier in this document, each appstate is a trigger that is based on its *immediately prior* trigger. In other words, the appstates represent the chain of triggers from the one that launches the application (nominally the DefaultBaseState) to the last trigger on every branch of the conditional logic tree. These appstates do nothing more than call the actual trigger function and any assignment functions needed.

The assignment functions should refresh expected values so that when the verification occurs in the calling testcase, you can do something to compare the proper expected values to the actual values. (For example, you might use the built-in Verify function or have some more elaborate construction of your own devising.)

So let’s consider a simple trigger appstate that uses the ExecuteAppState function I just showed you.

```
[-] appstate SCS_FilterSearch basedon Homescape_SCS
    [-] if (ExecuteAppState("FilterSearch", bDefaultAppStateFlow))
        [ ]
        [ ] GetSearchElements(rCurrentData, rFilterGroup)
        [ ]
```

```
[-] if (ApplySearchElements(rFilterGroup))
    [ ]
    [ ] Print("Trigger set True: SCS_FilterSearch()")
    [ ]
    [ ] // true state verification assignments
    [ ]
[-] else
    [ ]
    [ ] Print("Trigger set False: SCS_FilterSearch()")
    [ ]
    [ ] // false state verification assignments
```

What the code does:

The code first calls the ExecuteAppState function with the name of the appstate (“FilterSearch”, in this case). If the function call returns true, the appstate should be executed (as I described before) and if the function call returns false, the appstate should not be executed.

If the appstate is executed then, in the example above, a parameter is passed to a function identifying a group of data that will be used to filter a search.

Here this is hypothetical because, of course, it depends on the logic you have in place for SCS automation. I had a generic function called GetSearchElements() that would take in two records. One was where data should be read from (rCurrentData) and the other was where the data read should be stored (rFilterGroup). The end result is that rFilterGroup would contain a set of data that was used to filter a search (such as ["Venice", "Condominium Unit", "2 bathrooms", "2 bedrooms"]). Then the ApplySearchElements() function would take that filter group and try to apply it. If the ApplySearchElements() function could not use the data, for various reasons, it would return false; otherwise, it would return true.

So what happens here is that the trigger calls this ApplySearchElements() function. If the value returned is true, the trigger state is set to true and the “true/false state verification assignments” sections then indicate where verification statements can go.

(Note this would be the place to refresh the expected values that you will use in your verifications. This is a point I brought up a couple of times earlier in the document.)

What the code means:

Notice that the appstate is really fairly simple but also notice that it is doing more, in terms of direct logic, than many people often associate with appstates. This goes back to that conceptual shift that I mentioned is required if you want to utilize an inference engine.

Creating Testcases

Now let's consider a specific testcase.

```
[-] testcase VeniceCondo2Bed2Bath appstate SCS_FilterSearch
    [ ]
    [ ] // ... logic to generate search ...
    [ ] // ... logic to check search results page ...
    [ ]
    [ ] STRING sExpected = GetResultCount(rFilterGroup, iCount)
    [ ]
    [ ] LIST OF STRING sActual = SCSSite.Search.ResultsFound.GetText()
    [ ] STRING sVerify = "Results Count"
    [ ]
    [ ] Verify( sActual, sExpected, sVerify )
    [ ]
```

What the code does:

Before doing anything, the testcase calls the appstate SCS_FilterSearch. The appstate chain of triggers for that testcase brings the application to the point where expected values should match actual values. Nothing remains but to get the expected value, then run one of the built-in verification procedures. If it is safe to proceed down the path, even if the verification fails, you can wrap the verification in a do-except structure.

The idea is that each filter group (such as City = Venice, Type = Condo Unit, Bathrooms = 2, Bedrooms = 2) should give you a certain number of results that is indicated on the search results screen. It is the number of results returned that the testcase is checking.

The appstate makes sure we get to the correct city and search settings so that we can in fact check the appropriate results list for the correct count.

What the code means:

In other words, the appstate is telling your code that state you want SCS to be is one in which a search was executed such that the search was for all condominiums in Venice that had two bedrooms and 2 bathrooms. That is the appstate. The appstate will verify that it could drive the application to this state. The testcase then is just using this state to do what you actually want to know: check if the returned result count is accurate.

So note that it is possible for the appstate (or a chain in the appstate) to find a bug. For example, perhaps Venice does not show up in the city list. Or perhaps the bathroom fields cannot be selected because they are disabled. It is also possible for the testcase to find a bug: the search count is inaccurate.

Notice that if you had an appstate for every kind of search, you would have thousands of appstates. That is why I use a generic FilterGroup that can be read in at any time.

Results-Route Matrix

Now let's talk about creating a results-route matrix. One nice thing is that you can maintain a matrix of results and routes in any form you find useful. In table form, a results-routes matrix could look something like this:

Results (goals/testcases)	**Routes**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Include Route?	Y	Y	Y	Y	Y	Y	Y	Y	N	Y
Venice_Condo_2Bed_2Bath	T	T	T	T	T	T	T	T	T	T
Venice_Condo_2Bed	T	T	T	T	T	T	T	T	T	T
Venice_Condo_2Bath	T	T	T	T	T	T	T	T	T	T
Encino_SingleFamily	T	T	T	T	T	T	T	T	T	T
Encino_SingleFamily_Min	F	F	F	F	T	T	T	T	T	T
Encino_SingleFamily_Max	T	T	T	F	F	F	F	F	F	F
Encino_SingleFamily_MinMax	F	F	F	T	F	F	F	F	F	F
LosGatos_Any_RealAddress	F	F	F	F	T	T	T	F	F	T
LosGatos_Condo_RealAddress	F	F	F	F	F	F	F	F	T	F
Pacifica_Condo_2Bed_2Bath_Pets	F	F	F	F	F	F	F	T	F	F

The shading above indicates that verifying F (FALSE) results is optional. You can alter the statements in your testcases and in your testcase driver (which I will talk about momentarily) so that both TRUE and FALSE results will execute. In other words, in that case you want to prove the results are TRUE when we expect them to be TRUE and FALSE when you expect them to be FALSE.

However, in the testcase driver procedure, the driver will execute a testcase as a goal, *only if it expects results that are TRUE*. This is a matter of preference or necessity, depending on the nature of what you are verifying. I like to try to avoid stating conditional logic in negative terms as negatives are inherently more difficult to comprehend, especially nested negatives.

I showed you earlier in the document how to derive the routes, at least conceptually. How you do this automatically (and thus via code) is largely up to you. I used all pairing techniques for my approach and while I could have written these routines in SilkTest I instead opted initially for a Perl approach and then switched to Ruby.

The reason I even show you the results-route matrix like this is because (1) it is important to have one in mind [and you will; implicitly or otherwise] and (2) because it is what gives focus to the testcase driver, which I talk about next.

Testcase Driver

Now I can talk about creating the main testcase driver. The main() function driver is one function where the each route can be executed. In SilkTest, you generally either have a main() function that calls a bunch of other functions or you have a series of testcase functions in a file. Here is the code for this, assuming the main() approach:

```
[-] main()
    [ ]
    [-] for iCurrentRoute = 1 to iMaxRoutes
        [ ]
        [ ] Print("Current Route number: {iCurrentRoute}")
        [ ]
        [ ] lsExecutedAppStates = {...}
        [ ]
        [ ] bDefaultAppStateFlow = FALSE
        [ ]
        [-] if (IncludeRoute[iCurrentRoute]) == "Y"
        [ ]
            [-] for iCurrentResult = 1 to iMaxResults
                [ ]
                [-] if (routeMatrix.Read(
"{iCurrentRoute}{ResultIndex[iCurrentResult]}" ) == "T")
                    [ ]
                    [ ] Print("Current Result: {iCurrentResult}
{ResultIndex[iCurrentResult]}")
                    [ ]
                    [-] switch ("{ResultIndex[iCurrentResult]}")
                        [-] case "Venice_Condo_2Bed_2Bath"
                            [ ] VeniceCondo2Bed2Bath()
                        [-] case "Venice_Condo_2Bed"
                            [ ] VeniceCondo2Bed()
                        [-] case "Venice_Condo_2Bath"
                            [ ] VeniceCondo2Bath()
                        [-] case "Encino_SingleFamily"
                            [ ] EncinoSingleFamily()
                        [-] case "Encino_SingleFamily_Min"
                            [ ] EncinoSingleFamilyMin()
                        [-] case "Encino_SingleFamily_Max"
                            [ ] EncinoSingleFamilyMax()
                        [-] case "Encino_SingleFamily_MinMax"
                            [ ] EncinoSingleFamilyMinMax()
                        [-] case "LosGatos_Any_RealAddress"
                            [ ] LosGotsAnyRealAddress()
                        [-] case "LosGatos_Condo_RealAddress"
                            [ ] LosGatosCondoRealAddress()
                        [-] case "Pacifica_Condo_2Bed_2Bath_Pets"
                            [ ] PacificaCondo2Bed2BathPets()
                        [-] default
                             [ ] Print("Testcase not recognized.")
        [-] else
            [ ]
            [ ] Print("Route {iCurrentRoute} was skipped
intentionally.")
        [ ]
        [ ] bDefaultAppStateFlow = TRUE
```

What the code does:

The outer for loop keeps track of the routes. In the example above, based on my matrix, there are ten routes. General logging routines print the current route number, then re-initialize the lsExecutedAppStates list. The next step is to set the bDefaultAppStateFlow

flag to FALSE so that the ExecuteAppState() function will maintain an accurate lsExecutedAppStates list.

The first if construct in the example above is used to check if the current route should be included in the series of routes to process. Here the IncludeRoute() function is one I wrote that reads a matrix like the one I showed you. Having something like this in place is nice because it lets you exclude an entire route from the series. This gives you the ability to select a subset so that, for example, we could re-execute a route to verify a bug-fix or exclude a route until a bug has a fix or even because of data issues. (For example, perhaps the data entry that would put Encino listings in place has not been run yet.) If you skip a route, you can print a message to that effect in the logs with an explanation of why. In my matrix example earlier, I would exclude the ninth route from the series.

After this, the code enters a for construct that loops through all result testcases in the list. The next condition looks at each result to see if it needs execution within that route. General logging routines then print the number and name of the result. Finally, the switch statement is encountered.

The switch statement calls the testcase that is the current goal (result). After the script executes all result testcases and all of their corresponding appstates, the bDefaultAppStateFlow is *re-set* the TRUE. This is important because it tells the recovery system to close all open windows and take the application back to the DefaultBaseState in preparation for executing the next route.

What the code means:

The main() function essentially sets each goal within the route by peeling one testcase at a time from the list, examining whether it is true or false, then passing it to a giant switch statement that executes the testcase. Going back to the matrix, the driver starts with the first route at row 1, column 1, proceeds to the bottom row, then resumes with the second route in column 2, top row, and so on, until all columns and rows are done.

Obviously you can probably envision many other ways to do this and a main() function is not the only one. You could, for example, use a testplan as a driver, although I am not sure how well that would work.

The real point to note here this inference engine manages the order in which to seek goals. This approach improves speed and gives you almost complete control over goal-seeking behavior. More traditional or generalized rule-based expert systems are notoriously slow. Because this one is specialized to work with SilkTest relative to a specific application, I think it is clear that the performance is not a limiting factor.

Of course, that could still lead to the question: who cares? Or: why is this relevant? Well, control over goal-seeking gives us, for example, control over applications with bi-directional or circular window/page calling structures. This is where we can get to the same window/page from several directions (via different functional scenarios, for

example). Here, we could declare and verify those routes explicitly. Although not necessary, the switch statement lists the testcases in their relative position in the logic chain. This helps readability. In other words, like the appstates, we just put the testcases in the order in which verification needs to occur, first to last. That said, this is not a necessity.

Recapping the Logic

I threw a lot out there with this, so I want to try to summarize the basic flow of how things work, at least at a high-level.

The general idea here is the ExecuteAppState() function (now part of the new, modified recovery system) looks at the bDefaultAppStateFlow flag to manage a list of appstates that have already been executed. If the caller explicitly sets the bDefaultAppStateFlow flag to FALSE, then the ExecuteAppState() function checks to see if the calling appstate is in the lsExecutedAppStates list. If not, then the ExecuteAppState() functions adds the calling appstate to the list and tells the caller to go ahead and execute the appstate. If the calling appstate is in the lsExecutedAppStates list already (for that route), then the ExecuteAppState() function tells the caller to *not* execute the appstate.

Remember that the goals in each route are actually the execution of a chain of testcases and, further, a chain executed in the proper sequence. Each testcase chain represents one of the routing combinations through a set of conditional logic.

The out-of-the-box recovery system uses the appstates to re-establish a starting point for each testcase. The idea here is that ExecuteAppState() function can be used to conditionally execute an appstate. This gives you granular control over the recovery mechanism. You can then create an appstate for every trigger point in the application. Each of these appstate triggers may call more generalized functions based on categories such as object type, parent-child relationship, data sets or whatever else makes sense to reduce complexity and redundancy. For simplicity, the appstate does nothing except decide if the trigger needs execution or not. This keeps the rules represented by the recovery system clean, simple and relatively easy to maintain.

The benefit of this kind of design is that you remove all trigger chores from each testcase. Now the testcase can focus on verification exclusively. Then, because you have eliminated the back-and-forth “recovery actions” that would otherwise occur, the testcases can then have the same degree of granularity as the appstates. Each testcase within the routing set calls an appstate and each appstate calls the ExecuteAppState() function. The point there is that the ExecuteAppState() function eliminates redundant re-execution and tells the appstate whether it should execute its trigger. This eliminates the “back-and-forth” that manifests itself as the unnecessary opening, closing and reopening of application windows, as just one example, which you tend to get with the built-in recovery system.

The real point, for me, is that doing all this lets you organize your verification environment so that the *appstates run the input trigger procedures and the testcases run the results verification procedures*. This granularity and simplicity creates an extremely powerful and robust verification facility.