The Grey-Box Software Testing Method
The Functionality of Software Quality

Jerry Huth, Solid Step Software, Inc

***Abstract*— This paper introduces Solid Step Software's new approach to Software Quality Assurance (SQA), and discusses the author's experience using it in an application with a large Graphical User Interface (GUI). Unlike previous SQA approaches, Solid Step Test is used to build a specific functionality, Software Reproducibility, directly into the software to drastically reduce the costs and risks associated with the development and maintenance of software systems.**

***Index Terms*— Software Engineering, Software Quality Assurance, Automated Software Testing, Grey-Box Testing, Software Reproducibility, Playback Determinate Software**

I. Introduction

SOLID Step Software's new software testing product, Solid Step Test, constitutes a wholly new approach to the problem of Software Quality Assurance (SQA). Unlike previous SQA approaches, Solid Step's Grey-Box Software Testing Method embeds the functionality of automatic testcase creation directly into the software application to drastically reduce the engineering risks and costs associated with the regression testing, bug detection, error correction and functional verification of software systems. And unlike brittle black-box software testing techniques, Solid Step's testcases are always created automatically and never require any programming. Furthermore, Solid Step's patented reproducibility engine (US Patent 6,845,471 [1]) handles the general case of software processes, even applications with a complex Graphical User Interface (GUI) or asynchronous process interaction.

This paper discusses Solid Step's new SQA approach and details the author's experience using Solid Step Test in har*GIS Field Information Systems' "Truckmap" application. The main question the author wanted to answer was how well this SQA approach would work in a commercial application with a large GUI. As this experience shows, Solid Step Test has proven to be an extremely effective, efficient and practical approach to the software quality problem.

II. Ensuring Software Quality

In the 60 or so years since the dawn of the modern computer age, a daunting problem that early developers of complex computer software systems faced, and that software developers still face today, is the assurance of software quality. Maurice Wilkes, who led the team that built one of the first stored-program computers at Cambridge University in the 1940's, recounted years later that "people had begun to realize that it was not so easy to get a program right as had at one time appeared." He even remembered the precise moment when "the realization came over me with full force that a good part of the remainder of my life was going to be spent in finding errors in my own programs." [2, p. 5]

One of the finest recent papers on the topic of SQA, "What Is Software Testing? And Why Is It So Hard?" [3], was written at approximately the same time that Solid Step Test was being developed. As little has changed in the SQA field since then, this paper remains one of the best descriptions of why ensuring software quality is a hard problem, and it provides an excellent overview of the state of the art in SQA tools and practices.

Because SQA is such a hard problem in general, as that paper explains, over the years people have explored a complex array of methodologies and technologies in a bid to develop ways to aid the assurance of software quality, and in general to make software development easier. Despite these efforts, however, no one has found a way to build quality, or something that helps make quality much easier to ensure, directly into the software. And although people have explored various ways to somehow build quality into the software development process, in a bid to help raise the quality of the resultant software, these initiatives also haven't produced any clear SQA breakthroughs.

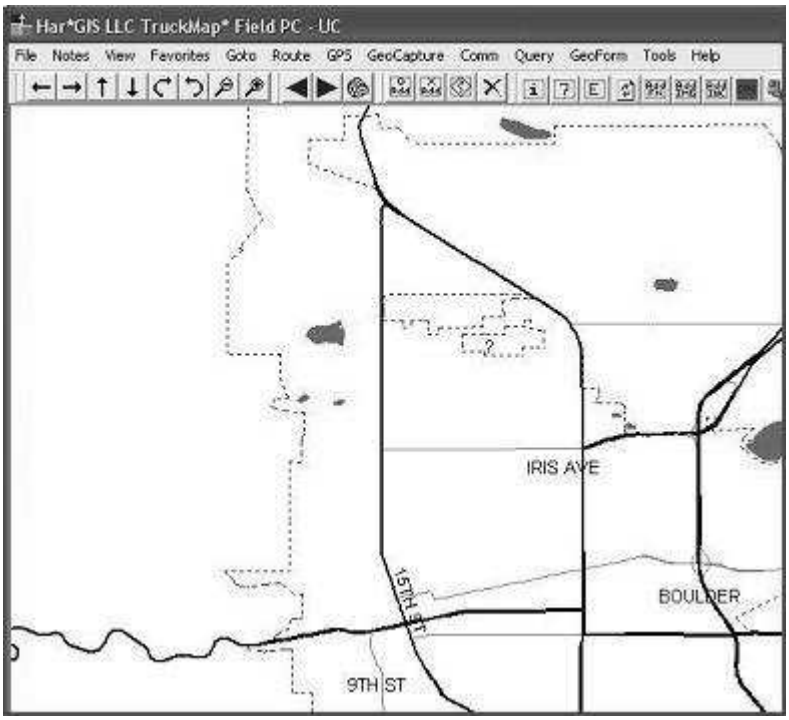

Fig. 1. Truckmap's Main Window

But Solid Step Software's new SQA product, Solid Step Test, represents just such a clear and effective breakthrough. It provides a specific functionality that is included in the software application - built directly into the software - that makes software quality assurance a radically easier task. It leverages the power of the computer to automatically deliver the most important data for ensuring software quality: The Software Testcase.

Solid Step Test gives developers an easy way to build into the software application the functionality of Reproducibility, or Automatic Testcases: the Ability to Automatically Create Testcases Whenever the Software is Run. And unlike the current state-of-the-art in SQA technology (i.e. brittle black-box software testing techniques), Solid Step's testcase "language" is not actually a programming language at all - it contains no control statements like "if", "while", "goto", etc - so individual testcases never require programming.

A. Embedded Graphical User Interfaces

Although Solid Step Test works for the general case of software processes - meaning that it can enable Reproducibility in any software process - one of the main kinds of software it was originally developed for is software with an *embedded GUI*. Applications with an embedded GUI include office software, like word processors, spreadsheet editors, stand-alone email readers, etc; computer-aided design software, like mechanical drawing tools, simulation packages, etc; video games; database applications with large stand-alone GUI's; web browsers; etc.

One of the main questions the author wanted to address when using Solid Step Test in har*GIS' application was how well this SQA approach would work in a commercial application with a large embedded GUI. har*GIS' "Truckmap" application, a mobile mapping application with a very large GUI, is a great example of just such an application.

Fig. 1 shows Truckmap's main window, which displays an area map with various data, such as roadway information, utility equipment locations, etc, overlaid on the map. Truckmap's users include construction crews, utility workers, etc, who are on the road a lot during the course of their work.

Users can create their own custom data to be overlaid on the map, which is supported by the menus and dialogs of the application's GUI. The kinds of data that can be created in this way include what are called *geonotes*, *fieldnotes*, *incident reports*, and *map sketches*. Fig. 2 shows a fieldnote (dark box at bottom left) and the corresponding Fieldnote dialog.

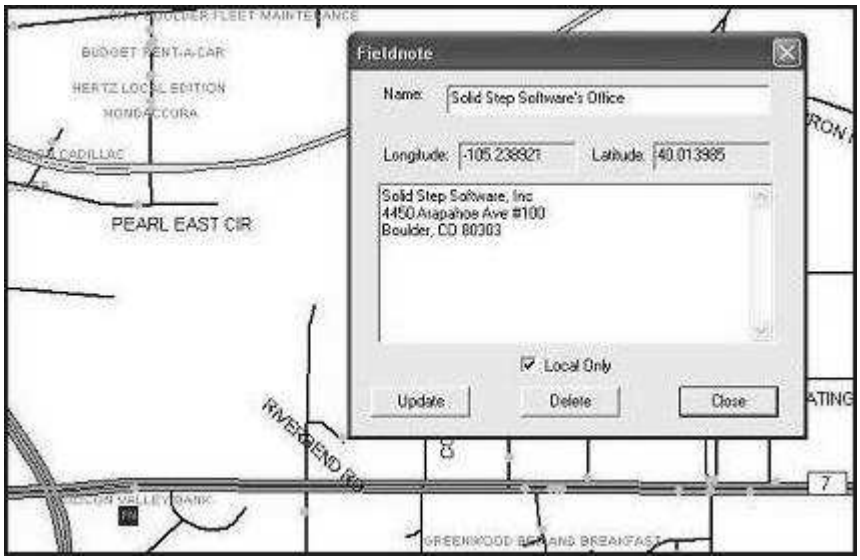

Fig. 2. Truckmap's Fieldnote Dialog

III. The Grey-Box Software Testing Method

Black-box software testing products have been available for a great many years, but as har*GIS' engineers found when they tried using them with their application, they suffer from a number of problems that greatly limit their usefulness in practice. Namely, black-box testing techniques are quite labor intensive because they tend to be very brittle, and in general require programming on a per-testcase basis (as evidenced by the full-featured programming languages that black-box products use for their testcases).

Because of the problems inherent with black-box testing, har*GIS' engineers had been running a manual test suite on their application. Although black-box techniques offer the promise of a certain measure of automation in the playback of testcases, there was little evidence of longterm manpower savings due to the lack of automation in testcase creation, because of the per-testcase programming that's required in general, and the maintenance costs associated with the brittle nature of black-box testing.

In stark contrast to brittle black-box testing techniques stands Solid Step's new SQA product, Solid Step Test, which uses a testing approach known as *The Grey-Box Software Testing Method*, for which Solid Step's trademark name is *the grey-box solution*^{TM}.

The main idea behind this approach is the following: by "covering" the main interfaces of the application - adding 1 or a few Lines of Code (LOC's) at the interface points to make calls into Solid Step's Reproducibility Libraries - Solid Step Test can automatically create testcases whenever the application is run. Then at any later time Solid Step's patented Reproducibility Engine coordinates all the actions in the testcase to automatically replay that exact run of the application.

The major advantages of this grey-box testing approach are that the testcases never require any programming, and it handles the general case of software processes, i.e. even applications with Complex GUI's or Asynchronous Processes.

A. Solid Step's Reproducibility Engine

To explain how Solid Step's patented reproducibility engine works, it's helpful to look at a precursor capture/playback engine that was in production use at Synopsys, Inc, as early as the late 1980's/early 1990's. Their flagship product had a large X Windows GUI that used a simple capture/playback engine to save and replay the GUI events that occured during the original run of the software. Jerry Huth was on the team that developed it, and the original idea to save and replay the GUI events can be attributed to the group's manager and technical lead, William Krieger, who was also a co-founder of Synopsys.

Fig. 3 shows that early playback engine. Because it's just a simple sequential loop that fires GUI

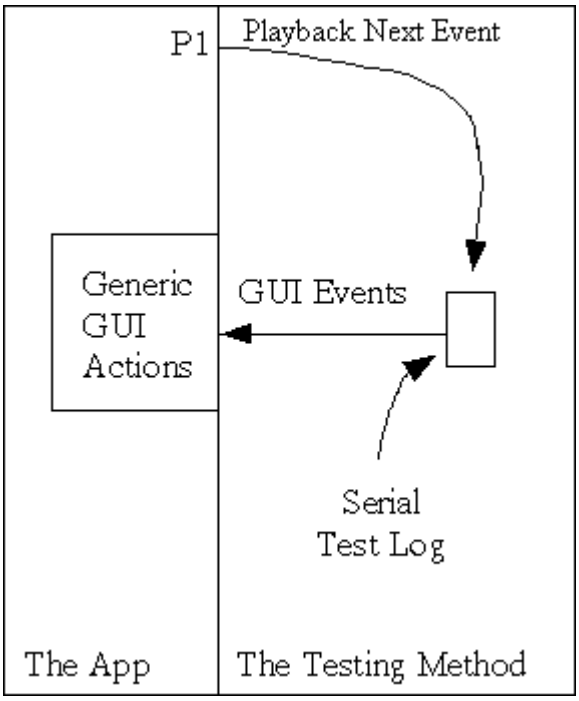

Fig. 3. Built-In GUI Event Playback

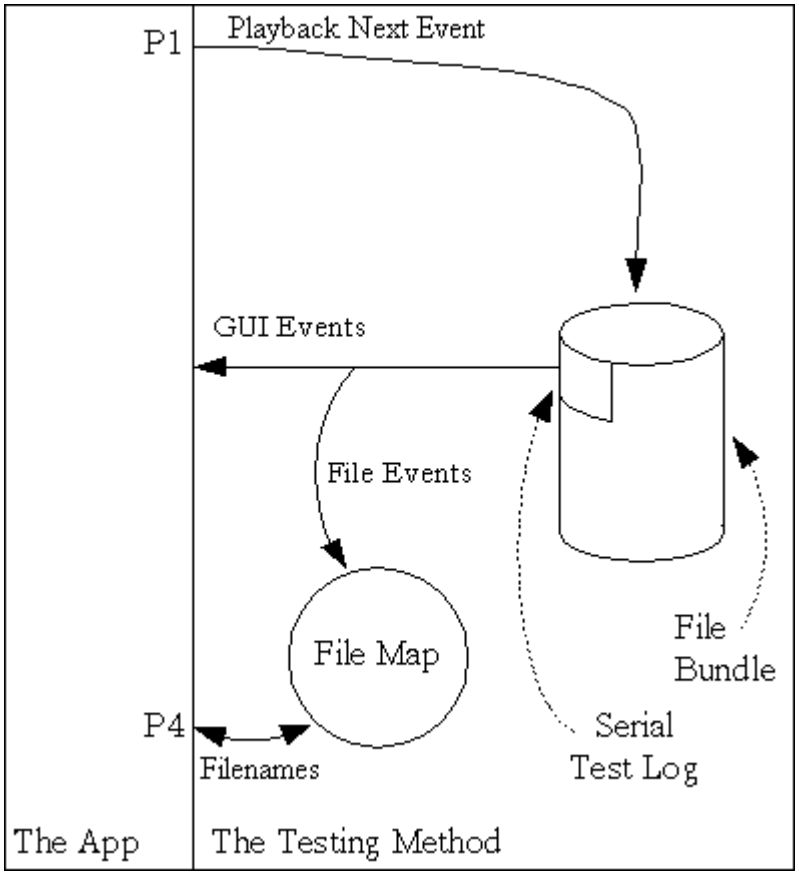

Fig. 4. Handling File Data During Testcase Playback

events at the application until none are left, it can only enable completely automatic reproducibility in the simplest kinds of software, i.e. applications that have no asynchronous interprocess communication and which don't read/write files. Synopsys' application certainly wasn't that simple, so the testcase log file (the sequential list of GUI events) generally did not by itself constitute a complete testcase - typically a human had to gather together and/or edit some files, including the testcase event log file itself, to create a complete testcase.

Although this simplistic version of the capture/playback engine has some obvious limitations, it proved to be an extremely reliable testing approach due to the fact that it was built directly into the application in the same way that Solid Step Test is built in. In fact, it was the primary testing platform for many production releases of Synopsys' software.

In later years however, as Synopsys moved away from the X Windows GUI platform toward MS Windows, and probably also due to the fact that neither Krieger nor Huth were at Synopsys during this transition period, that testing approach was never implemented in Synopsys' later applications that used the MS Windows GUI platform.

B. Automatic Testcase Creation

The principal design goal of Solid Step Test is to give the programmer an easy way to include the functionality of "completely automatic testcase creation" in their application. This means that no human has to gather together files, edit files, program test scripts, etc, or in any other way be required to expend effort to assemble the complete testcase.

Thus it is Solid Step Test's job to automatically save with the testcase any and all data required to run that testcase, and it is the job of Solid Step's patented reproducibility engine to coordinate all of the testcase data to replay that exact run of the software application.

C. File Data

A very important kind of data that must be stored with the testcase to enable reproducibility for applications that read or write data files is, of course, the data from the files. This also represents the first major step in the progression from the simple GUI capture/playback engine to Solid Step's generalized reproducibility engine.

Input file data is handled like this: During testcase creation, i.e. during the original run of the software application, input data files are copied into the file bundle that accompanies the testcase event log file. Additionally, a "file mapping event" is stored in the event log file. As illustrated in Fig. 4, the file mapping events are used during testcase playback to update the entries in the "file map", so that at any given time during testcase playback, the file map's entries can be used to serve up file mappings via function P4 that point the application into the file bundle to find the input files it needs to read. In a similar fashion, output data files are also stored in the file bundle to facilitate easy data verification during testcase playback, as described in the "Case Study" section below.

Database files are handled similarly to other input files, but with a twist to accomodate the fact that they are updated as the software runs. This is also explained in the "Case Study" section below.

D. Interprocess Communication

The next major step in the development of Solid Step's generalized reproducibility engine is the addition of the "gate" and its associated control logic, which gives Solid Step Test the ability to handle communication between software processes, particularly asynchronous interprocess communication. If not handled correctly, asynchronous events such as these lead to software indeterminacies that result in incorrect testcase playback.

Fig. 5 shows Solid Step's complete Reproducibility Engine. As the testcase is being played back, the reproducibility engine is informed via function P2 when asynchronous events occur. If an asynchronous event is expected at a certain time, but it hasn't happened yet, then the Gate closes and testcase playback is halted until the event actually happens. On the other hand, if an asynchronous event happens earlier than it is expected, then the reproducibility engine stores the event information until the event was expected to happen, i.e. it waits until the corresponding event in the serial test log is played back before processing the asynchronous event.

A sub-genre of interprocess communication is the "human action", which closes the gate during testcase playback to wait for a human to perform some specific task like loading paper into a printer, etc.

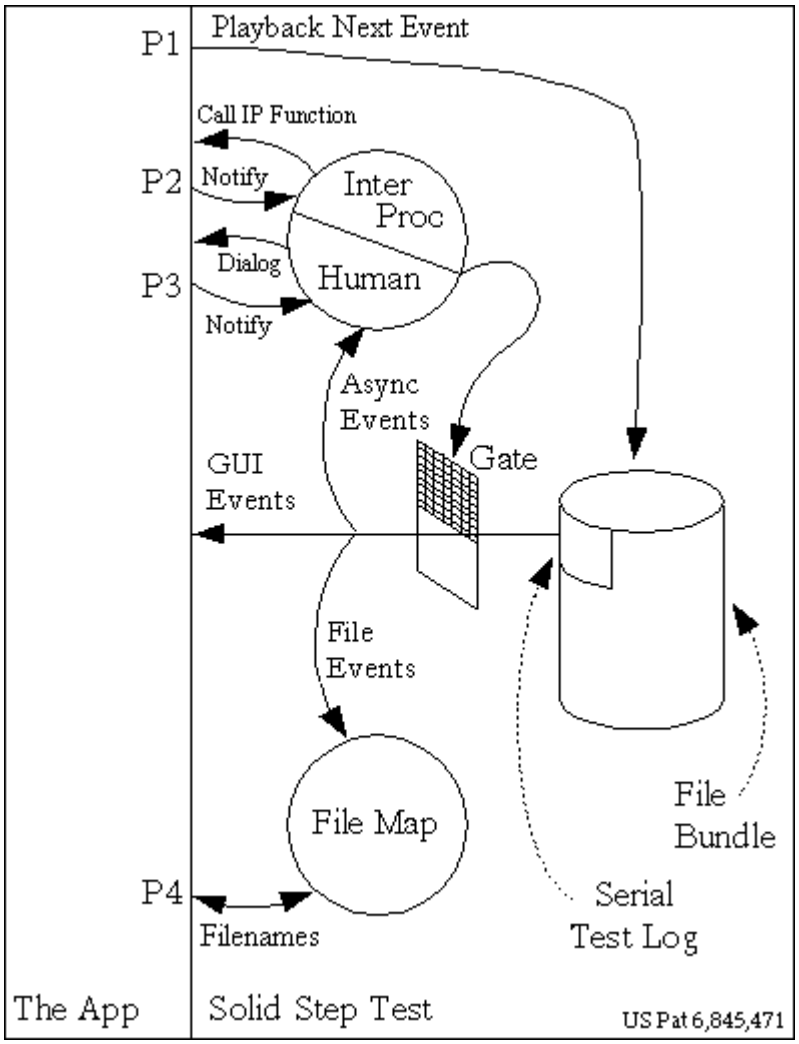

Fig. 5. Solid Step's Reproducibility Engine

Fig. 6 is an illustration of the serial test log, or event log, which stores the events that occur as the software is being run. For an application with an embedded GUI like Truckmap, the main process that runs the GUI naturally serves as an event arbiter, since it processes events in a serial fashion, so it can create the event log as it processes the GUI and other events.

Of particular interest is how the asynchronous events such as the Interprocess Communication and Human events control the Gate in the Reproducibility Engine to make the software application run correctly during testcase playback. In effect the serial test log acts as a sort of timing strip that opens and closes the Gate during testcase playback to keep the software processes in sync in the same way they were during the original run of the software. In this way, Solid Step Test can enable reproducibility not only in individual processes, but also in multi-process software systems.

E. The Playback Function

In the simple precursor playback engine, the playback process is essentially just a loop - it simply plays back events until there are none left. Fig. 7 shows the flowchart for the simple engine's playback function, it just always plays back the event and has no generalized knowledge about reproducibility.

By contrast, as shown in Fig. 8, the playback function at the heart of Solid Step's reproducibility engine knows how to handle all the events - the synchronous, asynchronous, file and human events - and in general encapsulates all the knowledge required to make any software process completely reproducible.

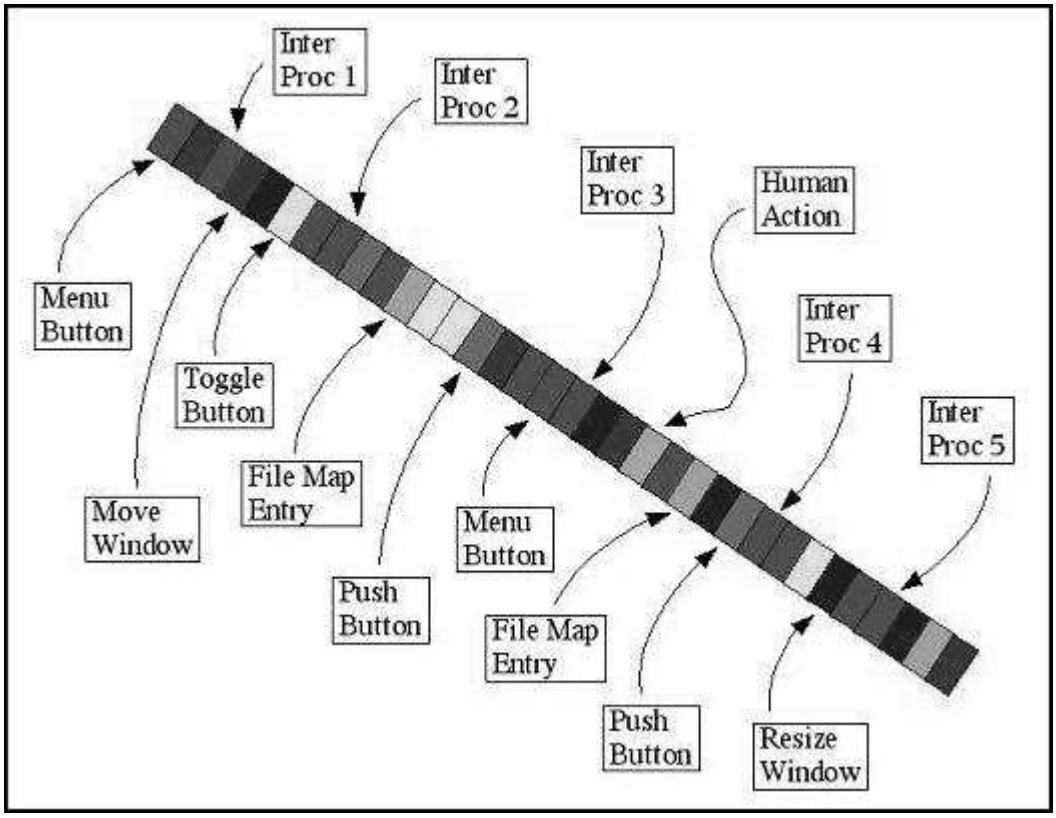

Fig. 6. Serial Test Log

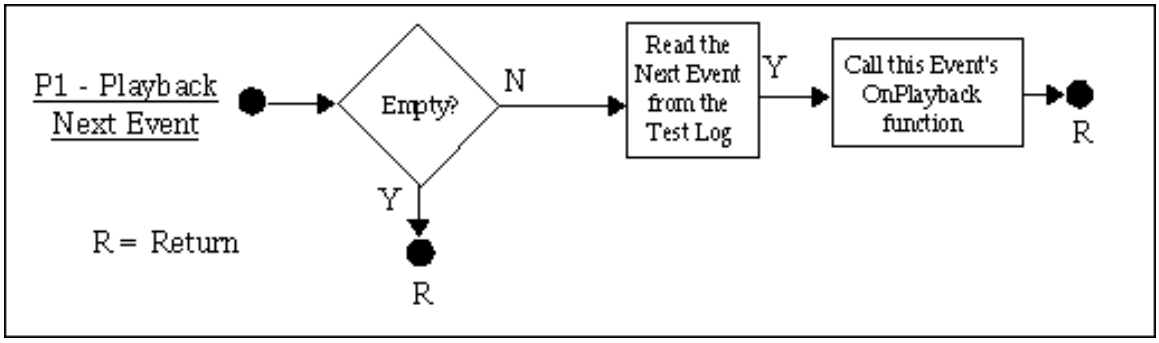

Fig. 7. Simple Playback Function

The first decision box in Fig. 8, "Gated or Empty?", simply means that if the Gate is closed, or if there are no more events in the serial event log file, then do nothing and return. The second decision box means that if the event is not a File, Human or Asynchronous event, i.e. if it is a basic Synchronous event, then simply call that event's playback function. The next two decision boxes tell what to do if the event is a File or Human event.

The last decision box, which is triggered only for non-Human Asynchronous events, tells what to do for the two possible cases (i.e. the asynchronous event either has or has not been received yet). If the event has not been received yet, then the Gate closes so the software process will wait for the event to happen. If the event has been received already, then the playback function for that event is called and the process continues running.

IV. Playback Determinate Software

There are a number of factors that can cause software to be indeterminate (to exhibit unpredictable behavior). Since the purpose of Solid Step Test is to make the software reproducible, which implies predictability, it is instructive to look at the various

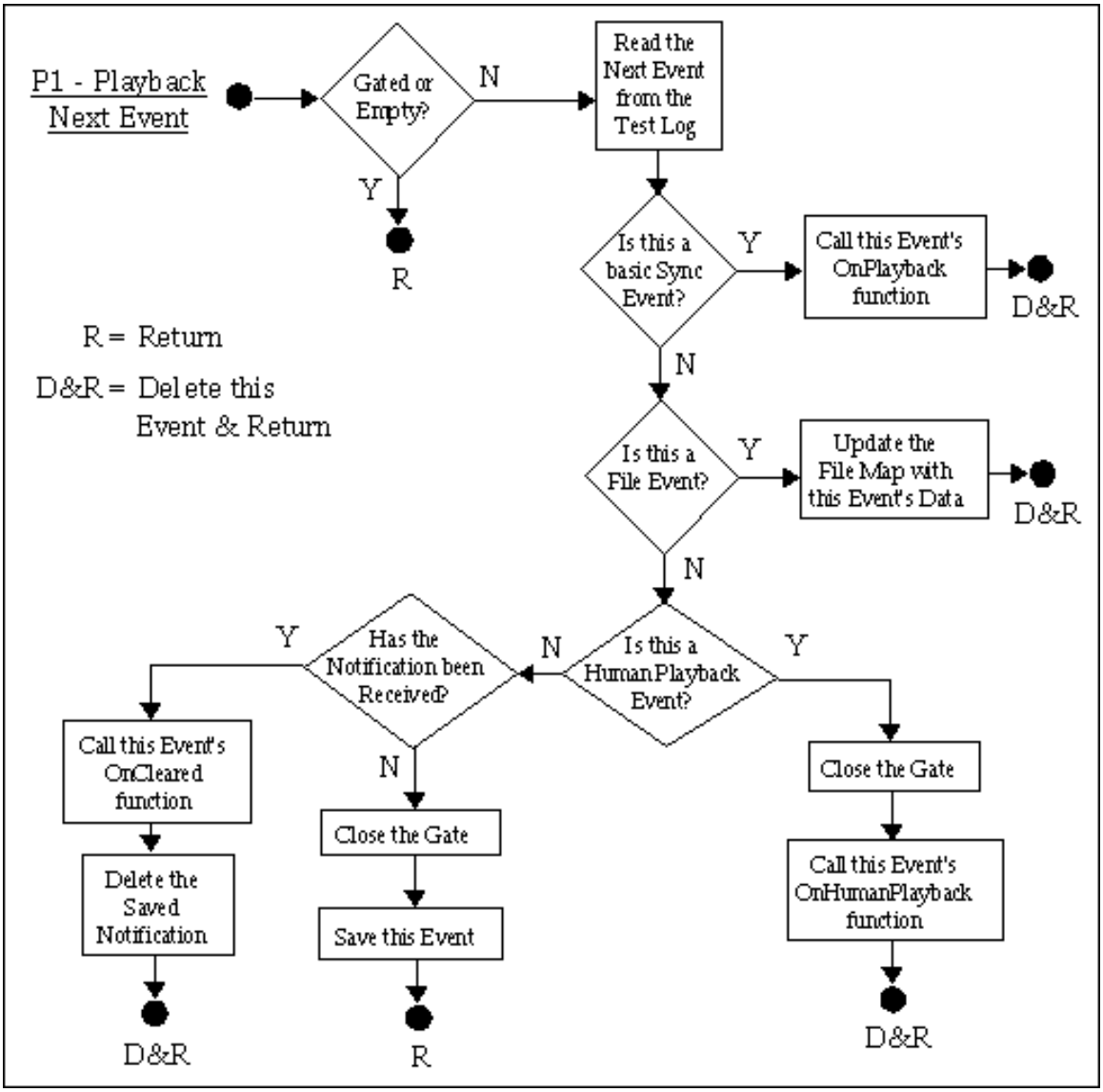

Fig. 8. Solid Step's Playback Function

types of software indeterminacies to understand how Solid Step Test handles them.

Excluding catastrophic system errors such as out of memory conditions, operating system crashes, etc, all software indeterminacies fall into one of three categories. The first category of software indeterminacy is when the code, for no good reason, does something that makes the software indeterminate. For instance, a generic module may sort data using pointer values. This type of indeterminacy is easy to fix by changing the code to remove the indeterminacy, for example to make it stop using pointer values in an unpredictable manner.

The second type of indeterminacy occurs when the software is actually designed to be indeterminate, such as when an algorithm uses random numbers. One easy way to handle this situation is to save the random numbers with the testcase, although depending on the application there may be other ways to handle it as well.

The third type of software indeterminacy is caused by asynchronous process communication. If not handled properly, multiprocess interaction of this sort can cause variations in testcase playback. This last category of indeterminacy is handled by Solid Step's patented reproducibility engine, as discussed in the previous section.

In the case of the second and third categories of software indeterminacy, it's interesting to note that although the software may be indeterminate when the user originally runs the application, it has nevertheless been made to be determinate for the

purposes of playing back the runs (testcases) of the software. In this case, the software is said to be *Playback Determinate*.

V. Case Study: Using Solid Step Test with C++/.NET

In this section the author discusses how Solid Step Test (Sst) was used in har*GIS' application, with code excerpts to illustrate the various ways that the interfaces are covered in a large GUI application that uses the C++/.NET platform. (See appendix for complete code excerpts.)

A. Object Registration

During testcase playback, messages are sent to the GUI objects to replay the individual GUI events (or actions) that are saved in the serial test log. Examples of objects receiving these messages include classes derived from CFrameWnd, CWinApp, CView, etc. When these objects are created, they are registered with the SstPlayer (the capture/playback engine) to give each object a unique identifier. For instance, Truckmap's CView-derived class is CTruckmapView, and the following Line of Code (LOC) was inserted in its constructor (ctor) to register it with the Sst libraries:

```
CTruckmapView::CTruckmapView()
{
  ...
  SstPlayer.RegisterType( this,
                 "TruckmapView") ;
```

And the following LOC was added in the object's destructor (dtor) to unregister it:

```
CTruckmapView::~CTruckmapView()
{
  ...
  SstPlayer.Unregister( this) ;
```

All GUI objects that receive notifications from Sst are registered in a similar manner.

B. Menu Items

Menu items are typically covered by adding 1 LOC in the callback function for the menu item. For instance, in Truckmap, the "Pan Left" menu item is covered like this:

```
void CTruckmapView::OnPanleft()
{
  SstCmdTarget( this, ID_PANLEFT) ;
```

In this implementation, SstCmdTarget is a class, not just a function. This LOC instantiates an object of that class, whose ctor stores the data in the serial test log. During testcase playback, the OnPanLeft() function is triggered when the SstCmdTarget class' OnPlayback() function sends the ID_PANLEFT message to the CTruckmapView object.

C. Keystrokes & Button Clicks

Keystrokes and button clicks are handled in a similar manner to menu items, i.e. by adding 1 LOC in the callback function:

```
void CTruckmapView::
        OnLButtonDown(UINT nFlags,
                      CPoint point)
{
  SstLBD( this, nFlags, point) ;
```

As in the previous example, SstLBD is a class whose ctor saves the data in the serial test log.

D. Window Position and Size

To handle changes to a window's position or size, Solid Step Test provides the SstMoveWindow class, which saves a window's position and size in the serial test log. However, since the .NET framework can handle changes to a window's position and size automatically, these events were not represented explicitly in Truckmap's code.

In order to cover these events, Truckmap's main window's OnMove() and OnSize() virtual functions were overridden and LOC's added to use the SstMoveWindow class:

```
void CMainFrame::OnMove(int x, int y)
{
    CFrameWnd::OnMove(x, y);
    SstMoveWindow( this) ;
}

void CMainFrame::OnSize(UINT nType,
                        int cx,
                        int cy)
{
```

```
        CFrameWnd::OnSize(nType, cx, cy);
        SstMoveWindow( this) ;
    }
```

E. Simple Dialog Boxes

Simple dialog boxes, which do little or no processing until the user closes it, are typically handled by adding 1 LOC in the ctor and dtor, and 1 or a couple LOC's in each callback function that can close the dialog. This example shows how the Fieldnote dialog is covered. The dialog, shown above in Fig. 2, has three buttons that can close it: an "Update" button, a "Delete" button and a "Close" button.

Of particular interest is the "Update" callback function, which not only saves the event in the serial test log, but also saves the current values of the text boxes and checkboxes of the dialog in the test log. This is required because the "Update" callback function uses the values of those GUI elements to modify the internal state of the Fieldnote object, while all the other callback functions for this dialog simply react to the event regardless of the values of the individual GUI elements of the dialog. (See Appendix for example code.)

F. Complex Dialog Boxes

More complex dialog boxes, which do a significant amount of processing as the user interacts with the individual GUI elements of the dialog, may need those GUI elements to be covered individually. For instance, Truckmap has a dialog named "CGNMgrDlg" (short for Geo Note Manager Dialog), which lists all the various kinds of geonotes the user has created. If the user clicks on a geonote in the list, the callback function for the list element fills in the other fields of the dialog with the specific values of that geonote.

To make this dialog reproducible, it was necessary to be able to record and play back the list selection actions. This was done by adding a new action class specific to this dialog, which when played back caused that callback function to be called. Also, the public function CallOnSelchangedFunction() was added to the CGNMgrDlg class since the OnSelchangeMpsktlistCtr() function was not public. (See Appendix for example code.)

G. Toolbars

Toolbar move events present a similar problem as the window move/resize events discussed above, namely that toolbar move events were not explicitly represented in Truckmap's code because the .NET framework handled those events internally. But unlike the case of window move/resize, the class that receives the events that need to be saved in the testcase also was not directly represented in Truckmap's code. So to handle toolbar move events, a subclass of the .NET toolbar was added to Truckmap, and the OnWindowPosChanged() virtual function overridden to handle the move events. The main window's toolbars were changed to use the new class, and the SstDockState class was used to save the position of all the toolbars. (See Appendix for example code.)

H. AfxMessageBox

The AfxMessageBox facility provided by .NET, which gives the application programmer an easy way to interrupt the application and give the end user a simple message, or ask a simple question, could have been covered in a couple different ways. The development team considered implementing their own message box functionality, which would have allowed them to make the message boxes completely reproducible, i.e. to make the message boxes appear during testcase playback.

But another much simpler way to cover this message box functionality was tried, and it ultimately worked so well that there seemed to be no need to replace .NET's message box functionality. This approach involved using .NET's message box virtual function feature to avoid displaying the message boxes during testcase playback and instead simply return whatever value was returned during the original run of the application. (See Appendix for example code.)

I. Simple Data Files

"Simple" data files, such as picture files that are used by a drawing program, are files that are only read and written at specific times, i.e. when the user directs the program to read or write the files. These files are covered by inserting a call to Solid Step's reproducibility libraries at the point where the file is read or written.

Although in general only input files need to be covered to enable reproducibility, in practice output files are often covered in the same manner to facilitate easy data verification. In both cases, during the original run of the program, Solid Step Test copies the file into the testcase's file bundle. Then during testcase playback, input files are read from the file bundle so the correct input file data is used by the application. Also during testcase playback, Solid Step Test redirects the output files to a special output file area and the output data is automatically verified against the output files stored in the testcase file bundle.

Although during testcase playback the application "thinks" it is reading input file data and writing output file data to and from the same file locations it was using during the original run of the program, Solid Step Test actually redirects the locations to files in the file bundle or in the special output file area. In this way, Solid Step Test lets the program run during testcase playback in what is essentially a "sandbox", so that it runs as it originally ran but without requiring the input files to be in their original locations and without causing the original output files to be overwritten.

Although Truckmap does not use simple input data files (it uses database files for all its data - see the "Database Files" section below), it nevertheless dumps the database fields to simple output files to make use of Solid Step's easy data verification functionality (see the "Data Verification" section below).

J. Database Files

In contrast to simple data files, database files are continually read and written as the application runs. Covering database files is done in essentially the same manner as simple data files, i.e. by adding a call to Solid Step's libraries at the place where the file is used in the application.

Like simple data files, database files are copied into the file bundle. During testcase playback, however, another copy of the file is made - a "sandbox" version - which the program reads and writes as the program runs. This lets the program run during testcase playback in the same way that it ran during the original run of the application, but without reading or modifying the files from their original locations, and without modifying the files in the testcase file bundle - i.e. as noted above, the program runs in a sandbox provided by Solid Step's reproducibility engine. (See Appendix for example code.)

K. The Windows Registry

For applications that use the .NET platform, another very important interface that must be covered to enable reproducibility is the Windows registry. As with other interfaces, it is covered by adding calls into Solid Step's libraries at the points where the Windows registry is accessed. The code example shows how the SstReg class, which provides functions with profiles similar to the registry functions originally used in Truckmap, is used in place of those functions. (See Appendix for example code.)

In this way the data returned from the Windows registry is saved with the testcase and during testcase playback that saved data is returned to the application so that it runs in the same way that it ran during the original run of the application.

L. Testcase Creation

When the interfaces of the application are sufficiently covered by calls into Solid Step's libraries, then Solid Step Test saves all the data in a testcase as the application runs, and at any later time Solid Step's reproducibility engine can coordinate all the data to replay that exact run of the application. In this way, Solid Step Test provides an "Always On" functionality that works behind the scenes to create testcases whenever the application is run.

There are a number of ways that Solid Step Test can be used in an application. In the most likely scenario, Solid Step Test can be configured to overwrite the previous testcase if the application had exited normally with no apparent errors. If the application exited abnormally, however, then Solid Step Test could automatically save the testcase for later evaluation by the application programmers. In another scenario which may be useful in certain situations, for instance during trial testing periods, Solid Step Test could be configured to save all the testcases, or to save a random sampling of testcases.

Another very common scenario when testcases will need to be saved is when an application programmer/tester is creating a functionality or regression test suite. In Truckmap a special dialog was added for this purpose, as shown in Appendix Fig. 1, and it lets the user specify the testcase name

and allows them to include a description of the functionality that the testcase is meant to verify.

M. Data Verification

Although Truckmap's output data is stored in a database file, since Solid Step Test includes an automatic data verification feature for regular (non-database) output files, code was added to dump Truckmap's data to regular files. Then when testcase playback is finished, Solid Step Test automatically verifies the data against the data stored with the testcase. (See Appendix for example code.)

The dialog in Appendix Fig. 3 lets the user choose which data to dump out and also lets the user insert a "Pause" into the testcase event log.

This example also illustrates how some kinds of data, such as the floating point numbers that represent latitude/longitude values, may not always need to be saved with the testcase. This is useful since testcases whose purpose is to verify functionality unrelated to the exact floating point values might be unnecessarily brittle if the floating point numbers were always saved with the testcases.

N. Case Study Discussion

The focus of this project was to see how well Solid Step Software's new SQA approach would work in a large commercial application with an embedded GUI. So even though some parts of the Truckmap application were beyond the scope of this project, such as the GPS features, its very large GUI provided an excellent opportunity to explore how this approach would work in a large software system that uses a leading development platform such as .NET. As this experience has shown, Solid Step Test works very well in practice with a full-featured platform like this since by its nature the platform provides all the hooks, callbacks, etc, needed to allow the application programmer to insert the calls to Solid Step's libraries required to make the application reproducible.

For instance, although the interfaces that need to be covered by calls to the Solid Step libraries were usually readily available in Truckmap's code, in some cases they were not represented in Truckmap's existing code. But even in those cases, the interfaces were easily exposed, either by overriding virtual functions in existing subclasses, or by creating subclasses and then overriding the virtual functions, as was done to cover the toolbar events.

It is expected that other full-featured platforms, such as Sun's Java platform, or the X Windows platform, etc, would also work well with Solid Step Test since by their nature as full-featured platforms they would include all the hooks, callbacks, etc, required to properly expose any interfaces that need to be covered by calls to Solid Step's libraries.

VI. What Is Software Quality?

As Jacco Wesselius and Frans Ververs pointed out in their landmark paper on "software quality control" [4], quality is a pervasive part of humans' everyday lives, yet its exact meaning seems to be somewhat fuzzy. When asked for a definition, "many people will remain silent or will vaguely utter things like 'excellence', 'reliability', 'it cannot be defined', 'everyone knows', etc."

Their paper surveys a variety of issues related to product quality in general which explain why it's such a hard concept to nail down. Examples include the difference between "quality of design" and "quality of conformance" (i.e. luxury vs economy), which stem from the fact that ultimately the definition of quality depends on the user (observer), and the reality is that even a single person's perception of quality can change - indeed is likely to change - over time.

With the aim of bringing some order to the ambiguities - some would say fundamental paradoxes - surrounding the nature of product quality in general, and especially the nature of software quality, Wesselius and Ververs define the "Three Components of Software Quality":

1) Objectively Assessable Component
   Criteria which lend themselves to objective verification
2) Subjectively Assessable Component
   Criteria requiring subjective validation
3) Non-Assessable Component
   Criteria, such as "future needs", that can't be immediately evaluated

In the search for better software quality assurance (or "control"), one important idea they put forth is to expand category 1 so that as much criteria as possible becomes objectively assessable. Another

very important idea is to use frequent feedback mechanisms, such as with the spiral process model, to lower risk as much as possible in the other components of software quality.

A. Automation & Preparedness

The most obvious benefit that Solid Step's new grey-box testing approach has for the software development process is in the way that it enables virtually complete Automation of testcase creation and playback, which drastically lowers the cost of software testing. This represents a major advance over black-box testing, which in general requires programming on a per-testcase basis, and which is difficult to use in practice because of the brittle nature of black-box testing techniques. It also has a major advantage over most other SQA approaches because it facilitates system-wide, real-world testing of the application. Furthermore, it's not limited to any particular kinds of software errors, as many SQA approaches are, since it offers the ability to reproduce any and all runs of the software system (excluding catastrophic errors like operating system crashes, etc).

But beyond the obvious advantage that such extremely thorough and robust testcase automation has for reducing the cost of software testing, another perhaps equally important benefit Solid Step Test has for the overall software development process is through its facilitation of Preparedness, by virtue of its "Always On" testcase creation. (Although this functionality is a feature that can be turned off, when enabled it is always working in the background.) And since it is a part of the software application itself, it can be used not only by the programmers during development, or by the QA engineers during testing, it can also be used to quickly identify errors that occur for end users of the software, in the exact way the application is used in their real-world environments.

So this new automatic testcase creation functionality, which represents an objectively assessable software development criteria that's built directly into the application, happens to also facilitate previously unheard of levels of feedback throughout the development process by virtue of its "Always On" functionality. And this kind of feedback is, as Wesselius and Ververs point out, critical to assuaging risks associated with the subjectively-assessable and non-assessable components of software development. In this way, Solid Step's new SQA approach, which has an instantly recognizable impact on the first component of software quality, also greatly helps with the other two components as well.

VII. Conclusion

Solid Step Software's new SQA product, Solid Step Test, can be thought of as "The Functionality of Software Quality" because it is a specific functionality, automatic testcase creation, that is included in the software application to make software quality assurance a radically easier task. For this reason it stands apart from all other SQA approaches because few if any others represent a functionality that is included in the software application itself, and certainly no others can enable completely automatic testcase creation for essentially any and all runs of the software.

One of the main kinds of applications Solid Step Test was originally developed for is software with an embedded GUI. har*GIS Field Information Systems' "Truckmap" application has a very large embedded GUI and therefore has provided a great opportunity to explore how this new SQA approach works in a real-world application. As this experience shows, Solid Step Test has proven to work extremely well for a large GUI application that uses a leading programming platform such as Microsoft's .NET platform.

But because Solid Step's patented reproducibility engine works for the general case of software applications, even those with asynchronous process interaction, this new SQA approach has the potential to become the standard for enabling the highest levels of software quality in many other types of software as well. Indeed it should be very interesting to see how well Solid Step Test works in practice in more applications, from GUI software to embedded applications, web-based software to command-line programs, and essentially every other kind of software application, especially now that completely automatic testcase creation is known to be an extremely reliable functionality in practice for at least one common type of software: applications with a complex embedded GUI.

REFERENCES

[1] J. Huth, *Apparatus for and Method of Action Synchronization for Software Reproducibility*, US Patent 6,845,471, 2005.

[2] S. Lohr, *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists and Iconoclasts – The Programmers Who Created the Software Revolution*, New York, NY, USA: Basic Books, 2001.

[3] J. Whittaker, "What Is Software Testing? And Why Is It So Hard?", *IEEE Software*, vol. 17, no. 1, pp. 333-337, Jan. 2000.

[4] J. Wesselius and F. Ververs, "Some Elementary Questions on Software Quality Control," *Software Engineering Journal*, pp 319-330, Nov. 1990.

Visit www.solidstep.com to download a demo of a Solid Step Test enabled application, learn more about Solid Step Software's new SQA solution, or to contact the author.