MS Windows Phone applications user navigation graph and input-feedback analysis

Quick guide

Navigation graph analysis and generation

The following instructions comprise the whole workflow, starting from a MS Windows Phone application packed as a XAP file, to obtaining two useful pieces of information. The first one is the inferred user navigation graph, while the second one is a series of navigation warnings. These warnings provide information about potential navigations that may be regarded as illegal with respect to the [Application Certification Requirements](http://msdn.microsoft.com/en-us/library/hh184843(v=VS.92).aspx). Specifically, these warnings deal with section 5.2.4 of the Technical Certification Requirements, which details the expected behavior of the physical back navigation button within phone applications. Roughly, these specifications stipulate that the behavior of this button should be that of navigating to a page previously experienced by the user during the ongoing session (these pages comprising what will be referred as the *navigation backstack*) and, additionally, that pressing the back button should result in the application exiting if (and only if) the current page is the initial page of the backstack.

The following instructions require only two steps that have to be carried out manually, the rest of the process is automatized. You’ll need the following tools to be able to carry out the complete analysis:

* Python interpreter (v2.6 recommended). Be sure to be able to execute .py files from the command line.
* A port of the line editing tool sed. Make sure you have the sed executable in your path.
* BCT, the Bytecode translator based on CCI, with the MS Windows Phone application instrumentation capabilities built in. Be sure to get the BCT\PhoneControlsExtractor directory as well; you may get all of this from the CodePlex site.
* The Boogie verifier tool, and an accompanying SMT solver to act as backend. This might require some configuration of its own, so be sure to check its documentation.
* GraphViz, or a similar tool able to handle graphs in DOT format.
* MS Windows Phone SDK libraries.
* You will need a few environment variables set up as well, namely:
  + BOOGIE\_PATH, which should point to your boogie executable.
  + BCT\_PATH, pointing to your Bytecode Translator executable.
  + CONTROL\_EXTRACTOR\_PATH should point to the control extractor script; typically residing in BCT\PhoneControlsExtractor\PhoneControlsExtractor.py
  + WPLIB\_PATH. This should point to wherever you have installed the MS Windows Phone SDK libraries. It is typically found in C:\Program Files (x86)\Reference Assemblies\Microsoft\Framework\Silverlight\v4.0\Profile\WindowsPhone.
  + DOT\_PATH should point to your DOT graph handler executable (dot.exe or similar).

Carry out the following steps to perform the analysis:

1. Extract the raw application files from the XAP package. The XAP package is essentially a renamed ZIP file, so it is only necessary to change its extension to ZIP and extract its contents.
2. The analysis requires understanding the application’s page and control structure. This structure can be posed by the developer in the form of XAML files. XAML is an XML-based language that, among other things, can describe the user interface of the application. These XAML files are not directly accessible, so they need to be extracted away.
   1. First, you need to locate the main application binary file. This will be a DLL file. If the package only has one DLL file, then that is the main application file. Otherwise, examine the AppManifest.xaml file in the package. The top-level element in this file is a <Deployment> element, which contains an EntryPointAssembly attribute that describes the name of the main application binary.
   2. Open the main application binary with Reflector, and locate the binary’s resource files. The necessary XAML files will be listed in here and can be copied away. Be sure to copy these in the same directory as the main application file.
3. The rest of the process is automatic. Run the navGraphBuilder.py script found on the BCT\PhoneControlsExtractor directory and follow its instructions. Running

navGraphBuilder.py --app <main application binary>

should suffice if you don’t need further configuration. This will result in many files being created, although the interesting ones will be NavigationReport and (assuming it was run with default options as in the command line shown) <main application filename>.pdf. The report file contains warnings related to any potentially illegal navigation the analysis may have found. This report also contains some (somewhat useful) pointers as to where the problem may lie. The PDF file is the actual inferred navigation graph, along with a few run statistics. Each node in this graph corresponds to a page created by the application, and edges between pages correspond to possible navigations between them (excluding navigations performed by the physical back button). The application’s entry node is identified by being pointed to by an edge with no originating page node.

More involved guide and caveats

There are a few things that may either go wrong during the process, or that are known to be incomplete for the time being.

* The tool relies on a flat structure of XAML files, that is, all XAML files should be extracted from the DLL to the same directory as the DLL. This may result in conflicts if the binary has a resource structure where different XAML files reside within different subdirectories (although I don’t know if this is possible and/or legal). The tool still has no support for this layout.
* If the XAML files have been compiled into .baml files (binary XAML), their extraction might not be feasible at all.
* The bytecode translation is the most critical point of the analysis and, presently, the translation is incomplete and may either fail or produce an illegal Boogie file. If that is the case, it may still be possible to carry on by manually correcting the Boogie specification and letting the tool run the remaining task. To this end, the navGraphBuilder script has a hidden argument that lets the user discriminate between the different tasks to perform. This functionality is activated by the –build <string>/-b <string> option. The string to be passed details the tasks to be performed, comprising a subset of the following characters (in order of execution, the order within the string passed is inconsequential):
  + - c: Analyze XAML files to extract layout information.
    - i: Instrumentate and translate the application file.
    - t: Test the Boogie file resulting from the translation. This step will fail if the Boogie file is invalid and requires modification.
    - b: create the Boogie queries required to build the graph.
    - q: run the Boogie queries.
    - g: build and output the graph.

If no –build <string>/-b <string> option is passed, or the string is empty, the default is to carry out all the steps. A typical scenario in the case of a faulty, but manually correctible, translation is to run the script a second time but with only the b, q and g flags set after manual correction (and possibly t, just in case).

* The resulting graph is potentially both *unsound* and *incomplete*, although it is possibly true that the degree of incompleteness is much less than that of unsoundness. More detail on these issues will be provided further ahead in this document.

<TODO> feedback analysis how-to

Implementation details

This section is intended to provide both a background on MS Windows Phone applications (their structure, design, navigation and layout framework), and also details on how we achieve the previous results and aim to achieve more precise results. The reader is not expected to know many details of MS Windows Phone application implementation, but is expected to be somewhat familiar with the .NET framework, event-based application development and the .NET runtime.

MS Windows Phone applications, being heavily GUI-based, are therefore implemented in an event-based fashion. Under this design, user interface controls such as buttons, boxes, and others are coupled with event-handlers, which are called by the phone framework whenever a certain event (such as clicked, dragged, activated, etc.) is performed over the control. These controls can be grouped along others in what are called *Containers*. Arguably, the most important of these containers, from the application’s user point of view, is the *Page*. A Page establishes a sense of context within the application, as developers usually use this distinction to present different functionalities. Furthermore, the MS Windows Phone framework itself enhances this distinction by performing easily recognizable animations when traversing (navigating) from page to page.

The current framework for developing MS Windows Phone applications allows for various ways of both creating this UI controls and for establishing their coupling with their event handlers:

* Both the controls and their coupling can be determined statically by the developer, writing the necessary code for them in the usual fashion.
* The developer may otherwise choose to create these controls dynamically (during runtime) and/or couple their behavior handlers also during runtime.
* Lately, capabilities have been introduced that allow for progressively decoupling the UI design from the application behavior itself. This is carried out through the use of separate code detailing the UI and their reactions to performed events.

The latter is achieved through the use of XAML (eXtensible Application Markup Language) code, given that MS Windows Phone applications are implemented under the same framework as Silverlight and WPF applications are. Everything that can be written in XAML can otherwise be expressed by a .NET programming language, as there is a direct mapping between XAML elements and CLR objects. However, XAML is easier to manipulate than code, especially for non-programming designers, which may possibly be involved in GUI application design.

Use of XAML for specification of user UI may range from simply declaring the existence of a certain control in the user interface, to making use of traditional source code completely unnecessary. For example, the following XAML code instantiates a **Button** control named **button1**, and this control may be referenced freely in other .NET programming language’s code:

<Button x:Name=”button1”/>

However, this declaration can be enriched so as to change certain static aspects of the control, for example, its displayed text:

<Button x:Name=”button1” Content=”Click me!”/>

Going further, XAML code may associate some of this control’s events to handlers that, presumably, will be written later in some .NET language. For example, the following code binds the **Click** event to a procedure that may respond to that event:

<Button x:Name = ”button1” Content = ”Click me!” Click = ”button1\_on Clicked”/>

The code behind button1\_onClicked might, for example, change the text to “You clicked me!”, like so:

public override void

button1\_onClicked(object sender, RoutedEventArgs a) {

button1.Content= “You clicked me!”;

}

Also, the handler code itself could be embedded into the XAML specification. The following XAML is equivalent to the previous combination of XAML and C# code:

<Button x:Name = ”button1” Content =”Click me!” Click = ”button1\_ onClicked”/>

<x:Code><![CDATA[

void button1\_onClicked(object sender, RoutedEventArgs a) {

button1.Content= “You clicked me!”;

}

]]></x:Code>

Furthermore, XAML provides more functionality to even make such code embedding unnecessary, through the use of Storyboards and Animations.

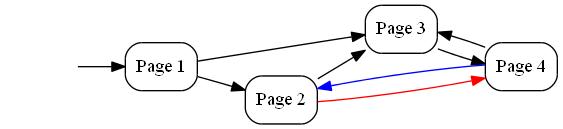
In light of this, any analysis that aims at being able to understand an MS Windows Phone application’s behavior, necessarily has to be able to understand behavior specified through use of XAML code. The first step carried out by our tool aims at this objective. The script PhoneControlsExtractor.py is essentially a XAML parser that extracts control information that will be used throughout the remaining parts of the analysis. As we’ve seen before, this extracted information will complement information that we will find by analyzing the code. In some cases, this information cannot be found anywhere else than in the XAML files.

The XAML analysis is far from complete, and only scratches at the surface of all the possible behavior that a developer may choose to encode in XAML. Complete XAML compliance in the parsing of this information is an important part that deserves much future work. Currently, the script provides extraction for a subset of possible controls, and for that subset, only a subset of interesting properties and handlers. Features such as embedded code, for example, are currently not being extracted away by the tool, among many others.

The second step of the analysis involves understanding of the application code itself. Because our initial scenario was based on analysis of compiled third-party applications, this section of the analysis requires handling IL code directly, rather than a definite .NET programming language. Therefore, the work on this section is based on the CCI framework, which essentially allows clients to decompile IL code, analyze it, and even change it on the fly (by adding, replacing, deleting code, etc.).

The first analysis we tackled was aimed at understanding page navigation behavior behind a given MS Windows Phone application. Specifically, the research question was to statically determine whether an application could be used in such a way that it triggers a navigation request on the press of the physical phone’s back key that is in direct violation of the Technical Certification Requirements (TCRs). The TCRs preclude the application from performing arbitrary navigations on back key presses. Specifically, it limits those navigations to pages that are presently on the navigation backstack, that is, to those pages that have been previously traversed by the user during the current session.

Unfortunately, for the general case this problem is undecidable, so we propose an approximate solution to the problem, based on an abstraction of the navigation structure of the application. The idea revolves around the fact that **if** we were provided by an oracle with the whole set of possible page-to-page dynamic navigations, excluding those triggered by the physical back button; and on the other hand, we were provided with all possible back key navigations, then an incomplete (but sound) check for invalid navigations is feasible.



For example, consider the hypothetical navigation structure proposed in the figure, where Page 1 is the initial page. Black colored arrows denote the navigation structure, while the blue and red arrows denote navigations that may be carried out as a response to a press of the physical back key. Given all this information it is clear that, with respect to the TCRs, the red arrow navigation is *illegal*. There is no sequence of pages induced by the navigation graph in which Page 4 precedes (directly or indirectly) Page 2. Therefore, the red navigation couldn’t possibly be part of Page 2’s backstack. On the other hand, the blue arrow navigation is possibly legal, as navigation sequences that get to Page 4 through Page 2 would make the back navigation legal. However, it cannot be asserted with complete certainty that *all* blue navigations are legal, as there exist (potentially) navigation traces to Page 4 that do not include Page 2 in the backstack (namely Page 1 followed by Page 3 and finally Page 4). Although incomplete, the knowledge gained is still useful in the process of automatically detecting TCR violations.

With the aim of implementing this approach, an analysis is needed that can yield two important pieces of information:

1. The navigation abstraction in form of the navigation graph (or an approximation to it), and
2. Information describing possible navigations performed by physical back key button presses.

Such an analysis needs to be aware of all possible ways in which an application might navigate from a page to another. Fortunately, the MS Windows Phone application SDK provides a fairly tight interface for accessing navigation facilities, namely through the System.Windows.Navigation. NavigationService class (which, for MS Windows Phone, differs substantially to the interface exposed for general .NET applications). This class exposes methods that allow clients to:

* Navigate to an arbitrary application page, via the Navigate(Uri) method,
* Back-navigate to the latest page on the state-based back stack via the GoBack() method, and
* Remove entries from the state-based back stack via the RemoveBackEntry() method.

For the first objective we are only focused on obtaining the forward navigation graph, so the analysis needs only be aware of Navigate() calls.

In order to build this navigation graph, our approach involves two phases. During the first phase, we traverse the code information built by CCI, while the second involves carrying out a static analysis of that very same code, using the Boogie verification tool. In the following, we describe both phases with some degree of detail.

During the first phase, we walk down the application’s metadata and code AST obtained from CCI. In the context of this phase, it becomes necessary to plug in missing information that was previously obtained by analysis of the XAML files.

<TODO> describe exactly what, where and how XAML information is being plugged in.

Along completing this missing information, we also monitor the application’s interaction with the NavigationService interface, in order to be able to track the current page being navigated at any point in the execution of the application, as well as the intended target page of the NavigationService call.

<TODO> describe the problems with Uri being based on String

With the navigation information having been tracked, the task of figuring out which page can navigate to which other page is forwarded to a static analyzer. We tried several approaches to this task, with varying degrees of success.

<TODO> describe main inference algorithm

However, although the NavigationService is tight, there are little restrictions on from where or when this interface may be accessed. The only restriction imposed (and enforced) is that only the GUI thread may be able to access it, but this does not remove much of the uncertainty regarding possible navigation origins and targets. Therefore, in order to obtain greater precision, we need a way to slice away some navigations between pages that can be deduced to be infeasible. We describe some ideas on this process and our implementation in the next section.

<TODO> describe current page slicing

<TODO> describe feedback analysis