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.

DISCLAIMER: the following does not work at all with applications built around the XNA Framework, which we haven’t researched at all yet.

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.

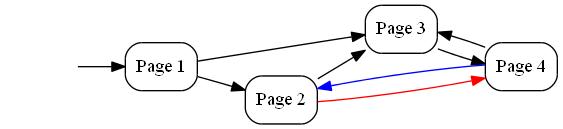
In light of this, any analysis that aims at being able to understand an MS Windows Phone application’s behavior, necessarily has to be able to understand behavior specified through use of XAML code. The first step carried out by our tool aims at this objective. The script PhoneControlsExtractor.py is 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.

Fortunately, the framework and API for handling these physical back key presses is very forgiving, in the sense that if a developer is oblivious to this issue, it is difficult for her to actually act against the TCRs. If left unmodified, the physical back key default functionality is to act precisely as described in the TCRs. The problem is that this default may indeed be overridden with arbitrary code and behavior, such as precluding the back key from navigating back to a page, or handling it in such a way that in navigates to an arbitrary page. The programmatic way to outweigh this default behavior is to override a certain procedure. A few scenarios can then take place:

1. The developer chooses not to override the default behavior.
2. The developer does override the default behavior, but does not perform any action that may influence the default navigation behavior, which in turn is carried out.
3. The override behavior is overridden by the developer, and although no navigation is performed, the default behavior is precluded from taking place. This may be achieved by severing the call chain explicitly, or implicitly, by throwing an exception.
4. The override behavior is overridden in such a way that it navigates to an arbitrary page.

The first two cases fall directly on the set of TCR sanctioned behavior, while the other two do not. The first objective is then to detect these two situations and, when possible, determine if the intended navigation is possibly legal, that is, if the navigation graph contains a path such that, when traversed backwards, can link the current page to the intended page for the back button navigation.

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

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

Such an analysis needs to be aware not only of the potential overrides to back key presses behavior, but also of all the 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. In particular, we need to keep track of all user interface controls that have been defined by XAML code, especially anonymous, as they have no direct identifiable correlation in the IL code, and its backing CLR object needs to be identified somehow. Since XAML-declared controls are defined within the context of a container, when we eventually traverse this container (its CCI TypeDefinition), we take the opportunity to look for the required backing fields and/or add them to the TypeDefinition if necessary[[1]](#footnote-1). This sets the ground to allow us to identify the control’s potential event handlers[[2]](#footnote-2). During this container’s type definition traversal, the container’s constructor is also injected with code that essentially performs the same tasks that the XAML loading framework would carry out. Namely, as the control may have been set initial values to its properties and event handlers via XAML code, we replicate this initialization within the control’s container’s constructor to reflect the developer’s intention as closely as possible[[3]](#footnote-3). However, XAML-declared objects and code and this dynamic injection of control information and behavior remain as a source of incompleteness in the analysis: theoretically, any control, properties and behavior handling declarations and modifications may be carried out through XAML code. However, in order to be able to track every possible declaration, we would need to be 100% XAML-compliant, a non-trivial task. In our work so far, our handling of XAML behavior has been discovery-based, and we have built on the approach as more features surfaced on XAML based applications, therefore missing not-yet-tracked information. Ongoing work is being carried out in order to attain full XAML compliance and minimize this incompleteness[[4]](#footnote-4). Additionally, for each control and property that is declared within XAML, the control’s interface for manipulating this property needs to be known; it is no use to just carry out the initialization but then fail to recognize when these properties get modified and continue assuming their initially set values. This information needs to be known at static analysis time, and although for standard controls it may be easy to collect, it wouldn’t be for user-created controls or those in external libraries. Deriving this information via a reflection approach can only help so much. This information might need to be provided by the developer himself in some form to be analyzed.

The second part of the IL code traversal phase deals with the understanding and tracking of navigational behavior. This task consists of two parts: first, determining whether the default back key press is being overridden; and second, tracking the possible pages that may be active (showing to the user) at the moment of a navigation request, and tracking the possible results of said navigation request.

Back key press overrides, as was described before, are fairly simple to identify. However, since they are handled at a page level, this overriding has to be checked for every possible page in the application. Possible pages are described in the same manner as controls, both through XAML and developer code, so identifying pages is achieved in a similar way as identifying controls. Given a page, identifying whether a back key override is present or not is easy, as an override must take one of the following choices: either override the interface method OnBackKeyPress, or else register a callback to the BackKeyPress event. Whichever the method chosen by the developer, we must then traverse the method (the override or the event handler) looking for arbitrary navigations (calls to the NavigationService interface) or call-chain severing (event handler cancelling or exception throwing). This method traversal must also take into account that the method may, in turn, call other several methods that may themselves carry out the undesired action.

This process is somewhat more involved in the case of the Page overriding the behavior via callback registration, as the callback is possibly an (anonymous) delegate. Since the current static analysis does not perform data-flow analysis, it might be impossible to know exactly which method is being registered as a callback, and therefore its code cannot be traversed.

Whether it is the case that the callback method can be traversed and is found to potentially violate specifications; or that the callback cannot be navigated, warnings are generated to make the user aware of this fact. In the first case these warnings are more helpful as they point out the potentially offending code; in the second case we can only hint at the possibility of code performing illegal behavior.

<TODO> describe source files where code that does each thing is found

The navigation graph building requires tracking the pre- and post- values of the current (from the point of view of the user) navigation page at each point where it might change. Furthermore, since we are interested in detailing the navigation structure by assuming at first that back key navigations are legal, we need only focus on forward navigations. These navigations are carried out by calls to the NavigationService interface Navigate method, which in turn takes an URI as argument. URIs are used extensively as abstractions of page locations and map directly to the XAML files containing the declarations for each page. URIs, in turn, are created from their string representation, and may be composed of anything legal referring to the URI standard[[5]](#footnote-5). So, at a URI creation point, the string that will be associated to this URI needs to be tracked in order to eventually be able to deduce the possible targets of a navigation call. In fact, for the objective of deriving the navigation graph, it suffices to track only the naming part of the URI that will reference to the XAML file for the Page.

However, strings are difficult to keep track statically. As we stated before, we are not performing any data flow analyses on the application yet, so in principle it cannot be told whether a string created in some part of the program is eventually used to create a given URI that, in turn, is used to perform a navigation call. However, if the actual *content* of the string can be extracted statically, then the scenario is different. In this case, we can track a constant, unique string variable for each static string potentially used in a URI, along with the static portion of the string being used to create said URI. If this information can be tracked, then it can safely be inferred that, when a navigation call occurs with a URI being passed such that it was created with a string whose static portion maps to “thispage.xaml”[[6]](#footnote-6),that then the application will effectively navigate to the page mapped by the thispage.xaml file[[7]](#footnote-7), and we can effectively track the (new) current page being visited by the user.

However, it isn’t always the case that the string can statically be derived. For example, it could be the case that, depending on program state, a certain string gets created in place of another. Also, the string to build the navigation URI might be recovered via reflection from the application’s resources. In these cases we are faced with complete uncertainty regarding the possible target of the navigation. We can only safely assume that the navigation may potentially reach *every single other page* in the application, as we cannot discriminate between them all. This is the major source of unsoundness in our approach, and one that we plan on mitigating by performing more data-flow analysis over the variables involved. However, these variables being of the string type, they could prove hard for a data-flow analyzer to track, as it will possibly be overwhelmed under the possibility that they are often difficult to abstract away.

With the navigation information having been tracked this way, 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.

All approaches are based on the same basic idea, that of pruning possible pairs of pre and post value of the current navigation page for each navigation request. In order to do this, we rely on the Boogie verification tool. Along the steps that were discussed previously, we obtain a translation from the application’s IL code to a representation in Boogie code, amenable to verification. We couple this Boogie code representation of the application with assumptions and assertions that follow the pattern:

assume $currentPage == <pre\_navigation\_current\_page>;

<navigation call>;

assert $currentPage != <post\_navigation\_current\_page>;

The assume statement has the effect, during verification, of sieving program traces that do not verify the assumption condition. In the example provided above, the assume statement effectively filters out all program traces where the current page previous to the navigation is *not* the value given by the pre\_navigation\_current\_page expression.

Conversely, the assert statement succeeds when the condition expression evaluates to true, and fails otherwise, in which case terminates the evaluation of the failing trace. In the example, the assertion succeeds only when the navigation has resulted in the current page being updated to page *distinct* to that specified by the post\_navigation\_current\_page expression.

It is necessary to remark at this point that both assumptions and assertions evaluate their arguments with respect to abstract values. That is to say, assertions and assumptions will be satisfied if every possible concrete value mapping to the abstract value effectively satisfy the condition, and fail otherwise. Under the light of this statement, we can understand the effect of the previous pattern. If the assertion were to *fail*, then we can certainly say that, since the assertion was reached, the assumption was satisfied. Therefore it is safe to assume that *at least one* of the concrete values of the pre navigation abstract values of the current evaluates to pre\_navigation\_current\_page. On the other hand, since the assertion failed, it must be the case that not all possible concrete values satisfy the condition, that is, the concrete value *may* have been that of post\_navigation\_current\_page. The complete perspective tells us that if the assertion fails, then there exists (at least) one *abstract* program execution trace in which, from a current page given by pre\_navigation\_current\_page it is possible to perform a navigation action that takes the user to the page given by post\_navigation\_current\_page. It remains to be seen if that abstract trace can effectively be concretized into an *actual* trace of the program. We can then (in an unsound way) assume that a navigation between those two pages is feasible. However, if the assertion does not fail for the complete program, we can safely assume that there is no such navigation possible in the whole program[[8]](#footnote-8).

If we were to evaluate these assertions for every possible instantiation of pre and post navigation pages (that is, for every possible pair of pages), then we can provide the navigation graph. Since we can know which all possible pages are, and the Boogie verification procedure is effective, this approach is feasible.

<TODO> talk about corral experience

Having found that whole program analysis was infeasible for reasonable MS Windows Phone applications, we veered towards more modular approaches. The most modular approach consists of verifying each method that can eventually perform navigation completely decoupled from the rest. Besides, all methods that are eventually reachable by those under analysis need to be either provided with contracts or else inlined, so that any influence they may have on the final result can be effected during the analysis. Since the original aim was to provide navigational information without any user (nor tester) interaction, we opted for the inlining approach.

Unfortunately, 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. As a result, the navigation graph obtained for the test applications by implementing this approach tends to be the completely connected graph, which carries no new information at all. In order to obtain greater precision, we need a way to slice away those navigations between pages that can be deduced to be infeasible.

In order to achieve this goal we leverage on both the known fact that navigation performing functions can only be called from the GUI thread, and on other assumptions regarding common development practice. These second type of assumption is unsafe, since in theory developers could write their applications in a non-standard way, but during the analysis on realistic apps, the assumption has proved to be in line with real world development approaches.

The approach relies on the fact that event handlers are rarely (if ever) directly called by the developers’ code, and rather left at its own design to be called back by the framework. In the case of GUI control handlers, then, it is guaranteed that these handlers will only be called as the result of GUI events being raised. In this case, there is no other possibility that they be called from the GUI thread, and the events can only be raised by the user actively interacting with the application. This results in that, for example, if Button A is contained within Page 1, we can safely assume in most cases that the event handlers for Button A will only be called in a context where Page 1 is actually being shown to the user as the current page. This assumption, in the context of the previously described approach, allows us to be more precise in discovering the pre-navigation current page, as we can pinpoint its value to a single page.

We applied this approach to all control GUI event handlers, including those implemented by (anonymous) delegates. This greatly aided in improving the precision of the navigation graph obtained by reducing its unsoundness.

<TODO> results

<TODO> unsoundness and incompleteness discussion, future work

<TODO> describe feedback analysis

1. Caveat: we are not handling yet the case in which an anonymous control is contained in turn by an *anonymous* container. In this case, the control is taken as a child of the first named container found in a traversal towards the outer containers. The anonymous container itself though, and its potential related behavior, may be lost in translation. [↑](#footnote-ref-1)
2. This explicit tracking of controls and handlers is initially unnecessary for the navigation graph derivation alone and as we are presently carrying out, but it turns useful in slicing away some behavior, as will be described later. It is also necessary for feedback analysis; and it will also be necessary when whole-program analysis is evaluated as an alternative to modular analysis. [↑](#footnote-ref-2)
3. It is necessary to note that, at runtime, these actions would be carried out by the MS Windows Phone framework. However, during IL translation, in order to minimize the amount of code to translate and therefore the load on the Boogie verifier later on, framework calls will be translated as stubs. This is acceptable in the general case, but we need to be more specific in situations such as initialization and others that will be described later. [↑](#footnote-ref-3)
4. Trying to handle more declarations via reflection is a possible way to augment XAML compliance without actually understanding the declarations, but at a risk of translating too much of the MS Windows Phone framework into the application itself (which impacts on the load given to the Boogie analysis). [↑](#footnote-ref-4)
5. See RFC 3986. [↑](#footnote-ref-5)
6. Details such as string normalization, URI deconstruction and string operation resolving are skimmed. [↑](#footnote-ref-6)
7. Actually, not quite so. Navigation events are completely asynchronous: they can be seen as being pushed into a queue of navigation actions. This queue may be manipulated at any time before the request is actually pulled from the queue by the MS Windows Phone framework and navigation is actually realized. Actions that could happen include pushing further navigations requests down the queue, or removing navigation requests altogether. However, for the sake of the analysis, we assume that a navigation that is requested is effectively serviced. In theory, this introduces a certain degree of unsoundness, but in practice it seems to correlate with reality. A comprehensive study with a large corpus of real-world phone applications is needed to support this assumption. [↑](#footnote-ref-7)
8. Modulo completeness of the approach (which, as we discussed, is not complete). [↑](#footnote-ref-8)