Automated Device Testing for Windows Phone 7

Initial System Design

Axelerate Solutions, inc

Abstract

The purpose of this document is to walk through the real-world proof-of-concept application of automated testing for Windows Phone 7 devices. This discusses the single and two-device testing system, as well as the real-world implementation details.

Introduction

After initial research, it was found to be reasonably possible to test Windows Phone 7 devices through an automated robotic system. The initial research focused on one of the largest problems: graphical user interface detection. This document focuses on the real-world details of implementation, including a testing procedure language, robot motion, and GUI verification.

It was found that a given testing procedure could be broken down into smaller steps. Each of these smaller steps is executed using “motion”, “extraction”, and “verification”. For each of these smaller steps, we attempt to execute it by first using a robot-**motion** that mimics human input. After this motion, we attempt to **extract** the resulting screen components of the real-world device, using the GUI extraction method defined and detailed in the initial research document. After extraction, we must **validate** that the real-world screen matches the expected screen.

For example, if we are attempting to call a specific phone number, it consists of going to the home screen, pressing contacts, then pressing the dial pad, entering the ten-digit phone number, and finally pressing the “call” button. This entire testing procedure can be split into smaller steps, such as the final step of pressing the “call” button. When we do this, we first tell the robot to press an on-screen location (or apply some other motion), then take a picture of the resulting screen to extract the GUI information, and finally validate making sure that this screen matches the resulting screen we expected.

The real-world implementation is written in C# and compiled with Visual Studio 2010. The only dependencies are the open source computer-visions library (AForge.net) and the serial-string communication protocol based on Adept’s proprietary controller language (V+). The later, V+, is simply a robot instruction language, and not a formal library. The application has a GUI front-end that loads testing scripts and configuration files, as explained in the Demonstration Manual document.

There are currently three major code bases that can be used for further development and demonstrations. These are “AutoTest_Demo”, “AutoTest_Backup”, and “AutoTest_MultiDevice”. The first is the single-phone demo solution, the second is a simplified “backup” demonstration application, and the third is a two-phone demo solution. Each project is roughly 2.5 thousand lines of code, and is outlined in the rest of this document.

Motion Control

Movement

The robot, an Adept Cobra S350 Robotic Arm, is controlled by a serial-string interface. Movement instructions are written in plain-English ASCII strings sent over a serial connection, giving commands to move the robot to specific locations, set speeds, and other options. These instructions are actually V+ instructions sent over a serial connection created by this demo application. This communication system is simply mimicking a user opening a terminal connected with the robot.

Though the robotic system can retain its own V+ functions or script, it is better to centralize all code into one source solution, and thus we only send movement instructions, not function calls, from the C# application to the robot. Examples of these instruction strings can be found within the V+ programming language documentation. Note that when sending instructions through the terminal or serial-string interface, all commands must start with the "do" keyword, which states that the rest of the line will be an instruction to execute. Also note that the V+ language is not case sensitive.

Example instructions, commonly used in the demonstration application, include:

- "do move world_origin:trans(10, 25, -100, 0, 0, 0)" – This moves the robot to the relative position (10, 25, -100) compared to the "world_origin" position. Note that the last three arguments are rotations along the x, y, and z axis which are left to default 0.
- "do where" – returns a specially formatted position of the world based on x, y, z, rx, ry, rz, with the later three elements being rotations along the x, y, and z axis.
- "do speed 100MMPS always" – Sets the target speed of all future movements to 100 millimeters per second.

A better approach would be to use the official Adept TCP/IP communication library for communication, which allows end-developers to call movement functions rather than form their own instruction strings.

There are only three motions currently implemented with the Adept robotic arm that can correctly execute onto any given device. The first, a simple "push", mimics a human finger-press at a given location. The second, a "drag" event, is a point-to-point drag event, such as a finger drag to scroll down, up, or side-to-side through menus for a given speed. The final movement type, a "flick", mimics closely a human flick event my pressing at a given location, and quickly moving towards another location while lifting up the probe.

All three of these movements are executed by sending move-to instructions with the demonstration application to the robot controller. None of these movements are pre-programmed into the robot or robot controller. Note that by pre-programming (sending the robot a generic function for "push", "drag", "flick", etc. into the robot controller) may increase run-time speeds, but it is not critically important as current communication speeds are sufficient. Also note that the flick motion is not consistent as it heavily depends on correctly configured probe depth.

Through experimentation, it was found that the coordinate transformations are not intuitive. When giving a position for the robot to go to, you must always define a new position relative to a pre-defined position. This position may simply be defined as (0, 0, 0), the world origin, but must at least be defined. This is why current movement instructions are always combined with the global variable "world_origin", which is the world origin saved as a variable. This is set at the beginning of each communication initialization.

There is another issue related to coordinate transformations, which are the inconsistencies of axis directions. This is unclear at the moment, but to correctly move the robot, one must invert the signs of the x and z coordinates before sending the go to instruction. For example, when sending the coordinate A, whose members include an x, y, and z components, you must actually send a new coordinate B, which is defined as B.x = -A.x, B.y = +A.y, B.z = -A.z. It is unknown why this is needed, but will correctly place the probe at the given locations. These can be confirmed by looking at the LCD screen of remote-controller device.

Screen-to-World Coordinate Conversion

The coordinate system that the robot operates in may not be correctly aligned with that of the target device's screen. This means that the robotic arm may have to translate across two coordinates, rather than one, when moving from the top-left corner to the top-right corner of the device. To fix this, the application needs to know all four corners of the device in term of probe coordinates, but also in terms of screen-coordinates (as defined by the user during configuration).

Once all four corners are known, a simple translation, rotation, and scale can take place to correctly convert screen coordinates to robot coordinates. There are several ways to do this, but the current implementation simplifies this by only dealing with the screen scale differences and lightly resolves the possible screen rotation.

The current approach is done by finding the width and height ratio between the real-world screen (measured in millimeters) and the expected screen (measured in pixels). We then multiply the target pixel coordinates, such as a button location, by this ratio and add the top-left origin of the device. This generates a real-world coordinate, which we can send to the robot. Note that this approach does not deal with any rotational differences.

The correct method would be to do a plane-to-plane mapping, which can be done through by specifically creating a transformation matrix that can map screen-coordinates to real-world coordinates. The projection-transformation, that resolves camera distortions, is critical as it help with correction position differences (i.e. the center of the screen might look larger than the sides). If needed, you could create a conformal map to correctly convert screen-coordinates to real-world coordinates without the need of applying a projection transformation.

Screen Extraction

GUI Extraction is heavily discussed in the Initial Report document. Using the AForge.net library, a GUI extraction algorithm was developed to specifically find all rectangle-like components on-screen.

Once the source screen is found by the user's configuration, a custom GUI extraction algorithm must be applied to extract all possible GUI buttons. We first take the screen image, convert it to grayscale to simplify image data, blur it to ignore smaller edges, dilate non-black components for better edge detection, apply edge detection, and finally apply blob-finding.

The output of the GUI extraction algorithm will place several dozen yellow or green boxes on top of the original image. The yellow boxes represent unlikely, but possible, UI components. Green boxes, with labels, represent very likely, but not guaranteed, UI components. Based on internal configuration, there may or may not be many false-positives. Note that when doing multi-device demonstrations, the sensitive of the algorithm is much higher and is done so because the device's screens at the time of the development process, were giving high-levels of reflection.

The current solution does not to categorization (linking rectangles with correct titles) or Optical Character Recognition (parsing strings on-screen and saving them within GUI rectangles) simply because of the technical difficulty.

Verification

The current virtual UI verification algorithm takes the list of expected GUI buttons and compares it with the list of extract GUI buttons. For each possible pair, the averaged corner distance is calculated and compared with a static value. If the pairs are too far apart, then they are considered not matching. If an expected GUI button is never matched, the verification process fails.

A more valid approach would be to not only comparing locations, but to also compare icons, text, and color. This is not implemented in the current solution because of the technical difficulty. Another significant issue is that depending on the screen resolution and size, some buttons should match, but are larger because of the physical screen's size. There should be a scaling factor in the verification algorithm, but may introduce a significantly higher algorithmic complexity.

Test Procedure Scripting Language

A test description language is necessary for implementing this automated testing system. In this scripting language, all that is required are motion instruction we want to execute and the resulting screens we are expecting. The current scripting language is an element-per-line file format that expects certain elements to be defined at certain locations. The exact file format is defined in the source file "TestProcedure.cs" at the top of the text as a comment. A more correct scripting language format should be written in XML to support complex movement instruction, as well as more detailed resulting screens.

Configuration

Configuration is explained in detail in the Automated Testing Demo Manual document. It is important to note that no device-specific information is saved in the scripting language. All configuration is saved per-device, and can be retrieved or modified as needed.

Known Issues & Limitations

Phone Sleep Time

When setting up tests to complete, it is very possible that a phone might go into sleep-mode or go into a locked mode. The tester must change the phone settings so that it doesn't automatically fall sleep or accidentally block the testing user. This is explained in the Manual Testing document.

Scripting Language

The language of the script file should be derived from XML, because of its clarity and maintainability. The current format is a plain-text element-per-line file that can easily break if there are too many whitespaces or newlines. This "correct" file format must support multiple movement types, different types of devices (configuration should be saved in independent files), and varying screen sizes and ratios. The later aspect is very critical, as on-screen GUI components may be valid, but fail for validation as their sizes are different.

Script Maintainability

Doing this "motion", "extraction", and "verification" loop is effective at run-time and correctly simulating user input through an automated process, but is not easily maintainable. If the GUI were to change in any way, or to have a test script extended, it will take a large effort to rewrite the script. A script-developer or designer tool should be build so that a new testing procedure can be written by simply looking at current user behavior (via a video stream or pictures) and parsing that into a new script file.

Different Robot Support

It may be required to use different robots to execute certain tests. In these cases, nothing should be changed because the communication system (V+ through serial-string interfaces) is standardizes across all ADEPT robots.

Commercial Visions Library

AForge.net is an open source project, and thus cannot be use commercially because of the license restrictions. Redeveloping a visions library is not cost or time effective because of its difficulty and special need for complex processor-level optimization. Several replacement have been found:

Matrox: Machine vision and imaging software; Good with object detection, but might be too big and cumbersome. Includes standalone tools and OCR support.

Cognex: VisionPro; Fast, but not too well documented. Includes a pseudo scripting language. Includes OCR that detects character string locations but may not convert them.

Adept: AdeptSight; Fast, well documented. Includes many end-user tools and applications. OCR support unknown.

Automated Configuration

One of the goals of this project is to keep the system as automated as possible. Configuration, the process of defining certain aspects of a new device before running a testing procedure, is time consuming and tedious. It might be possible to automate certain aspects of configuration. Specifically corner detection, both for the physical device and screen, are possible. The endoscope built into the probe may be used to automatically detect physical corners of the device, while stickers or painted dots can be used to detect the screen corners at run-time. This is discussed in more detail in the Initial Research document, within the computer visions section.

Reflections and Variable Lighting

The problem of having reflections on the device screen (such as the probe or camera itself) is mostly resolved by diming the room the device is working in, but is still an issue. The most simple and accurate solution would be to cover the robotic chassis' working space with a non-translucent material such as black plastic.

Camera-Arm Coordinates

There are several ways to convert arm coordinates with camera coordinates. Each has its weaknesses and strengths, as well as may require reconfiguration after each adjustment of the physical platform. The current method requires extra time to measure the screen twice (once by the robe, once by the camera) but works well for this current situation.

Getting the Correct Probe Depth

Setting the probe depth is complex and tedious. There should be an automated method of finding the best depth using a range-finding sensor, or other device.

GUI Categorization

The real-world solution should have better tools to categorize and thus compare the extracted GUI screen. Once a possible GUI component is extracted, it is important to compare any internal strings or icons to better and more correctly verify screen comparisons.

Camera Auto-Focus

The camera currently waits a full five seconds before taking a picture because it needs the time to stabilize the camera focus. To save time, the camera should be placed in the best position to take a picture of the screen, have that focus saved, then keep the focus variable locked. This saves time and prevents possible issues where the camera is still focusing when it attempts to verify the screen UI.

Hidden GUI Components

A critical feature to this automated testing system would be the search of a button that isn't on-screen. If the testing procedure requires to press a button that is in a given menu, but must be strolled to, this software should be smart enough to scroll and actively seek the button. Similar cases are when valid buttons are on-screen, but not in the right order.

Aliasing Camera / Screen Distortion

Certain devices and screens generated an aliasing distortion that creates several dozen lines running over the screen of the device. This is hard to fix in software and is significant enough to return false-positive information since the colors match other buttons and the screen background.

OCR Libraries

Microsoft has supported OCR (Optical Character Recognition) in past products, but no longer supports it in Office 2010. There is also no major work within the Microsoft Research group or the internal open-source code base. The best solution at the moment is to look at computer vision libraries listed above in the section titled "Commercial Visions Library".

Results & Conclusion

Creating an automated testing system that can interact with one or two phones is very possible. A proof-of-concept demonstration was created, in which a fully scriptable environment was designed. Within this system, several example test procedures were created and could be executing using one phone, or between two phones. From this real-world implementation, several known issues and limitations were found. This is described in detail in the above document. Most notably is the time needed in configuration, the sensitivity of the visions algorithm, and the current lack of OCR (optical character recognition) support.