Cloud Twin: Native Execution of Android Applications on the Windows Phone

Ethan Holder, Eeshan Shah, Mohammed Davoodi, and Eli Tilevich
Dept. of Computer Science
Virginia Tech, Blacksburg, VA 24061, USA
{eholder0,eeshan9,mdavoodi,tilevich}@cs.vt.edu

***Abstract*—To successfully compete in the software marketplace, modern mobile applications must run on multiple competing platforms, such as Android, iOS, and Windows Phone. Companies producing mobile applications spend substantial amounts of time, effort, and money to port applications across platforms. Creating individual program versions for different platforms further exacerbates the maintenance burden. This paper presents *Cloud Twin*, a novel approach to natively executing the functionality of a mobile application written for another platform. The functionality is accessed by means of dynamic cross-platform replay, in which the source application's execution in the cloud is mimicked natively on the target platform. The reference implementation of Cloud Twin natively emulates the behavior of Android applications on a Windows Phone. Specifically, Cloud Twin transmits, via web sockets, the UI actions performed on the Windows Phone to the cloud server, which then mimics the received actions on the Android emulator. The UI updates on the emulator are efficiently captured by means of Aspect Oriented Programming and sent back to be replayed on the Windows Phone. Our case studies with third-party applications indicate that the Cloud Twin approach can become a viable solution to the heterogeneity of the mobile application market.**

I. Introduction

The current market for mobile applications is highly fragmented between the plurality of platforms, including Android, iOS, Blackberry, and Windows Phone. There is great economic benefit to supporting successful mobile applications on all major platforms, so as to maximize the potential customer base. Unfortunately, porting a mobile application across platforms incurs software development costs. Furthermore, supporting an application across multiple mobile platforms exacerbates the maintenance burden, as each bug fix and feature enhancement have to be applied to all the supported platforms.

Recognizing the need for heterogeneity, mobile application designers have created frameworks for cross-platform mobile development, such as PhoneGap [18]. These platforms typically leverage the mobile web browser that executes applications written in JavaScript and CSS. Despite the widespread use of cross-platform mobile frameworks, developing native applications remains the preferred practice in the mobile software market. Native applications (i.e., written for a specific platform using the platform's API) have a unique look-and-feel expected by the customers; they also take advantage of platform-specific features such as the platform's native maps (Google Maps for Android [5], Apple Maps for iOS [6], and Bing Maps for Windows Phone [7]).

In this paper, we present a solution to the heterogeneity problem of the mobile application market that does not require manual porting of applications nor shifting the development into cross-platform frameworks. Our solution, called *Cloud Twin*, makes it possible to execute mobile applications written for one platform natively on another platform. The basic idea behind Cloud Twin is that a mobile application has two isomorphic versions: *the source*, executed on a cloud-based edge server, and *the target*, executed on a local mobile device.

A mobile application is preprocessed, and its source UI screen is automatically translated to the target platform. Then the UI actions performed on the target are captured and sent to the source, at which time they are replayed using an emulator. The resulting UI changes on the source are then detected, transfered, and applied to the target application. In addition to its basic services, Cloud Twin also specially handles sensor input as well as time and location services. In other words, it ensures the target's environment is used by both versions of the application. The reference implementation of Cloud Twin natively executes Android applications on the Windows Phone.

The most surprising insight we have derived from experimenting with our prototype implementation is that the mimicking functionality of Cloud Twin is quite efficient, with the resulting latencies not adversely affecting the user experience. With the edge server running within the same administrative domain and connected to by a Wi-Fi network, the latencies of executing common UI actions in a typical application never surpassed the one second threshold [19], thus making the Cloud Twin approach feasible and useful. In particular, we were able to natively execute several small but real Android applications natively on the Windows Phone, with the users not suspecting that they were natively interacting with applications written for a different platform. Although these initial experiences were only informal trials, they nonetheless indicate that Cloud Twin has the potential to become a practical solution to the problem of making a mobile application available on a variety of platforms.

The rest of this paper is structured as follows. Section II gives an overview of the main components of Cloud Twin. Section III provides the initial evaluation results. Section IV discusses the advantages and limitation of Cloud Twin. Section V compares Cloud Twin to the related state of the art, and Section VI outlines future work directions and presents concluding remarks.

```
if (view instanceof Button) {
    Element newElement = document
      .createElement("Button");
    Button button = (Button)view;
    addComponentProperties(view, newElement,
      "button");
    newElement.setAttribute("text", button
      .getText() + "");
    newElement.setAttribute("textSize",
      button.getTextSize() + "");
    newElement.setAttribute("textColor",
      String.format("#%06X", button.
      getCurrentTextColor() & 0xFFFFFF));
    int[] location = new int[2];
    button.getLocationOnScreen(location);
    newElement.setAttribute("xPos",
      location[0] + "");
    newElement.setAttribute("yPos",
      location[1] + "");
    element.appendChild(newElement);
}
```

Fig. 1. Aspect code that captures a button's properties.

II. Cloud Twin Design and Implementation

Cloud Twin first automatically translates the initial screens of the source. Then it continuously captures, transmits, and replays user actions and UI updates, until the user switches to another application. In the following discussion, we detail each major component of Cloud Twin in sequence.

A. Preprocessing

At the source application, Cloud Twin intercepts the execution point at which the UI is created but not yet displayed to the user. Then it examines the runtime UI tree to detect the constituent interface components and the listeners attached to them. This step of the approach extends on our prior work [1]. The interception is accomplished by means of Aspect-Oriented Programming (AOP) (we use AspectJ [12]), and the UI tree is walked by means of reflection. Cloud Twin features an aspect library that can capture all the UI components for the Android platform [4]. Figure 1 shows sample AOP code to intercept a button object and its properties. Thus, Cloud Twin assumes that the source application's language has an AOP extension and supports reflection, an assumption that holds true for the majority of mobile platforms.

B. Runtime Processing

Cloud Twin represents the running source application as a collection XML structures, modeling each application view. The mapping between the actual UI and its XML representation is defined for each major Android [4] layout and component including: `LinearLayout`, `ScrollView`, `ListView`, `RelativeLayout`, `Spinner`, `Button`, `TextView`, and `MapView`. Figure 2 shows a sample Android source XML layout and the corresponding intermediate form's XML structure generated from the sample. Although this selection of UI elements does not encompass the full range of available layouts, it covers the most common cases. Furthermore, the Cloud Twin framework enables the developer to easily support customized UI elements.

```
//Android XML Page Layout
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
 "http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:orientation="vertical"
 android:id="@+id/mainPanel0">

   <TextView android:id="@+id/textView0"
   android:background="#000000"
   android:textColor="#FFFFFF"
      android:layout_width="fill_parent"
      android:layout_height="50dp"
      android:textSize="40dp"
      android:text="0" />
</LinearLayout>

// Intermediate Language XML
<?xml version="1.0" encoding="UTF-8"?><Layout>
<LinearLayout id="mainPanel0" height="430"
 width="320" background="null"
 visible="true" orientation="vertical">
<TextView id="textView0" height="50" width="320"
 background="#000000" visible="true" text="0"
 textSize="40.0" textColor="#FFFFFF" xPos="0"
 yPos="50"/>
</LinearLayout>
</Layout>
```

Fig. 2. Sample Android XML page layout and corresponding intermediate language XML.

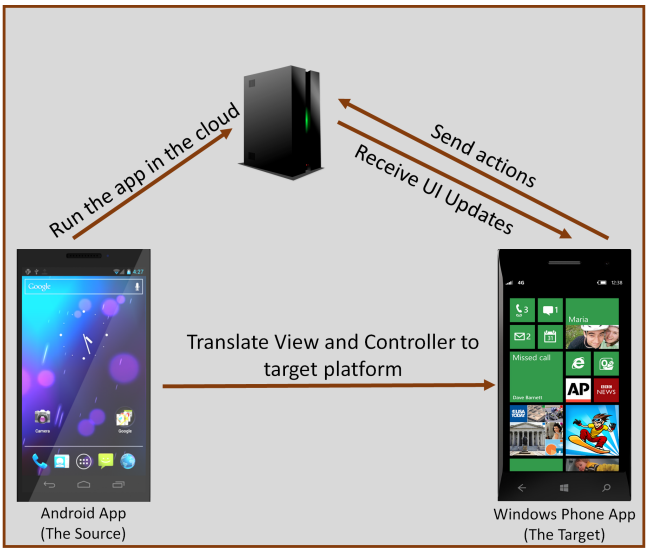

Fig. 3. The reference implementation of Cloud Twin.

C. The Source Setup

The source application runs on an emulator, whose functionality is exercised by means of an event handler. The emulator replays the UI events captured on the target. An aspect library intercepts all the updates to the source's views to encode and apply them to the target's view. The actions being triggered

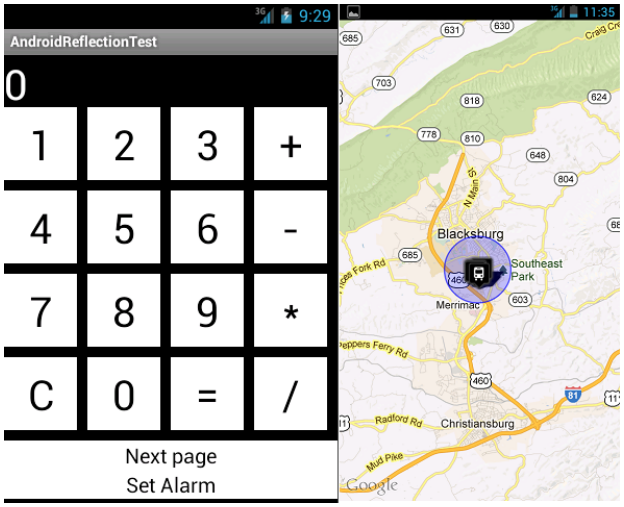

Fig. 4. Android Application (The Source)

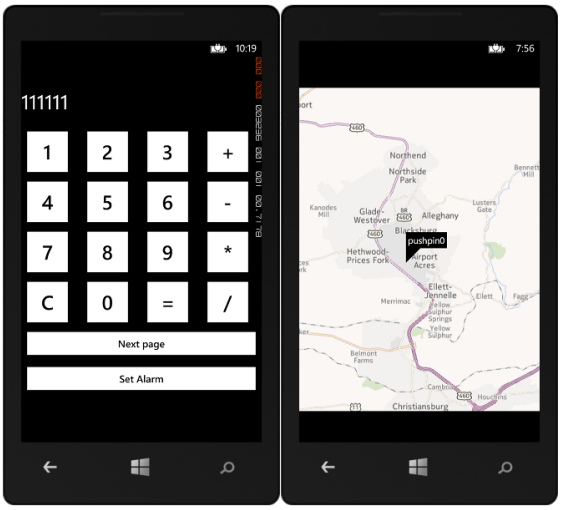

Fig. 5. Windows Phone Application (The Target)

on the target application related to the events taking place on the source application are similar in nature to the differences in client and server-side scripting.

A typical UI processing scenario is the target application executing a listener to process an event. However, the listeners are modified to capture the events they trigger, so that the triggered events' parameters are transmitted to the source. There, the event is replayed on the source application using the emulator, with the resulting differences in the source's UI sent back to the target. Thus, the source emulator has to repeat the UI interactions performed on the target.

Figure 3 outlines the execution process. To repeat the UI interactions on the source, Cloud Twin takes advantage of the MonkeyRunner test-automation framework [14], which interfaces with the emulator. To translate Windows Phone screen positions to equivalent Android emulator positions, Cloud Twin features a collection of simple Python scripts.

D. The Target Processing

Cloud Twin utilizes the XML intermediate form created by the source application to subsequently generate a target language application. In the reference implementation of Cloud Twin, Windows Phone applications are created by generating XAML and XAML.CS files from the intermediate XML files. The reference implementation utilizes Visual Studio 2012 [11] with WebSocket4Net [16] and WPtoolkit [17] plugins to expedite target application generation and processing.

When translating the intermediate XML files, each XML object's fields, such as height, width, id, position, etc., are converted to XAML fields. To process events, the corresponding objects' listeners [15] in the XAML.CS files are overwritten to communicate with the emulator via persistent web sockets. However, when it comes to referencing the target application for sensor processing (e.g., obtaining location or accelerometer data), the emulator utilizes another collection of aspect files. By modularizing these steps into intermediate objects, we aimed at making Cloud Twin extensible to be able to support other source and target mobile platforms in the future.

E. The Source Execution

The final major component of Cloud Twin is executing the target application by communicating with the source's emulator. This piece of functionality takes advantage of web sockets with persistent connections as a means of ensuring efficient distributed asynchronous communication. One of the key design objectives of Cloud Twin is to ensure that remotely replaying UI interactions does not incur high latencies. Using persistent web sockets reduces the aggregate latencies by establishing and maintaining a single connection. The Cloud Twin communication protocol is also fundamentally asynchronous and event-based. As a result of these design decisions, we have never experienced the latency of processing a single UI interaction surpassing the one second boundary.

III. Evaluation

In our preliminary evaluation of Cloud Twin, we aimed at showing that the approach is useful and feasible. To demonstrate the usefulness of Cloud Twin, we executed several small Android applications on the Windows Phone. To show feasibility, we micro-benchmarked the latency of Cloud Twin processing various UI events.

The example applications that we managed to execute successfully came from open-source tutorials for programming the Android platform. One such application combines the features of a calculator, a map, and an alarm. Figures 4 and 5 shows the screenshots of this application in the source Android version and the target Windows Phone version, respectively. The calculator functionality demonstrates how Cloud Twin supports the use of major UI components, including buttons, text boxes, and linear layouts. In addition, this application demonstrates an interesting range of smartphone functionality, such as the use of the GPS receiver and the alarm facility. In particular, to provide a meaningful user experience, the

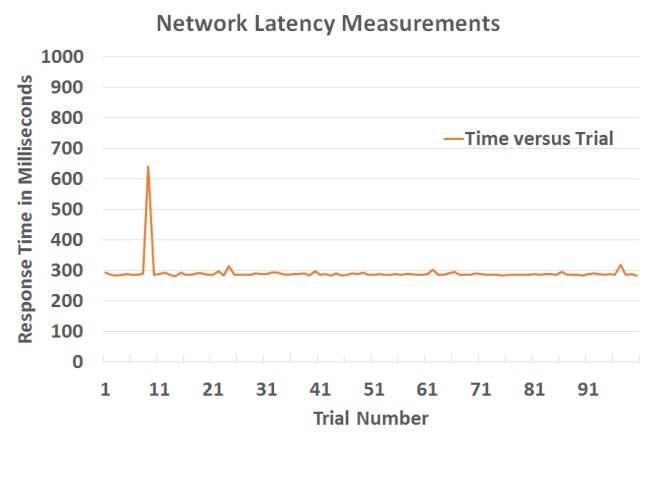

Fig. 6. Measurements of the latency of the network.

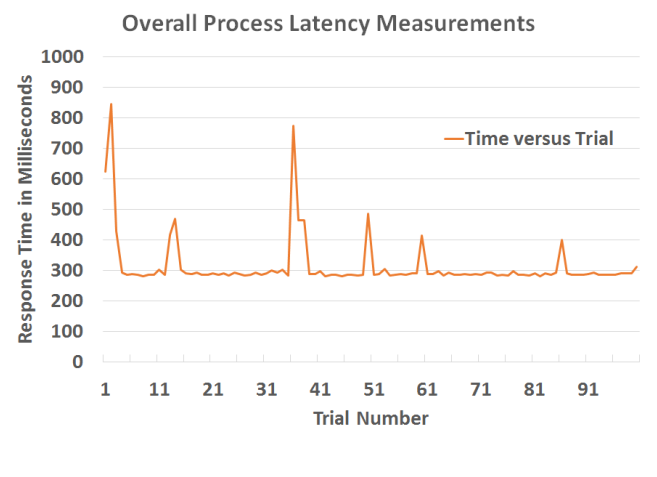

Fig. 7. Measurements of the latency of the overall UI update process.

executed application must use the GPS receiver and alarm of the Windows Phone (the target application). To that end, Cloud Twin redirects such location-specific operations to the target application by intercepting the appropriate API calls in the source by means of aspects.

To assess how feasible the Cloud Twin approach is, we isolated the following latency parameters:

1) Latency of Network Communication
2) Latency of a Complete UI Update
3) Latency of Interacting with Windows Phone

We have measured the latency of network communication and that of performing a complete UI update. Based on these measurements, we then extrapolated the latency of interacting with the Windows Phone when using Cloud Twin.

A. Latency of Network Communication

We isolated the latency of network communication using web sockets by measuring the total time it takes to send a message from the target's web socket and receive a response (i.e., a network roundtrip). Thus, this measurement sheds light on how long it takes to send a message, to process the message at the source, and to transmit a response back to the target. The physical network used in this benchmark is a Wi-Fi network of 100mbps.

Figure 6 shows the numbers for a 100 network roundtrips. For each roundtrip, the average latency is about 290 milliseconds, with a maximum of 641 milliseconds and minimum of 281 milliseconds. Although Wi-Fi networks may not always be available when executing mobile applications, these latency numbers should be comparable to using a high-end cellular network such as 4G.

B. Latency of a Complete UI Update

We isolated the latency of a complete UI update by measuring the total time it took between pressing a button on the target application and updating a text label in response. This measurement encompasses the following sequence of events: (1) the button pressed, (2) the resulting event is captured and transmitted to the source application, (3) the press is replayed on the source, (4) the text label update is intercepted, (5) the update is sent back to the target, (6) the target's label is updated with the received data. Thus, this measurement demonstrates a realistic response time a user would encounter when interacting with the reference implementation of Cloud Twin. Note that this measurement includes the latency of network communication discussed above.

Figure 7 shows the numbers resulting from repeating the measured operation a 100 times. The overall average latency was 314 milliseconds, with a maximum of 846 milliseconds and minimum of 281 milliseconds. The measured UI scenario is typical for modern user interfaces. The important insight is that the response time never exceeded the one second threshold, thus not compromising the user experience [19]. Future work will assess whether Cloud Twin can achieve comparable efficiency when processing more complex UI scenarios.

C. Latency of Interacting with Windows Phone

We isolated the latency of the user interacting with a Windows Phone target application by computing the differences between the results of the complete UI update benchmark and the average network latency (i.e., 291.05 milliseconds). Figure 8 show the resulting extrapolated latency incurred by the Windows Phone device itself. The average latency of only 23.15 milliseconds, with a maximum of 554.95 milliseconds (846 - 291.05) and minimum of -10.05, indicate that the Windows Phone executes quite efficiently. However, there are still some peaks with higher than average latency that could potentially jeopardize the user experience.

IV. Discussion

A. Advantages

Cloud Twin provides several software engineering benefits when companies need to make their mobile applications

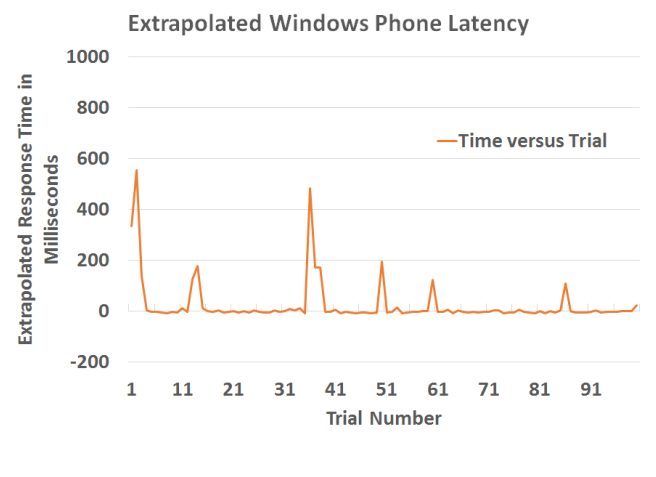

Fig. 8. The theoretical latency of the Windows Phone emulator.

available on multiple heterogeneous platforms. When fully realized, the Cloud Twin platform should be able to support the principle of "Write Once, Run on any Mobile Platform." Not only does Cloud Twin execute mobile applications on other platforms, but it does so natively, using the platform UI components and sensors. In the presence of a high-end network, Cloud Twin imposes only a modest latency overhead, almost undetectable for most users. Executing the majority of functionality on the edge server also provides the benefits of saving battery power [8]. The net effect of these advantages leads to reducing software development efforts and costs as well as reducing power consumption.

B. Limitations

The main limitation of Cloud Twin lies in its range of applicability. Applications with custom UI components, especially those that make use of animations, can not be translated by Cloud Twin without additional input from a user. One such domain is mobile gaming, for which the Cloud Twin approach is inapplicable. Thus, we envision that the primary beneficiaries of the Cloud Twin technology would be business users who want to access productivity-enhancing applications on their own mobile devices. Business applications commonly feature a standard UI structure, which Cloud Twin already supports natively. If some customized UI components become mainstream, the Cloud Twin framework can be extended to support them, thus benefiting the average user.

Another drawback to Cloud Twin is that the latency of multiple operations is not composable: the aggregate latency is the sum of the time taken by each individual UI operation. To optimize this inefficiency, we could batch individual UI operations to minimize the network communication latency. We may indeed pursue this optimization as a future work direction. Having said that, our experiences with the reference implementation of Cloud Twin indicate that the average latency never becomes unreasonably high to frustrate the user.

Another potential limitation of Cloud Twin concerns how the approach affects security, particularly the permission scheme in place. Although the distributed communication in Cloud Twin is indistinguishable from those in standard Web-enabled mobile applications, the very introduction of network interaction may weaken security.

The approach also circumvents the built-in mobile permission schemes on both the source and the target to enable emulated execution and sensor access. The assumption is that the source is secured on a server. However, in future work, we plan to explore with supporting more fine-grained permission schemes during the translation process. Thus far, we have not taken a close look at security. Once the approach matures, an increased focus will be placed on security.

Yet another limitation to Cloud Twin's applicability is lacking support for platform-specific hardware, such as Near Field Chip (NFC) on Android. Cloud Twin cannot support the UI interactions that involve platform-specific hardware, without equivalents on all major platforms. However, the extensible architecture of Cloud Twin makes it possible to support newly introduced hardware components.

Lastly, the overhead of maintaining cloud-based emulators can become an obstacle to widespread use of the technology. Cloud Twin requires the availability of numerous dedicated edge servers, as running multiple emulators may take inordinate amounts of memory and processor resources. Depending on the underlying cloud infrastructure cost model, Cloud Twin may not scale well for high-volume commercial deployments.

V. Related Work

Cloud Twin builds on our prior work in which we used aspect-oriented programming and reflection to reverse-engineer UIs at runtime with the purpose of subsequently translating them to other platforms [1]. Cloud Twin employs the same strategy for extracting UI elements. Specifically, this mechanism is used to produce the initial UI screen of the target application. While in our prior work, we focused on extracting UIs and statically translating them to multiple additional platforms, Cloud Twin translates and updates UIs across platforms continuously at runtime.

Cloud Twin conceptually relates to the work performed to map various platform APIs to one another. Mobile platform vendors commonly provide publicly accessible mappings that show which APIs of the target platform can be used to emulate the functionality of the source platform. For example, Microsoft provides such mappings between Android and the Windows Phone [9]. These mappings specifically relate API calls from one language to equivalent API calls in the other language in a dictionary-like fashion. Cloud Twin differs by using an intermediate form that abstracts away the logic of either language. Thus, Cloud Twin differs by lending itself to being easily extended to other platforms and languages. As long as the source language can be represented by means of the Cloud Twin intermediate language, the source platform application can be supported on other target platforms.

The intermediate UI form of Cloud Twin resembles the universal UI representations of independent UI models, such

as those used in UIML [10] and the aforementioned PhoneGap [18]. UIML and PhoneGap enable platform independent design and development of user interfaces. UIML employs an XML base language to subsequently generate user interfaces in a desired language. However, these and other platform independent approaches require that mobile applications be constructed using a particular language and the accompanying framework. By contrast, Cloud Twin assumes that mobile applications have already been constructed using their native platform APIs. Thus, Cloud Twin enables the execution of such applications natively on other mobile platforms.

VI. Future Work and Conclusions

The reference implementation of Cloud Twin makes it possible to execute Android applications natively on the Windows Phone. The ultimate goal of Cloud Twin is to combine the benefits of full source-level cross-platform porting and designing for a particular mobile platform. The reference implementation has demonstrated that the Cloud Twin approach is feasible and useful. However, to turn Cloud Twin into a practical software tool, we plan to enhance our implementation along the following lines.

A. Increasing the UI Component Coverage

Since more UI components are available than the ones chosen for the Cloud Twin prototype, extending the existing set would allow for more complete coverage when translating the UI tree. Additionally, customized components that are largely used within the community, such as the Android Sherlock Action Bar, could be added to the existing set.

B. Supporting Additional Platforms

The reference implementation can only translate from an Android-based application to a Windows Phone-based application. However, Cloud Twin can be extended to other source platforms. As the only requirements are the support of AOP and having a test-automation framework, other mobile platforms are amenable to this approach as the source, including the Windows Phone and iOS platforms. A fixed intermediate form and the modular nature of the Cloud Twin implementation should facilitate future extensions.

In addition to source platforms, Cloud Twin could also be extended to allow additional target platforms, as the benefits multiply when generating a plurality of target applications. Utilizing the existing intermediate form as a starting point and using the current basic grammar as a template, future work could extend Cloud Twin to support the targets including Android and iOS.

C. Automating Adoption of API Evolution

Based on the work described in projects, including Rosetta [2] and MAM [3], Cloud Twin can be enhanced with the ability to automatically update itself in response to the changes in source and target APIs. The current Cloud Twin implementation defines exactly which components on Android are written to the intermediate language and exactly what those intermediate language components are written to in the Windows Phone. The ideas presented in Rosetta and MAM allows for analysis of different language APIs to map them to each other. Enhancing with such technology, Cloud Twin should be able to continuously update itself in response to the updates in platform APIs by analyzing the differences between the source and target APIs in different application versions.

D. Collaborative Mobile Applications

Based on the work in Cloud Twin to abstract an application's UI away from its backend source, it is conceivable that multiple applications could be combined into one by connecting to multiple sources at once via one unified UI. One group UI could easily be comprised out of multiple translated UIs by simply adding page changes in each individual UI to direct control flow. Then each translated UI could communicate separately with its own backend source via its own emulator service. The ability to dynamically combine arbitrary application UIs could foster novel mobile development paradigms.

Acknowledgments

This research is supported by the National Science Foundation through the Grant CCF-1116565.

References

[1] E. Shah and E. Tilevich. Reverse-engineering user interfaces to facilitate porting ot and across mobile devices and platforms. In *Workshop on Next-generation Applications of Smartphones*, 2011.

[2] A. Gokhale, V. Ganapathy, and Y. Padmanaban. Inferring Likely Mappings Between APIs. In *ICSE*, 2013.

[3] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *ICSE*, 2010.

[4] Google. Android API reference. http://developer.android.com/reference/packages.html.

[5] Google. Google Maps Android API v2. https://developers.google.com/maps/documentation/android/.

[6] Apple. Map Kit Framework Reference. http://developer.apple.com/library/ios/documentation/MapKit/Reference/MapKit_Framework_Reference/.

[7] Microsoft. Bing Maps APIs. http://msdn.microsoft.com/en-us/library/dd877180.aspx.

[8] B. Zhao, B. C. Tak, and G. Cao. Reducing the Delay and Power Consumption of Web Browsing on Smartphones in 3G networks. In *ICDCS*, 2011.

[9] Windows phone interoperability: Windows phone API mapping. http://windowsphone.interoperabilitybridges.com/porting.

[10] M. Abrams, C. Phanouriou, A. L. Batongbacal, S. M. Williams, and J. E. Shuster. UIML: an appliance-independent XML user interface language. *Computer Networks*, 31(11-16):1695-1708, 1999.

[11] Microsoft. Visual Studio 2012. http://www.microsoft.com/visualstudio/eng/products/visual-studio-ultimate-2012#product-edition-ultimate.

[12] Eclipse. AspectJ: Crosscutting Object for Better Modularity. http://www.eclipse.org/aspectj/.

[13] Eclipse. Eclipse Downloads. http://www.eclipse.org/downloads/.

[14] Google. MonkeyRunner Android Development Tools. http://developer.android.com/tools/help/monkeyrunner_concepts.html.

[15] Microsoft. Windows Phone API Reference. http://msdn.microsoft.com/en-us/library/windowsphone/develop/ff626516(v=vs.105).aspx.

[16] Apache. WebSocket4Net. http://websocket4net.codeplex.com/.

[17] Microsoft. Windows Phone toolkit 4.2012.10.30. http://phone.codeplex.com/.

[18] R. Ghatol and Y. Patel. Beginning PhoneGap: Mobile Web Framework for JavaScript and HTML5. Apress, 2012.

[19] R. B. Miller. Response Time in Man-Computer Conversational Transactions. In *AFIPS*, 1968.