A Common Platform API for Android

Arno Puder

San Francisco State University
Computer Science Department
1600 Holloway Avenue
San Francisco, CA 94132
arno@sfsu.edu

Abstract. Cross-platform frameworks for mobile devices promise to facilitate the porting effort of applications between different smartphones. Our approach is to cross-compile Android applications to other platforms such as iOS or Windows Phone 7. Doing so requires to refactor the Android source code base in a platform-dependent and platform-independent part separated by a *Common Platform API*. This paper discusses the cross-compiling of Java-based Android applications and the design and implementation of the Common Platform API.

1 Introduction

Smartphones have become the major driving force in the mobile market. Currently iOS and Android dominate the scene with Microsoft's Windows Phone 7 (WP7) and HTML5-based platforms such as Tizen or Firefox OS vying for market share. From a developers perspective it is desirable to be present in as many app stores as possible to increase dissemination and thereby revenue. However, making an application available on different platforms requires significant efforts. This has to do with the fact that smartphone platforms have developed into technology silos where cross-platform approaches are made difficult through technical and legal means. Apple in particular has tried in the past to ban other execution platforms other than its own on iOS. Making an application available on different platforms necessitates to reimplement it in a different programming languages. Android uses Java, iOS uses Objective-C while WP7 requires either C# or VisualBasic [6,5,2] (see Figure 1).

To some extend Android is the most liberal smartphone platform, not only because its core code base is released under an Open Source license [1]. Android was designed to run on a variety of devices with different hardware capabilities. An Android developer is expected to write applications in such a way that they adapt to specific capabilities (such as different screen resolutions). For this reason we have chosen Android as the canonical platform for our cross-platform framework, called XMLVM [9]. Android applications can be cross-compiled to other platforms with the help of our byte-code level cross-compiler. The cross-compiled application should have the look-and-feel of the target platform. E.g., an Android button should be mapped to the native button of the respective platform.

© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2013

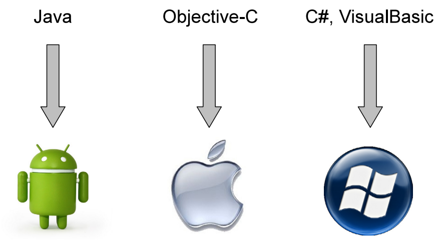


Fig. 1. Technology silos

In previous work we have shown how to cross-compile from Android to iOS [7]. However, the necessary changes to the Android code base were tightly linked to iOS. Targeting another platform such as WP7 would have required to redo this work leading to two independent variations of the Android library that need to be maintained separately. Instead, we decided to refactor the Android code base in such a way that platform-dependent parts are clearly separated from the platform-independent parts by what we call the *Common Platform API* (CP-API). Ultimately, the refactored code base increases reusability and maintainability of our Android compatibility library.

This paper is organized as follows: Section 2 discusses the limits to cross-platform approaches and Section 3 presents various cross-platform frameworks. Section 4 introduces our cross-platform approach and more specifically the design of the CP-API. In Section 5 we briefly discuss our prototype implementation before providing conclusions and an outlook in Section 6.

2 Limits to Cross-Platform Frameworks

Cross-platform frameworks are hampered by legal and technical hurdles. Legal limitations are often tied to UI style guidelines. E.g., Apple will reject applications that have an “Exit” button to terminate the application. In iOS the only permitted way to exit an application is via the device’s home button. Apple also permits the use of dynamic execution technology such as virtual machines for the purpose of loading additional code from a server.

In this section we focus on the limits of cross-platform approaches from a technical perspective. Table 1 gives an overview of some key differences between Android, iOS, and WP7 both from the hardware and software perspective. Developers for Android and WP7 devices can expect the presence of certain hardware buttons such as menu or search buttons. If the application is to be ported to a platform without those buttons (e.g., iOS), the same functionality needs to be incorporated into the UI in a different way. Likewise there exist differences in the physical screen resolutions between the platforms. Android makes no assumption on the screen resolution while iOS and WP7 being closed systems prescribe a limited number of resolutions.

Table 1. Comparison between Android, iOS, and WP7

	Android	**iOS**	**WP7**
Buttons	Back, Menu	None	Back, Search
Screen Res	Flexible	Limited	Limited
Language	Java	Objective-C	C#, VisualBasic
Memory Mgt	Garbage Collection	Reference Counting	Garbage Collection
Layout	Declarative	Absolute	Declarative
Intents	Yes	No	No

From the software side, each platform uses a different programming language. Cross-compiling between Turing complete languages is possible, so this does not present an obstacle [3]. However, iOS uses reference counting for memory management, so any cross-platform framework would need to address this. When it comes to layouting a UI, Android and WP7 support declarative UI descriptions while iOS expects the programmer to place every widget in terms of absolute coordinates. Android introduced a powerful late binding mechanism called intents that has even been adopted in other frameworks (e.g., W3C's WebIntents [10]). However, iOS and WP7 do not offer a comparable feature. Here again the question arises how missing functionality of one platform can be compensated on another.

The list presented in Table 1 is by no means exhaustive. There are numerous other differences between platforms. E.g., in Android, the label inside a button can be placed anywhere within the borders of the button. iOS and WP only allow the label to centered. The implication of these differences is that cross-platform frameworks will need to make compromises. Either an application will need to settle for a lowest common denominator in terms of functionality, or extra efforts must be made to overcome platform differences.

While it would be possible to provide a custom widget under iOS that looks like a button and that can place its label in the top-right corner, doing so is not advisable since it would break that platforms UI idioms. The challenge of cross-platform frameworks is to provide some best practices that make it easier to port an application. While certain features should not be used, as will be shown in a subsequent section, it is possible to mimic other features without compromising the UI idiom of a platform. The following section discusses various cross-platform frameworks.

3 Related Work

Several frameworks promise to facilitate the development of cross-platform applications. In the following we briefly discuss the approach taken by Cordova, Adobe AIR, and In-the-Box. Each framework will be classified with regards to the mobile platforms it supports, the programming languages it offers, the API it uses, the IDE it can be used with and finally the license under which it is released.

Apache Cordova (formally called PhoneGAP) is an Open Source project that addresses web developers who wish to write mobile applications. It is available for iOS, Android, WP7 and other platforms. Applications need to be written in JavaScript/HTML/CSS. But instead of downloading the application from a remote web server, the JavaScript is bundled inside a native application. E.g., for iOS devices a generic startup code written in Objective-C will instantiate a full-screen web widget via class `UIWebView`. Next the JavaScript that is embedded as data in the native application is injected into this web widget at runtime. Special protocol handlers allow the communication between JavaScript and the native layer. All iOS widgets are rendered using HTML/CSS mimicking the look-and-feel of their native counterparts. Cordova supports a common API for sensors such as the accelerometer. Platform-specific widgets have their own API. Cordova is available under the MIT Open Source license at `http://incubator.apache.org/cordova/`.

Table 2. Comparison of Cross-Platform Frameworks

	Cordova	**In-the-Box**	**Adobe AIR**	**XMLVM**
Platforms	iOS, Android, WP7, others	iOS	iOS	iOS, Android, WP7
Language	JavaScript	Java	ActionScript	Java
API	Common Sensor API	Android	Graphics-only	Android API mapped to iOS, WP7
IDE	Xcode	Eclipse	N/A	Eclipse
License	Open Source	Open Source	Commercial	Open Source

Another cross-platform framework is the Adobe Integrated Runtime (AIR) for iOS development. Adobe AIR includes an (Ahead of Time) AOT compiler based on the LLVM compiler suite that translates ActionScript 3 to ARM instructions. This facilitates porting of existing Flash applications while not relying on an installation of a Flash player on the iOS device. AIR offers API based on ActionScript to the device's sensors, but does not provide access to the native iOS widgets which limits AIR applications to games. AIR is available under a commercial license at `http://www.adobe.com/products/air/`.

A project called In-the-Box takes yet another approach: Android's virtual machine, called Dalvik [4], is ported to iOS to execute original Android applications under iOS. From iOS perspective, Dalvik and the class files comprising the Android app are bundled into one native binary. Back in 2009 Apple relaxed the terms and conditions of their SDK to allow such deployments. The benefit of this approach is complete Android compatibility. However, one major downside is that In-the-Box creates iOS apps that have the look and feel of Android applications. It can also not solve the problem of Android's hardware button that do not exist under iOS. In-the-Box is released under the Apache Software License and is available at `http://www.in-the-box.org/`.

Table 2 summarizes the distinguishing factors of the various cross-platform frameworks. Our framework XMLVM is similar in the respect that it offers one programming language (Java) for different mobile devices. It also includes an AOT compiler to translate Java to native applications in order to avoid the installation of a Java virtual machine on the target platform. Similar to In-the-Box, XMLVM also relies on the Android API for application development. However, one major difference to In-the-Box is that the Android API is mapped to the native API of the respective platform. E.g., an Android button is mapped to a native `UIButton` when cross-compiled to iOS.

4 Cross-Compiling Android Applications

This section provides some details of our cross-compilation framework. First, we discuss the its design principles. Next we describe how to expose non-Java API of the target platform in Java followed by the introduction of the Common Platform API.

4.1 Design Principles

Our approach is to cross-compile Android applications to other platforms such as iOS or WP7. Considering the unique features of every platform, it is not possible to cross-compile arbitrary applications. Certain best practices must be followed such as not making use of the Android menu button. It is important that the cross-compiled application uses the UI idioms of the target platform. It is not acceptable to have and iOS or WP7 application that looks and feels like an Android application as is the case of the aforementioned In-the-Box framework. The implication is that certain features in Android will not be mimicked on the target platform. E.g., if an Android application does not place the label in the center of a button, the cross-compiled version would still do so.

In order to accomplish this, the Android code base needs to be refactored in such a way that Android widgets can easily be mapped to their native counterpart of the target platform. The goal of the refactoring is to distinguish between platform-dependent and platform-independent parts of Android. Significant portions of Android are platform-independent and can be cross-compiled as-is to the target platform. Most importantly, Android's layout manager, activity lifecycle and the intent system have little dependence to the native layer. E.g., the layout manager reads declarative layout descriptions from the file system to compute a layout. The implementation of the various layout managers such as `LinearLayout`, `RelativeLayout`, or `GridLayout` have no other external dependencies.

The refactoring yields a Common Platform API (CP-API) that isolates the platform-dependent parts of Android. Adding a new platform will only require to implement the CP-API. Designing the CP-API is the contribution of this paper and will need to balance re-use of the existing Android code base vs. the ability to do a deep integration to achieve the native look-and-feel of the target platform. The following section first discusses the adding of a Java layer over a non-C platform followed by a description of the CP-API.

4.2 JNI for Non-C Platforms

The first step is to expose the native API of the target platform in Java. That is to say, an API expressed in a language L needs to be accessible in Java. Given a solution to this problem, it is possible to write apps for this platform in Java using the native API of that platform. The challenge consists in the fact that the native programming language may follow different paradigms than Java. E.g., Objective-C used for iOS development supports dynamic typing and the memory management mechanism is based on reference counting. While creating a Java API from a platform's native API is mostly a mechanical process, one has to decide how to generate strongly typed interfaces common to Java programming based on an API that exploits dynamic typing. We have studied this problem in earlier work [8].

Once a Java API has been generated, the question remains how an invocation of a Java method results in a call to the corresponding native method. The Java Native Interface (JNI) [6] specification introduced a mechanism by which a Java application can break out from the VM sandbox to access the native layer. JNI describes how data structures are passed between the VM and C-based applications. Since the JNI is limited to the C programming language, it cannot be used for platforms that do no provide access to the C layer.

For that reason we have extended the JNI model by keeping the Java interface (via the `native` method modifier) and allowing arbitrary programming languages on the native side. In the following we give an example how the API of class `Button` in WP7 can be exposed in Java. Class `Button` extends from base class `ButtonBase` in WP7 and has amongst others a method `setContent()` to set the label of the button. This method is marked as native and consequently has no implementation:

Java: WP7 Button Wrapper

```java
public class Button extends ButtonBase {
  native public void setContent(String content);

  //...
}
```

As can be seen in the listing above, the implementation of the class is left empty since its only purpose is to provide a Java API against which the developer can implement an application. Properties in C#, such as `Button.Content`, are represented by appropriate getter/setter methods. Our cross-compiler translates the wrapper class to the target language; C# in this case. For methods marked as native the cross-compiler inserts special comment markers into the generated code. The programmer can inject manually written code between these comment markers. This code is tying the wrapper class together with the native class it wraps. The following code excerpt demonstrates this concept for the `Button` class.

C#: Cross-compiled WP7 Button Wrapper

```
public class Button : ButtonBase {

  public virtual void setContent(java.lang.String n1) {
    //XMLVM_BEGIN_WRAPPER
    wrapped.Content = Util.toNative(n1);
    //XMLVM_END_WRAPPER
  }

  //...
}
```

Note that the wrapper class above is not implementing the widget itself, but only wraps the WP7 API `Button` class. Code between `XMLVM_BEGIN_WRAPPER` and `XMLVM_END_WRAPPER` comments is manually written C# code which gets injected on either method- or class-level during cross-compilation. The comment markers allow the manually written code to be automatically migrated if it should become necessary to regenerate the wrappers. Method `setContent()` converts a `java.lang.String` instance to a native C# string via a helper function and sets the `Content` property of the wrapped button to the converted string. Once the native API of the target platform has been exposed in Java, it is possible to implement the platform-specific portions of the refactored Android code base.

4.3 Common Platform API

The Common Platform API, CP-API for short, isolates the platform-specific parts of Android. The platform-independent parts can be reused while the CP-API needs to be implemented for each target. In the following we discuss the CP-API for the view hierarchy. Every UI framework features a view hierarchy featuring a variety of widgets. The view hierarchy typically has a common base class from which the various widget classes are derived. Android's base class of the view hierarchy is class `View`, the base class for iOS is `UIView` and for WP7 the class is called `Panel`. The base class combines various capabilities that are inherited to all derived classes. The following code excerpt shows the API for setting a background color/image:

Java: View hierarchy

```
// Android
public class View {
  native public void setBackgroundDrawable(Drawable d);
  // ...
}

```

```
// iOS
public class UIView {
  native public void setBackgroundColor(UIColor c);
  // ...
}

// WP7
public class Panel {
  native public void setBackground(Brush b);
  // ...
}
```

In Android the background of a widget can be an arbitrary `Drawable`. A `Drawable` can be static color, a gradient, an image, or a custom drawable where the application can manually draw the background. iOS is more restrictive and only allows a background to be a static color (represented via class `UIColor`). More complex backgrounds in iOS require to place a separate `UIView` that serves as the background. WP7 is more flexible by allowing the background to be a `Brush`. A `Brush` can be a static color, a gradient, or an image. But unlike Android it is not possible to provide an application-specific custom `Brush`.

Considering the differences in functionality, the CP-API introduces a Java interface that serves as an abstraction for the common base class of the view hierarchy:

Java: CommonView interface

```
public interface CommonView {
  public void setBackgroundDrawable(Drawable d);
  // ...
}
```

Given this interface, the question arises how to implement it under iOS and WP7. In case a specific `Drawable` is supported by the respective platform, it can be mapped directly to the native API. E.g., a solid color `Drawable` can be directly mapped to an appropriate `UIColor` under iOS and a `Brush` under WP7. The more interesting case is when the `Drawable` is not supported by the native platform. In this case we rearrange the view hierarchy by adding an extra view that represents the background as shown in Figure 2. If the application sets the background on view V_3, a new view B_3 is inserted into the view hierarchy. View V_3 is the child of B_3 and its size and position are changed such that V_3 completely overlaps with B_3. It is then possible to render the `Drawable` in B_3. Since the Z-order of B_3 is such that it is below V_3 it effectively serves as the background.

Interface `CommonView` therefore serves as an abstraction of the platform-specific portions of an Android `View`. A platform-specific implementation has to be provided based on the native API. The device-independent portions of Android need to be refactored to make use of the interface. Instantiating platform-specific views is done via a factory. The main entry point to the CP-API is a singleton

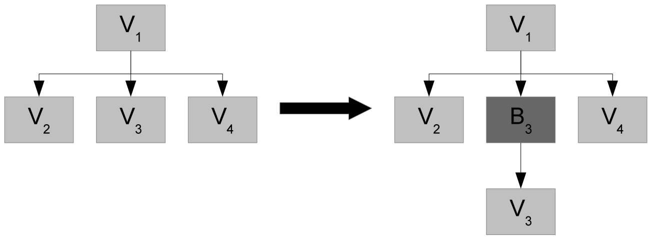


Fig. 2. Adding a background to a view

implementing the `CommonPlatformAPI` interface that provides access to the various subsystems:

Java: CP-API and Widget Factory
```java
public interface CommonPlatformAPI {
  CommonFileSystem getFileSystem();
  CommonAccelerometer getAccelerometer(SensorManager sensorManager);
  CommonWidgetFactory getWidgetFactory();
  CommonFontFactory getFontFactory();
  CommonPowerManager getPowerManager();
  CommonMediaPlayer getMediaPlayer(MediaPlayer mediaPlayer);
  // ...
}

public interface CommonWidgetFactory {
  CommonView createCommonView();
  ButtonAdapter createButton();
  ImageViewAdapter createImageView();
  TextViewAdapter createTextView();
  RadioGroupAdapter createRadioGroup();
  // ...
}
```

The `CommonWidgetFactory` interface can be obtained via the top-level `CommonPlatformAPI` interface and it allows the creation of the various Android widget adapters. The adapter interfaces declare the platform-specific API of the corresponding Android widgets ensuring reduced overhead for the refactoring of the platform-independent portions. The following code excerpt shows the declaration of the `ButtonAdapter` interface:

Java: ButtonAdapter interface
```java
public interface ButtonAdapter extends CommonView {
  void setText(CharSequence text);
  void setOnClickListener(OnClickListener listener);
  // ...
}
```

The interface features a subset of the methods declared in the Android class `Button`. In the following we show how the two featured methods of `ButtonAdapter` are implemented for iOS and WP7:

Java: Implementation of ButtonAdapter for iOS

```java
public class IOSButtonAdapter implements ButtonAdapter {

  private UIButton nativeButton;

  public IOSButtonAdapter() {
    nativeButton = UIButton.buttonWithType(UIButtonType.RoundedRect);
  }

  @Override
  public void setText(CharSequence text) {
    nativeButton.setTitle(text, UIControlState.Normal);
  }

  @Override
  public void setOnClickListener(final OnClickListener listener) {
    nativeButton.addTarget(new UIControlDelegate() {

      @Override
      public void raiseEvent(UIControl sender, int eventType) {
        listener.onClick(IOSButtonAdapter.this);
      }
    }, UIControlEvent.TouchUpInside);
  }
}
```

Class `IOSButtonAdapter` is a wrapper of a native iOS `UIButton`. The methods declared in interface `ButtonAdapter` are implemented based on the `UIButton` API, e.g., the `setText()` method is mapped to the corresponding `setTitle()` method of the `UIButton`. Another example is method `setOnClickListener()` that defines a delegate in the application to be called when the user taps on the button. The iOS `UIButton` features a method `addTarget()` that serves the same purpose. The iOS delegate has to implement a callback method called `raiseEvent()` that simply delegates the click event to the Android application. This example shows that for upcalls done by Android to the application it is possible to use the original Android interfaces (`OnClickListener`) and it is not necessary to create special wrapper interfaces in the CP-API.

Analogous to the iOS implementation, the following code excerpt shows the same implementation of the `ButtonAdapter`, this time for WP7:

Java: Implementation of ButtonAdapter for WP7

```java
public class WP7ButtonAdapter implements ButtonAdapter {
  private OnClickListener listener;
  private System.Windows.Controls.Button nativeButton;

  public WP7ButtonAdapter() {
    nativeButton = new System.Windows.Controls.Button();
  }

  @Override
  public void setText(CharSequence text) {
    nativeButton.setContent(text);
  }

  @Override
  public void setOnClickListener(OnClickListener listener) {
    this.listener = listener;
    nativeButton.Click.__add(new RoutedEventHandler(this,
                                                    "button_onClick"));
  }

  public void button_onClick(Object sender, RoutedEventArgs e) {
    listener.onClick(this);
  }
}
```

In this case `WP7ButtonAdapter` is a wrapper for a native WP7 `Button`. The `setText()` method here is mapped to the equivalent `setContent()` method. The previous section showed how the Java version of this method is routed to the native C# method via code injection. The Android click listener is installed via WP7's event and delegate model. Method `__add` is the Java version of C#'s overloaded $+=$ operator with which a delegate can be added to the `Click` event. Method `button_onClick()` will be called whenever the user pressed the WP7 button. Its implementation delegates the call to the Android application via the usual `OnClickListener`.

Figure 3 visualizes the structure of the refactored Android code base. The platform-independent portions are common to all supported platforms and contain modules such as layout management or activity lifecycle. Classes such as `android.widget.Button` are refactored into platform-independent parts that access platform-dependent implementations via interfaces of the Common Platform API. Adapter classes implement the CP-API based on features of the respective target platform.

5 Prototype Implementation

The concepts presented in this paper have been implemented as part of the XMLVM project. Android 2.3 served as a starting point for the refactoring effort. The platform-independent portions include the Activity lifecycle management, Intents, and layout management. The CP-API covers the majority of the Android widgets as well as the complete sensor API (accelerometer, gyroscope, GPS, camera, etc). Platform-specific implementations exist for iOS and WP7. Android applications complying to the best practices mentioned earlier

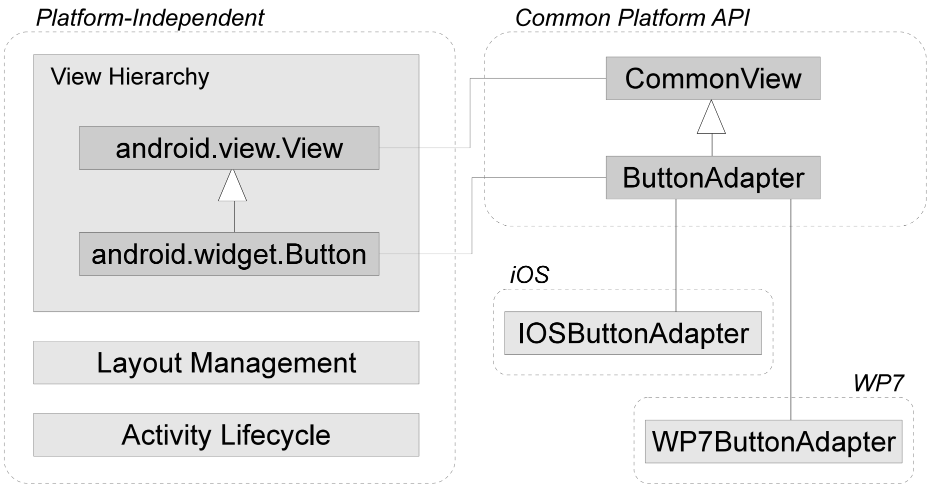


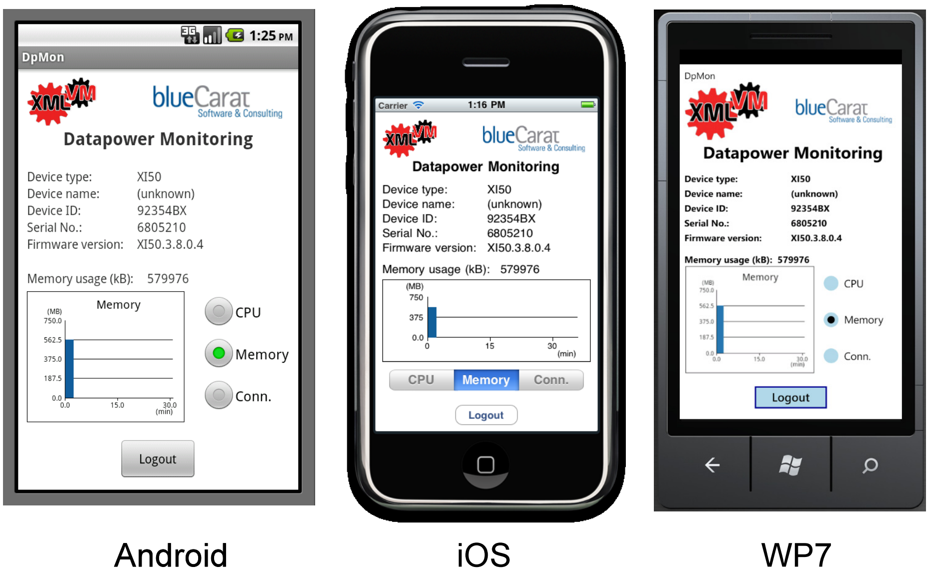
Fig. 3. Refactored Android code base

can be cross-compiled to Objective-C and C#. The refactored Android library is cross-compiled to those languages as well, yielding in native applications for the respective platform.

To demonstrate the feasibility of our approach we have cross-compiled an existing Android monitoring application. We have used the same application to show the cross-compilation from Android to iOS [7]. Based on the CP-API we have added a platform-specific version that allows the same application to be cross-compiler to WP7. The application issues HTTP requests to a network appliance and displays usage statistics in a custom widget that draws a graph. The original Android version uses a `RadioButton` group (see Figure 4). The corresponding `RadioGroupAdapter` of the CP-API maps this Android widget to a `UISegmentedControl` under iOS and a `RadioButton` under WP7. Since a `UISegmentedControl` is wider than high, Android's layout manager automatically stretches the custom graph-drawing widget, resulting in a native look-and-feel of the application on all platforms.

6 Conclusions and Outlook

Porting smartphone applications to various mobile platforms requires significant efforts. Various cross-platform frameworks seek to facilitate this process. The approach taken in this paper is to cross-compile Android applications to other platforms. It is important to keep the UI idioms of the target platform and not make the cross-compiled application look and feel like an Android application. To accomplish this the Android code base needs to be refactored in order to introduce a Common Platform API that isolates the platform-specific portions of Android.



Datapower Monitoring

Device type:	XI50
Device name:	(unknown)
Device ID:	92354BX
Serial No.:	6805210
Firmware version:	XI50.3.8.0.4

Memory usage (kB): 579976

Fig. 4. Example

This approach works well for Android applications that follow certain best practices, such as avoiding the use of the menu button. In some cases the best practices require unnatural workarounds in order to cross-compile an application. In the future we plan to investigate a partial cross-compilation approach where only certain portions of the Android application are cross-compiled. For those parts of the application that are not cross-compiled the developer would have to provide a customized implementation for the target platform that can exploit its capabilities that may not be present in Android.

Acknowledgements. Markus Neubrand and Oren Antebi have implemented the concepts described in this paper as part of their master thesis work at the San Francisco State University.

References

1. The Android Open Source Project. Dalvik eXchange (DX), `http://www.git://android.git.kernel.org/platform/dalvik.git`
2. ECMA. C# Language Specification, 4th edn. (June 2006)
3. El-Ramly, M., Eltayeb, R., Alla, H.A.: An Experiment in Automatic Conversion of Legacy Java Programs to C#. In: ACS/IEEE International Conference on Computer Systems and Applications, pp. 1037–1045 (2006)
4. Google, Inc. The Dalvik virtual machine, `http://en.wikipedia.org/wiki/Dalvik_virtual_machine`

5. Kochan, S.: Programming in Objective-C, 4th edn. Addison-Wesley Professional (December 2011)
6. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, 2nd edn. Addison-Wesley Pub. Co. (April 1999)
7. Puder, A.: Running Android Applications without a Virtual Machine. In: Venkatasubramanian, N., Getov, V., Steglich, S. (eds.) Mobilware 2011. LNICST, vol. 93, pp. 121–134. Springer, Heidelberg (2012)
8. Puder, A., D'Silva, S.: Mapping Objective-C API to Java. In: MobiCASE, Mobile Networks and Applications, Seattle. Springer (2012)
9. Puder, A., Lee, J.: Towards an XML-based Byte Code Level Transformation Framework. In: 4th International Workshop on Bytecode Semantics, Verification, Analysis and Transformation. Elsevier, York (2009)
10. W3C. WebIntents (2012), `http://www.w3.org/wiki/WebIntents`