RI 09009 April 27, 2009 Computer Science

IBM Research Report

MobiVine - A Middleware Layer to Handle Fragmentation of Platform Interfaces for Mobile Applications

Vikas Agarwal, Sunil Goyal, Sumit Mittal, Sougata Mukherjea

IBM Research Division
IBM India Research Lab
4, Block - C, Institutional Area, Vasant Kunj
New Delhi - 110070. India.

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

MobiVine - A Middleware Layer to Handle Fragmentation of Platform Interfaces for Mobile Applications

Vikas Agarwal, Sunil Goyal, Sumit Mittal, Sougata Mukherjea

IBM India Research Laboratory, New Delhi
{avikas, gsunil, sumittal, smukherj}@in.ibm.com

Abstract. Rapid enhancements in computing power, memory, display, etc., have propelled mobile phones as a platform to deploy and execute a variety of applications. To foster creation of rich mobile applications, popular platform vendors such as Android, iPhone and Nokia S60 offer extensive middleware support. This includes not only helping developers code and package their application modules in a format suitable for deployment, but also providing 'interfaces' to a) access information on the mobile device (for e.g. user location), and b) invoke device capabilities (like camera), from within the applications. Although usage of such *platform interfaces* leads to richer modules, it requires the developer to deal with application fragmentation arising due to heterogeneity in syntax, semantics and implementation of these interfaces across different platforms. In this paper, we first look into the problem posed by this fragmentation and characterize it's uniqueness in the mobile setting. Thereafter, we present 'M-Proxy', a semantically structured unit to absorb platform interface heterogeneity, and use it as a building block to develop 'MobiVine', a middleware 'de-fragmentation' layer for mobile applications. We demonstrate how MobiVine can be seamlessly integrated with existing platform middlewares using the notion of 'M-Plugins'. We also analyze and evaluate the effectiveness of MobiVine through implementations for three mobile platforms - Android, Nokia S60 and Android WebView.

Keywords - Mobile Platforms, Fragmentation, Abstraction Layer

1 Introduction

With rapid enhancements in computing power, memory, display, etc., mobile phones have emerged from being regarded as a mere communication device to being a platform for deploying and executing a plethora of applications. Today's devices enable not only simple applications like ring-tones and calendar, but also complex ones such as games and document processing softwares. In true essence, a mobile phone can be regarded as a mini-sized computer, drawing upon increased user expectation and expertise. New technologies are being invented to support enterprise and personal use of the mobile platform, including migration of legacy desktop applications to mobile platforms [2, 5], mobile device management, new user interfaces, mobile commerce, etc.

To help developers rapidly create rich mobile applications, popular platform vendors offer extensive middleware support. This includes allowing a developer access to different platform resources, such as the underlying operating system, middleware components, useful libraries and tools, etc. For instance, Android[1] provides a Linux-based open-source platform onto which independent third party developers can create and port their applications, while making use of services like search, Gmail and Google Maps in the process. Similarly, Nokia S60[2], iPhone[3] and many other platforms offer their own application development set-up. Furthermore, most platforms also expose interfaces that provide access to, from within the applications, a variety of richness available on the mobile handset - information (user's contacts, calendar, geographic location, etc.) as well as functionality (making calls, sending SMSes, using the camera, etc.). Richer applications for a mobile platform can be composed by combining application logic with these *platform interfaces*. For example, using the location information available on the mobile phone, one can design a number of location-based applications - directory services, workforce management solutions, etc.

In this paper, we look at the problem posed by fragmentation of mobile platform interfaces. As illustrated in the next section, these interfaces differ in both syntax and semantics, make use of platform specific data structures and properties, throw platform specific exceptions, and are also characterized by inconsistencies in implementation by different vendors. This has bearing on the portability of mobile applications across multiple platforms. It is important to note, however, that solutions have been proposed for similar problems in the desktop world. For example, standardization approaches such as POSIX[4], have been successful in providing uniform APIs for accessing OS services. Similarly, programming techniques such as abstraction through models aim to hide heterogeneity in a manner that makes the applications easily portable. Although attempts are being made to apply similar techniques in the mobile setting, they fall short of being equally effective due to various unique characteristics of this domain - (a) tight coupling of the application development and deployment model to platform middleware, (b) strong desire by vendors to offer differentiated API functionality to developers, and (c) rapidly evolving platforms and arrival of new ones.

Our contributions in this paper can be summarized as:

- We study the problem of application fragmentation due to heterogeneity in platform interfaces, and characterize the refinements needed to apply corresponding solutions in the desktop/OS world to a mobile setting.
- We propose 'MobiVine', a middleware layer based on 'M-Proxy' (a semantically structured unit to absorb fragmentation of platform interfaces) and 'M-Plugin' (an integration artifact) to handle fragmentation issues for mobile applications.
- With the help of a prototype implementation for three platforms, we showcase the usefulness of our approach and evaluate its effectiveness.

[1] Android. See http://www.android.com
[2] S60 Platform. See http://www.s60.com
[3] Apple iPhone. See http://www.apple.com/iphone
[4] See http://www.opengroup.org/onlinepubs/009695399

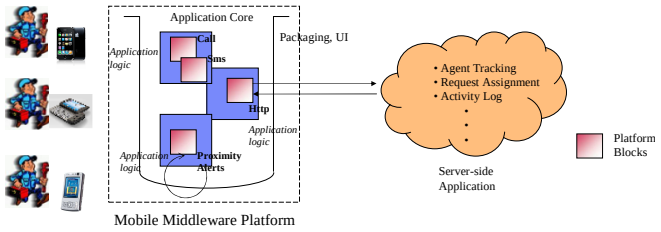

Fig. 1. Mobile Workforce Management Application

The rest of this paper is organized as follows. In Section 2, we articulate the fragmentation issues in mobile applications due to usage of platform interfaces. We then propose and describe MobiVine in Section 3, followed by its implementation in Section 4 for three different platforms. Section 5 presents an evaluation of MobiVine with respect to various software engineering and performance metrics, and Section 6 provides a discussion of related work. Finally, we conclude in Section 7 with some directions for future work.

2 Motivation

Consider a mobile workforce management solution that helps an enterprise manage its on-field agents, with respect to their tracking and task assignment. Figure 1 shows how such an application can be built as a combination of modules running on the middleware platform of agents' mobile devices and a server-side application. As the figure depicts, each device-side component has a core that is presented via a suitable user interface (UI) to the agent, and is also packaged as per the requirements (for deployment and execution) of the corresponding middleware. The core of the application can be further partitioned into consisting of several *platform* blocks bound together by *application logic* blocks. Platform blocks make use of platform interfaces and embed the associated functionality or data within the application logic blocks. For example, one can determine the proximity of an agent to a pre-specified region via the *addProximityAlert* interface. Similarly, to communicate with the server side application[5], one can use the *Http* interface. Further, additional functionality such as *sendSms* and *makeACall* enable quick communication of the agents with, say, the region supervisor.

From a developer's perspective, it is desirable to roll out the workforce management solution to multiple platforms, such as Android, Nokia S60, WebKit [15], etc. While the server-side component can be built using Web standards, porting of the device portion from one platform to another may involve a number of efforts on part of the developer. For example, S60 platform follows J2ME standard, iPhone requires programs to be developed in objective-C, while for WebKit, applications need to be in HTML and JavaScript. Similarly, the UI needs to be developed in accordance with the physical attributes of the device, such

[5] that does the tasks of book-keeping, request allocation, etc.

as screen size and resolution. Although fragmentation of mobile applications is a known and well-studied problem in general [3, 9, 10, 12, 13, 16], an interesting set of challenges appear because of the usage of platform interfaces within an application. As an example, let us examine the interface for adding location proximity alerts on Android and S60 platforms:

On Android, the exposed interface is

- *LocationManager.addProximityAlert (double latitude, double longitude, float radius, long expiration, Intent intent) throws SecurityException*

On S60, on the other hand, the corresponding interface is

- *LocationProvider.addProximityListener (ProximityListener listener, Coordinates coordinates, float proximityRadius) throws SecurityException, LocationException, IllegalArgumentException, NullPointerException*

Figures 2 (a) and (b) present code fragments corresponding to the invocation of *addProximityAlert* API in the workforce management application for Android and S60 platforms, respectively. As expected, there is a lot of variation in the code due to differences in *syntax* as well as *semantics* of this interface across the platforms. For example, on Android, registration of an alert leads to two sets of events - one for the device entering a proximity region, and the other associated with exiting. Further, multiple such events are generated, governed by the expiration period. These features are missing on S60, and additional code needs to be incorporated to support the same. This also leads to differences with respect to place(s) in the code where business logic needs to be included for handling received alerts. On the syntactic front, Android uses the *Intent* and *IntentReceiver* objects to realize a callback function for the alert mechanism. On S60, on the other hand, one needs to define an implementation for the abstract *ProximityListener* class. Similarly, there is diversity in terms of the name of the interface, as well as in the 'name', 'data type' and 'ordering' of various parameters. Finally, the interfaces differ in the set of exceptions thrown; the conditions under which these exceptions are thrown are determined by the underlying platform specification.

Two classical approaches for handling fragmentation of this form in the desktop world are (1) standardization of APIs, and (2) usage of programming techniques whereby the interface calls are wrapped, i.e. abstracted, in distinct modules which are then, automatically or otherwise, ported across the platforms. As discussed in Section 6, standardization efforts in the mobile domain [6] are currently piece-meal and fractured, hampered further by the desire of platform vendors to keep their products differentiated from others. Lack of standardization necessitates a defragmentation model based on programming techniques. While such an approach still applies conceptually, we argue that in a mobile setting it needs to cover the following:

(1) Each interface is bound by properties and attributes specific to the underlying middleware. For example, on Android, to obtain a handle of *LocationManager* (that offers the proximity alert interface), the developer has to input the 'application context' of the corresponding Android application. Similarly, on S60, a

[6] for instance those related to extending the mobile Java platform (J2ME)

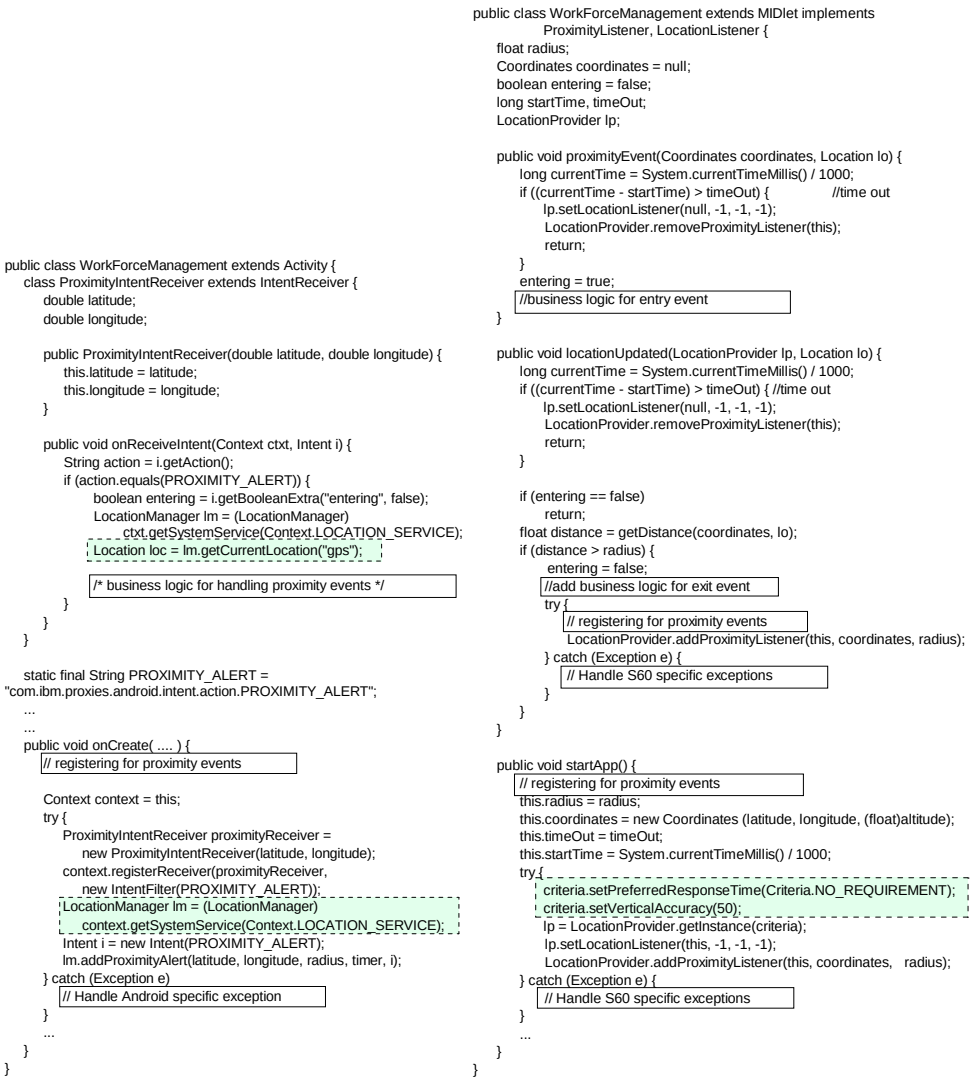

Fig. 2. Code fragment for invoking proximity alert on (a) Android, and (b) Nokia S60.

LocationProvider instance is returned by the middleware on the basis of constraints like accuracy desired, preferred response time, etc. - values for which are provided by the developer.

Across the wide range of platforms available, such attributes and properties are 'inherently different', i.e. cannot be abstracted out. Therefore, the defragmentation model should absorb heterogeneity in a dual manner - in addition to providing uniformity of semantics and syntax, it should offer enough flexibility to incorporate attributes and properties specific to a platform, preferably through

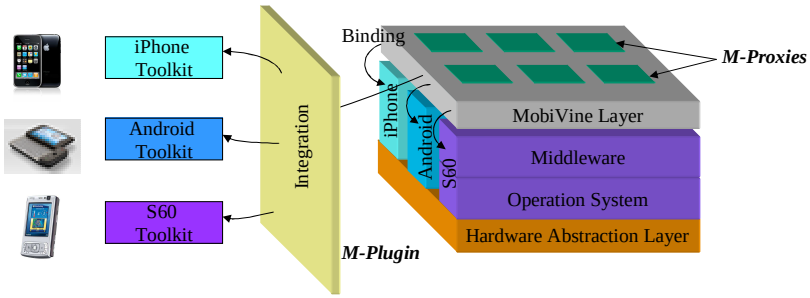

Fig. 3. MobiVine Overview

a mechanism *generic* across platforms.

(2) Application development for a platform, including packaging of various modules and their deployment, is tightly coupled to the underlying middleware. For example, on Android, the application extends an `Activity`, while on S60, it needs to extend the `MIDlet` class. Further, during deployment on S60, the entire application is packaged as a single jar file, that is qualified further with various permissions, Over-The-Air (OTA) deployment properties, profile configuration etc.

To foster creation of applications, mobile middleware platforms typically offer their own development environment or toolkit. Any de-fragmentation model should, therefore, be structured in a manner that enables it to be seamlessly integrated *within the existing toolkit*, thereby automatically supporting the desired programming, packaging and configuration style for an application.

(3) The mobile market scenario is highly dynamic, with new platforms and features being added with considerable rapidity. Thus, it becomes meaningful to have a de-fragmentation technique which is easily *extensible*, i.e. new mobile platforms, along with their underlying programming styles, specific attributes and properties, as well as feature updates are incorporated with ease.

Although various techniques have been proposed in the literature for abstracting heterogeneities in a mobile setting [4, 6, 7, 11], there is no model that incorporates the above characteristics. We propose a middleware solution based on such a model next, and thereafter describe its implementation and present its evaluation.

3 MobiVine Architecture

Figure 3 presents an overview of MobiVine, a middleware layer to handle fragmentation of platform interfaces for mobile applications. As shown in the figure, MobiVine consists of two main components. The first component, called 'M-Proxies', helps abstract the heterogeneity in interfaces across different platforms while binding to the underlying middleware stacks. The other component - 'M-Plugins' - integrates MobiVine with existing tooling & deployment infrastructure while utilizing the information kept in a structured format inside M-Proxies.

3.1 M-Proxy Model

Conceptually, the M(obiVine)-Proxy model is used to realize various platform blocks of a mobile application. The proxy model structure, as shown in Figure 4, consists of three planes. Whereas the first plane is directed towards removing fragmentation at the broad semantic level, the second plane provides syntax binding in the form of data types for parameters, callback handlers, return values, etc. across different programming languages. The third plane contains binding information for the underlying mobile platform middleware, including various platform specific properties and attributes. Intuitively, at each plane in our design, we capture a subset of the total information, and make it consistent in a manner so that it can be built upon by the subsequent plane(s) with further concrete information.

With the help of 'add proximity alert' interface introduced in the previous section, we now describe in more detail each of these planes[7].

Semantic Plane - In the first plane, called the *semantic* plane, we fix the structure of the interface, in terms of the method name, number, meaning and order of each parameter along with their dimensions, as well as the return value. For instance, the listing below defines a common interface semantics for adding proximity alerts in Android and S60.

```
<method>
  <name>addProximityAlert</name>
  <description> ... </description>
  <parameterList>
    <parameter>
      <name> latitude </name>
      <dimension> 1 </dimension>
      <description> ... </description>
    </parameter>
    <parameter> .. longitude .. </parameter>
    <parameter> .. altitude .. </parameter>
    <parameter> .. radius .. </parameter>
    <parameter> .. timer .. </parameter>
    <parameter>
      <name> proximityListener </name>
      <callback>
        <parameterList>
            <refLat, refLong, refAlt, currentLocn, entering>
        </parameterList>
      </callback>
    <parameter>
  </parameterList>
</method>
```

The listing depicts a common method name, which in practice can be chosen as the most accepted one across different platforms, or as per the discretion of the proxy creator. Uniformity and consistency of parameters is also maintained. For example, in contrast to code in Figure 2, now there is a common definition of callback parameter for receiving alert notifications.

Syntactic Plane - In the second plane, called the *syntactic* plane, we bind the interface structure with concrete data types required for different programming

[7] The fragments produced here are simplified versions of our actual implementation, based on an XML Schema defined by us. For clarity, we omit full Schema details.

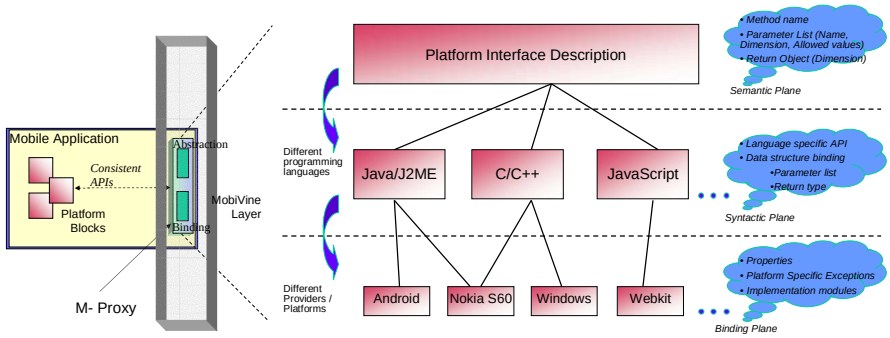

Fig. 4. M-Proxy Model

languages. Accordingly, multiple such information portions are provided, one for each language. In the listing shown below, we have attached data types corresponding to the Java language.

```
<method>
  <name>addProximityAlert</name>
  <qualifiedParameterList>
     <parameter> double </parameter>
     <parameter> double </parameter>
     <parameter> double </parameter>
     <parameter> float </parameter>
     <parameter> long </parameter>
     <parameter>
       com.ibm.telecom.proxy.ProximityListener
       <callback>
         <methodName>proximityEvent</methodName>
         <qualifiedParameterList>
           <double, double, double,
               com.ibm.telecom.proxy.Location, boolean>
         </qualifiedParameterList>
      </callback>
  </qualifiedParameterList>
</method>
```

It is important to note that types for a language need to be defined as per the structure of that language. For example, while in Java we have a callback 'object' that receives notifications, in JavaScript (or C) we can specify a function (or a function pointer) for this purpose. For one language, however, data structures will stay same across platforms. For instance, we have defined common 'ProximityListener' and 'Location' structures for both Android and S60 platforms, as depicted above.

Binding Plane - In the third and final plane, the *binding* plane, we provide implementation modules that realize this interface on different platforms. This is also the place where we include platform specific attributes (through the notion of a 'property list') as well as the underlying exception set.

```
<method>
  <name>addProximityAlert</name>
  <description> S60 implementation </description>
  <qualifiedClassName>
    com.ibm.S60.location.LocationProxy
```

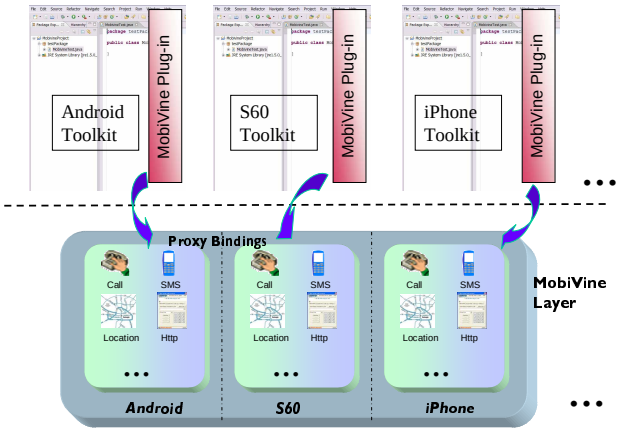

Fig. 5. MobiVine Plugins

```
  </qualifiedClassName>
  <exceptionList>
        javax.microedition.location.LocationException
          ...
  </exceptionList>
  <runtimeExceptionList>
        java.lang.SecurityException
          ...
  </runtimeExceptionList>
  <propertyList>
    <property>
      <name>preferredResponseTime</name>
      <description>Preferred max. response time </description>
      <dataType>int</dataType>
      <allowedValues> ... </allowedValues>
    </property>
       ...
  </propertyList>
</method>
```

In the listing above, binding of the interface for S60 platform is captured. It contains information about the class that implements the proxy, and the list of exceptions that are thrown on this platform. There are also various properties specific to S60, such as the preferred maximum response time required internally by the interface for polling of updates. Each of these properties can be further qualified by a default value and a set of allowed values on this platform.

3.2 M-Plugins

An important software engineering consideration from MobiVine's perspective is to make the proxies available in the existing toolkits, rather than propose a new development environment altogether. The gap between M-Proxies and an existing toolkit is bridged by a *M(obiVine) Plugin*. A plugin, by definition, is a package for extending a software, without explicitly requiring access or modifications to the underlying code base of the software. As shown in Figure 5, a plugin is developed for each platform, and supports the following four features:

(1) M-Proxy Visibility. A plugin reads and parses all three planes of a proxy and 'seamlessly' integrates the proxy in the toolkit set-up, i.e. makes it visible at all places where toolkit allows embedding of platform interfaces in the application.

(2) M-Proxy Presentation. Once a developer selects a proxy, M-Plugin presents a dialog box containing all parameters, properties, and corresponding description, possible and default values, etc. as described in the proxy structure.

(3) M-Proxy Configuration. M-Plugin enables configuration of a proxy by allowing user to make selections and provide values for variables and properties in the dialog box. It also generates code for invoking the configured proxy interface taking into consideration all user inputs, and offers preview of the generated code.

(4) M-Proxy Embedding. A plugin 'embeds' the implementation artifacts of a chosen and configured proxy within the application being created. This task can be dependent on the semantics of the platform. For example, the S60 platform requires the application to be bundled as a single J2ME MIDlet suite. In this case, the proxy binding jar(s) are merged with the application jar before deployment.

3.3 Other Features

Enterprise or third-party application developers can utilize the MobiVine architecture by creating proxies and plugins for their desired platforms and interfaces, and churning out rich, easy-to-port mobile applications. For them, apart from handling diversity of platform interfaces, the MobiVine layer acts as a gateway for enhancing the entire application development process:

Proxy Enrichment. A proxy can be enriched by adding extra functionality on top of the native one. For example, proxy for fetching location information can be made to offer output in various formats - radians, degrees, etc. Similarly, proxy for invoking 'Call' can provide the utility for coordinating the number of retries in case the callee is unreachable. Security and other policy modules can also be added to provide a layer of trust, authentication and access control.

Extension. The MobiVine architecture can be easily extended to absorb new platforms. In this case, if the semantic and syntactic planes already exist for other platforms, one requires to publish only the binding artifacts for proxies corresponding to a new platform. Moreover, as the proxy structure remains same across platforms, a *common* proxy interpretation routine can be used to develop plugins for different platforms.

In practice, proxies should be developed for an interface that exists on more than one platform, and not necessarily on 'all' platforms. This removes the requirement of the proxy set being determined by the least common denominator of functionalities across different platforms.

4 Implementation

In this section, we consider three mobile middleware platforms - Android, Nokia S60 and Android WebView, and describe an implementation of our proposed

architecture. Whereas the first two platforms enable creation of native mobile applications in the form of Android projects and J2ME MIDlets respectively, the last one is an Android implementation of the WebKit platform [15], and is used to render mobile Web applications developed in HTML and JavaScript. For each platform, we implemented proxies for various platform interfaces, and created plug-ins that embed these proxies in the corresponding middleware set-up.

4.1 M-Proxies

As outlined earlier, each proxy is a structured component, capturing information about the consistent, abstracted interfaces as per the three planes presented in Section 3. For this purpose, we designed five Schemas in the XML format - one for handling the semantic plane, one each for handling Java and JavaScript styles at the syntactic plane, and two at the implementation plane for binding Java (for S60 and Android), and JavaScript (for WebView). Fragments of XML documents corresponding to these Schemas were presented in Section 3. Due to space constraints, we do not detail the full Schema here.

On Android and Android WebView platforms, proxies were developed for four interfaces - Location, SMS, Call and Http interaction, using the Java packages provided in the Android SDK release m5-rc15. For Location proxy, we used the `android.location` package, `android.telephony.gsm` package for the SMS proxy and `org.apache.http` package for the Http proxy, while the phone call proxy was implemented using the `android.telephony.IPhone` class. On the other hand, for S60, proxies were created for three interfaces - Location, SMS and Http interaction. Call proxy could not be created in this case because the core functionality was not exposed on the S60 platform. All proxy modules were developed on top of Nokia S60 3rd Edition SDK for Symbian OS, supporting Feature Pack 2 for MIDP (Beta release). For Location, we made use of the `javax.microedition.location` package, while the SMS and Http proxies were implemented based on the `javax.wireless.messaging` and `javax.microedition.io.Connector` packages respectively.

For each proxy, creating implementation modules to expose abstracted interfaces required working on top of the native platform ones. While this can be done using the classical notion of wrapper code, in the case of M-Proxies, the following efforts require special mention:

Handling platform specific attributes as proxy properties. As discussed, any platform-mandated information should not form part of a common API, but should still be provided to the implementation module for that platform. In M-Proxies, this is enabled through a generic *setProperty()* method. On Android, for example, an instance of the `LocationManager` class is obtained by specifying the name of the system service, 'Location_Service' in this case, for an *application context*. Rather than passing application context as input parameter in the API, we provide it separately through the *setProperty()* method available with each proxy. Similarly, on S60, an instance of the `LocationProvider` class is obtained by passing criteria such as desired accuracy, preferred response time, etc. as

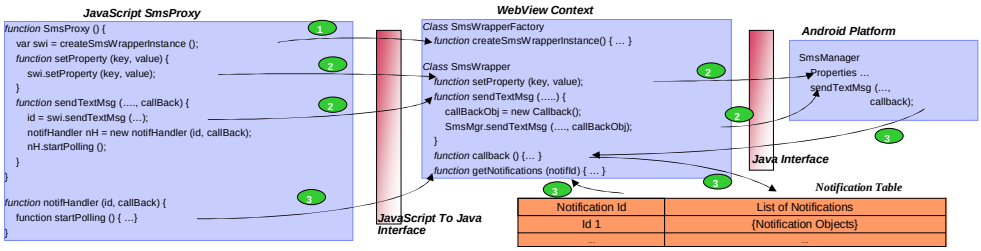

Fig. 6. JavaScript Proxy Implementation

parameters in the native interface. However, as with Android, such S60 specific criteria is not made a part of the common proxy API, albeit, included through notion of properties in the implementation.

Handling callbacks on Android. The native callback mechanism on Android is provisioned through the use of *Intent* and *IntentReceiver* classes that deal with detection of events and taking appropriate action, respectively. On the other hand, however, the proxy API for the proximity interface defines a *ProximityListener* object that requires a specific callback method - *proximityEvent* - to be called whenever proximity is detected. Hence, while implementing the `addProximityAlert` API on Android, we first create an 'Intent' for listening to 'Proximity_Alert' events, and attach it to the `LocationManager.addProximityAlert` method on the native Android platform. Thereafter, corresponding to this Intent, we define an 'IntentReceiver' and pass the *ProximityListener* object to it. Now, whenever a proximity alert event is detected by the Intent, it gets passed to the IntentReceiver, where the listener's *proximityEvent()* method is invoked. This way, the use of Intent and IntentReceiver is hidden from the application developer, requiring her to deal only with the *ProximityListener* object.

Creating Android WebView proxies. As mentioned before, Android WebView renders applications written in Web content language, such as HTML and JavaScript. To enable platform interfaces such as `sendTextMessage` and `getLocation` within an HTML content, Android offers a generic API *'addJavaScriptInterface()'* to add a Java object inside a WebView application, treat it as a JavaScript entity, and use the same for invoking a native platform interface. For example, with the help of this technique, a developer can provide an instance of `SMSManager` in a WebView window and through it send text messages by accessing the underlying `sendTextMessage` interface. Similarly, location information can be retrieved and embedded within the display content through a `LocationManager` object. We use this generic mechanism as the basis to implement JavaScript proxies for the WebView platform, following a three-step procedure. Through a schematic view of our implementation of `sendTextMessage` interface in the SMS proxy (Figure 6), we illustrate these three steps:

1. Enabling notion of a JavaScript proxy object - In the WebView context, we create 'Wrapper' Java classes that help connect a JavaScript proxy implementation

to the native Android platform. These classes are accessible in the JavaScript domain through the *addJavaScriptInterface()* call. In the case of SMS proxy, we define a `SmsWrapper` class, through which functionalities exposed by the SmsManager on the Android platform are accessed. Whenever a SMS proxy object is instantiated in the JavaScript domain, an instance of `SmsWrapper` class is generated by invoking the `SmsWrapperFactory` class, and its handle is kept within the JavaScript proxy instance (variable `swi` in Figure 6).

2. Exposing JavaScript proxy interfaces - JavaScript interfaces for the proxy were implemented by invoking the underlying Java interfaces through the handle of the `SmsWrapper` instance, i.e. `swi`. For example, to send a message, we use the `swi` handle to call the `sendTextMessage` functionality on the Android platform. Similarly, to set a property associated with the JavaScript proxy instance, we go through the `swi` handle to set the property of the `SmsManager` on the Android platform. Finally, exceptions thrown by the native interface invocation are propagated to the corresponding proxy with the help of error codes, wherein an error code is defined for each possible exception.

3. Providing support for callbacks[8] - Towards this, the `sendTextMessage` function in the JavaScript proxy object first passes all parameters, except the callback information, to the corresponding `sendTextMessage` interface defined in the `SmsWrapper` class. Here a new 'Callback object' is created that listens to notifications from the Android platform. All notifications thus received are stored within the WebView context using a `Notification Table`. The notifications in this table are retrieved periodically by the JavaScript proxy instance with the help of *startPolling()* function in its `notifHandler` object. To map notifications received from the Android platform to an underlying invocation in the JavaScript proxy domain, the `swi` handle makes use of the identifier returned as part of invocation of the `sendTextMessage` interface.

4.2 M-Plugins

The current development tools for Android (including WebView) and Nokia S60 platforms are built as extensions to Eclipse[9]. Eclipse is a popular open source meta application framework, that allows extensions to development platforms using the concept of *Eclipse plug-ins*. Therefore, we use the Eclipse framework to create MobiVine Plug-ins for these mobile platforms. Each MobiVine Plug-in is composed of three main components, viz *Proxy Drawer*, *Proxy Configuration View* and *Platform Specific Extensions*. All these components are next described in fuller details.

Proxy Drawer. The Proxy Drawer is a store of proxies, containing both the proxy representations and the associated implementation modules. Figure 7(a) shows the proxy drawer, listing all implemented proxies for the S60 platform.

[8] callback notifications received by an underlying Java object are not available to the invoking call in JavaScript.

[9] http://www.eclipse.org

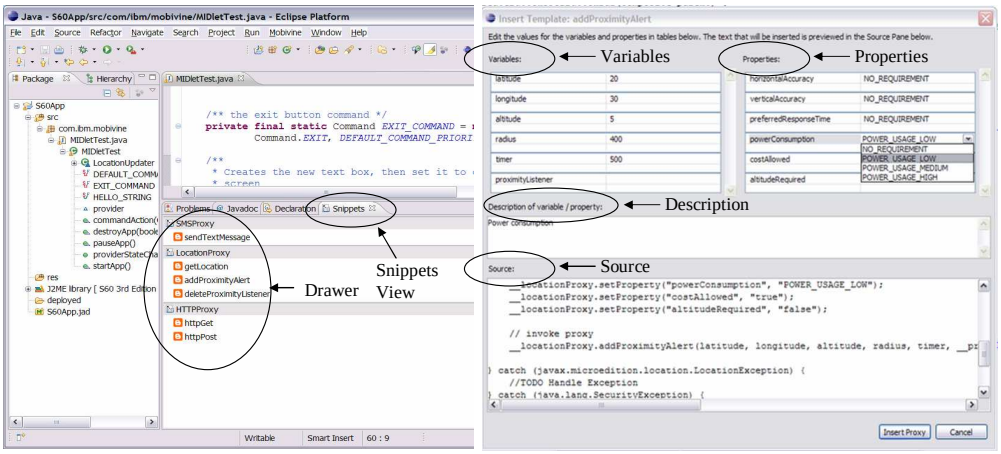

Fig. 7. (a) Proxy Drawer of MobiVine Plug-in. (b) Proxy Configuration Dialog

Proxies are organized in the drawer as *categories*, whereby each proxy is shown as a category with the APIs of the proxy presented as *items*. Through the drawer, any proxy API can be added to the code either by dragging and dropping the corresponding item to the desired location, or by double clicking the item to insert at the current cursor location. The MobiVine Proxy Drawer was implemented by extending the *Snippet Contributor* plug-in of Eclipse, and offers a *Snippets view* with drawers of items that can be dragged and dropped in a text or visual editor. Contents of the drawer, i.e. proxies and APIs in the form of categories and items respectively, are specified in `plugin.xml` file of the plug-in.

Proxy Configuration and Code Generation. The Snippet Contributor plug-in allows specification of a handler class that generates the content to be embedded on drag-and-drop action. The content itself can be unformatted text, formatted text such as an XML or java code, or binary data based on needs of the target editor. Configuration dialog to accept user inputs using M-Plugin's handler class is shown in Figure 7(b), for `addProximityAlert` interface on the S60 platform. While parameters of the common proxy interface are presented under the `Variables` column, S60 platform specific Properties are presented under the `Properties` column. Associated default value, allowed values and description is also provided for each parameter and property. For instance, from the snapshot we see the allowed values for the `powerConsumption` attribute. Preview of generated code considering all user inputs is available in the `Source` view.

Platform Specific Extensions. As mentioned earlier, the task of integrating proxy binding modules with rest of the application can be dependent on the semantics of the platform. MobiVine plug-in provides such platform specific extensions to ease integration. For S60 and Android, these extensions deal with absorbing the proxy implementation jars in the resource structure - including classpath - of the corresponding projects. Further, for S60, functionality is also provided to merge jars of all chosen proxies with the application jar before

deployment, since the platform requires the application to be bundled as a single J2ME MIDlet jar. Finally, for the Android WebView platform, extensions are provided for incorporating JavaScript proxy implementations within a WebView project, as well as for injecting the associated Java 'Wrapper' objects through the *addJavaScriptInterface()* calls, as discussed previously.

5 Evaluation

In the previous sections, we talked about the M-Proxy model and how it can be seamlessly integrated into mobile application development environments. It is also important, however, to evaluate the effectiveness of our approach based on some software engineering principles and performance metrics. Towards this, we consider the following questions:

(1) Is the code easily *portable* across multiple mobile platforms using proxies?

(2) Is it *less complex* to develop and debug mobile applications that include platform functionalities through proxies?

(3) Is it easier to *maintain* code and migrate to new versions of the platforms?

(4) Is the application *performance* affected by the usage of proxies?

```
public class WorkForceManagement extends Activity
        implements ProximityListener {
    ...
    public void onCreate( ... ) {
        // registering for proximity events
        try {
            LocationProxyImpl loc = new LocationProxyImpl();
            loc.setProperty("context", this);
            loc.setProperty("provider", "gps");
            loc.addProximityAlert(latitude, longitude, altitude, radius,
                timer, this);
        } catch (Exception e) {
            // Handle Android specific exceptions
        }
        ...
    }

    public void proximityEvent(double refLatitude, double refLongitude,
        double refAltitude, Location currentLocation, boolean entering) {
        /* business logic for handling proximity events */
        ...
    }
}
```

(a)

```
public class WorkForceManagement extends MIDlet
        implements ProximityListener{
    ...
    public void startApp(...){
        //registering for proximity events
        try {
          LocationProxyImpl loc = new LocationProxyImpl();
          loc.setProperty(..);
          loc.addProximityAlert(latitude, longitude, altitude, radius,
                timer, this);
        } catch (Exception e) {
            // Handle S60 specific exception
        }
          ...
    }

    public void proximityEvent(double refLatitude, double refLongitude,
        double refAltitude, Location currentLocation, boolean entering) {
        /*business logic for handling proximity events*/
        ...
    }
    ...
}
```

(b)

Fig. 8. Code fragment for proximity alert using proxies on (a) Android & (b) Nokia S60.

We try to answer the above questions with the help of code fragments from the work force management application described earlier in Section 2. These fragments, captured in Figures 8 and 9, correspond to the invocation of *addProximityAlert* API in the application through the proxy model.

Portability: Compared to code depicted in Figure 2, code using the proxy model (c.f. Figures 8 and 9) is mostly similar as far as the interface usage

```
<script type="text/javascript" >
    function JSInit(...) {
        try {
            // registering for proximity events
            var loc = new LocationProxyImpl();
            loc.setProperty("provider", "gps");
            loc.addProximityAlert(latitude, longitude, altitude, radius, timer, proximityEvent);
        } catch (ex) {
            // Handle Android specific exceptions
        }
        ...
    }

    function proximityEvent(refLatitude, refLongitude, refAltitude, currentLocation, entering) {
        /* business logic for handling proximity events */
        ...
    }
    ...
</script>
```

Fig. 9. Code fragment for proximity alert using JavaScript proxy on Android WebView

is concerned, and leads to portability not only across different platforms, but also across languages. Here, our M-Proxies absorb the API differences inside the implementation modules, and present a uniform interface to a developer. Further, platform specific attributes are absorbed through a consistent *setProperty()* method. Finally, as can be seen from the figures, now the code around the API is also similar as both the inputs as well as the output attached to the API are same. For example, the `currentLocation` object in *proximityEvent()* is of the same type on both Android and S60 platforms, whereas without proxies, it refers to platform specific objects that imbibe varying semantics and syntax.

Complexity: As shown by various fragments, the code on both Android and S60 using the proxy model (Figure 8) is much simpler and smaller compared to the one without the proxy model (Figure 2). This is because the proxies have hidden the complexity of using platform specific APIs from the developers, such as the one related to the use of `Intent` and `IntentReceiver` classes for handling callbacks on Android. Similarly, business logic for handling proximity alerts on S60 is now concentrated at one place, instead of being scattered in the code presented in Figure 2. Moreover, well-developed proxy modules can make the task of testing and debugging an application a lot easier. For instance, proxy bindings can be designed to efficiently handle exceptions on different platforms.

Maintenance: In the case of proxy model, any changes in the application code around platform interfaces, such as addition of new features or business logic, can be applied at the same place on different platforms since the code is similar. This makes it easier to maintain an application *'across the platforms'*. Looking from a different perspective, as mobile platforms evolve, new version of a platform may have different APIs as compared to the previous versions. For example, the new release 1.0 [10] of Android platform takes a `PendingIntent` object in *addProximityAlert* API, instead of an `Intent` object. In such scenarios, the application code would normally need changes to take care of these differences.

[10] See, http://code.google.com/android/documentation.html

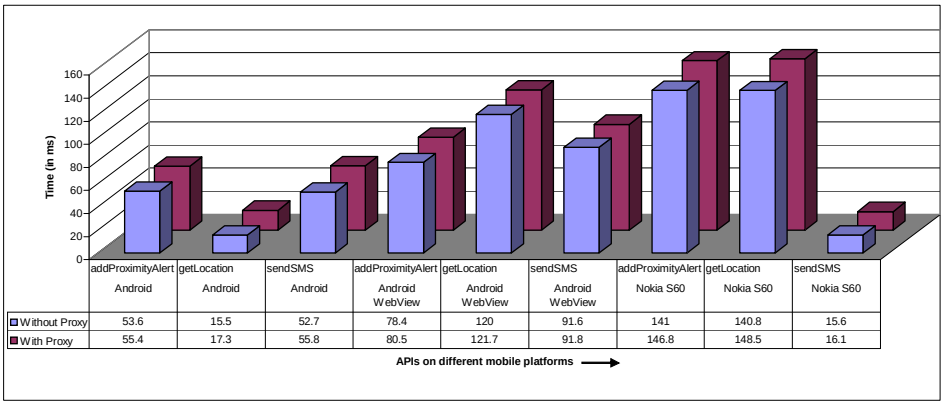

Fig. 10. Time taken for invoking APIs on Android, Android WebView and Nokia S60

However, using our approach, the differences can be absorbed inside proxies for this version of the platform, thereby requiring no changes in the application. This makes the application easier to maintain *'as platforms evolve'*.

Performance: MobiVine layer is primarily a design-and-development-time artifact, helping a developer embed M-Proxies in her application. The run-time overhead of MobiVine with respect to an application is therefore restricted to the cost of invoking platform interfaces through the M-Proxy model. We conducted some experiments to measure this overhead. Towards this, Figure 10 shows the time taken for invoking various APIs with and without proxies on different platforms, where for each API we took an average of ten executions.

It can be seen from the figure that the overhead of proxy is a small fraction of the corresponding native interface. This is because the additional code inside a proxy (on top of invoking the underlying interface) is limited to a few extra calls dealing with data-type conversions, platform specific attributes and other small de-fragmentation logic. Considering the time a typical application spends in UI interaction, business logic, etc., we conclude that the cost of using a proxy model in most cases is negligible as compared to the total application runtime.

6 Discussion and Related Work

Mobile devices are becoming increasingly sophisticated, providing enhanced performance, larger memory size as well as a number of new features such as GPRS, Music Player, Camera, etc. Due to the pervasive nature of these devices, innovative applications designed to run on them have the potential of changing business processes and human-lifestyle in a number of profound ways. While mobile phones and their underlying software have traditionally been closed environments, changes are being brought about with the advancement of open

platforms such as Brew[11], Symbian UIQ[12] and Android. Software development approaches specifically for mobile applications have been proposed; for example [1] presents an agile development approach for mobile platforms. Numerous efforts are also being undertaken in the realm of porting a desktop application to mobile platforms. For instance, the Sun Java Toolkit for Connected Device Configuration (CDC)[5] facilitates porting Java SE programs to mobile devices. Similarly, [2] examines the ability of common desktop applications to gracefully handle error conditions when ported to more unreliable mobile devices.

Fragmentation is the inability to "write once and run anywhere" [10]. Fragmentation in mobile applications is caused by several types of diversity: (1) **Hardware diversity**, such as differences in screen parameters (size, color depth, orientation, aspect ratio), memory size, processing power, input mode (keyboard, touch screen, etc.), additional hardware (camera, voice recorder), and connectivity options (Bluetooth, IR, GPRS, etc.). (2) **Software Platform diversity** in mobile devices caused by factors such as differences in platform/OS (Symbian, Nokia OS, RIM OS, Android, BREW, etc.), API standards (MIDP 1.0, MIDP 2.0, etc.), optional/proprietary APIs, variations in access to hardware (e.g., full screen support), maximum binary size allowed, etc. (3) **Environmental diversity**, such as that in the deployment infrastructure (branding by carrier, compatibility requirements of a carriers back-end APIs, etc.). Note that desktops do not have hardware, software or environmental diversity of such scale; fragmentation in desktop applications is thus much lower compared to mobile applications.

Standardization efforts have been undertaken to overcome mobile platform diversity. For example, extensions to the Java platform have been proposed through Java Specification Requests (JSRs) to support APIs for new features in mobile devices. One such effort - JSR 248[13] - is designed to provide a comprehensive set of APIs for Mobile Service Architecture. Ideally, applications written for a device based on standards such as Java ME as the software platform should work on all devices supporting Java ME. However, in practice, different devices support basic Java ME along with a select set of JSRs (decided independently by each mobile OS/Platform developer), resulting in diversity even among these devices. The Open Mobile Terminal Platform (OMTP) forum has launched a project, called BONDI[14], to drive a standardized approach for letting web applications access key local capabilities on the mobile device, while provisioning a user controlled security layer. Currently, most of these standardization efforts are fractured themselves, with each effort being promoted by a set of vendors. The success of this approach, therefore, remains to be seen in the long run.

A number of software solutions have been suggested in both the mobile as well as the desktop world for dealing with fragmentation. This includes deriving different versions of the application suitable for different platforms from a single code base using techniques like meta-programming and automatic generation. A

[11] http://brew.qualcomm.com/brew
[12] http://www.uiq.com
[13] http://jcp.org/en/jsr/detail?id=248
[14] http://bondi.omtp.org/default.aspx

tool that supports this approach is the NetBeans Mobility Pack [9]. On the other hand, [3] describes an automatic generator to develop UIs to fit different screen sizes. One can also create applications by utilizing an abstraction library that hides diversity among different APIs. OpenKODE [12] is a library containing APIs for media and gaming applications that sits over Mobile OS and abstracts resources to minimize changes when porting applications between Linux, Symbian and Windows based platforms. Similarly, TWUIK [13] is a UI library for mobile applications that enables a single UI to adapt to multiple devices. MobiVine layer based on the structured M-Proxy model can be seen as a refinement of these approaches by including platform specific attributes and enabling integration with the application development offering of a middleware platform.

As mobile phones experience enhancements in functionality, features and capability, there will be a desire to host services on the device itself that are accessible by other mobile phone users, or more generally, by Web users. For example, one could visualize a query to a cell phone regarding its real-time information like current location and off-hook, on-hook status. An initial effort in this direction is in [14] which talks about realizing Web service applications in a smart space over devices such as mobile phones, PDAs, etc. Similarly, Celadon [8] supports applications, users, and devices in mobile commerce spaces, while providing a runtime and a tooling environment. Our M-Proxy model can be utilized to design such spaces more efficiently by creating proxies for various features that enable a mobile device to interact in these environments.

7 Conclusions and Future Work

To foster creation of rich mobile applications, popular platform vendors such as Android, iPhone and Nokia S60 offer 'interfaces' to a) access information on the mobile device (for e.g. user location), and b) invoke device capabilities (like camera), from within the applications. In this paper, we presented MobiVine, a middleware layer to handle fragmentation of such platform interfaces for mobile applications. We described MobiVine as consisting of two main components. The first component, called 'M-Proxies', helps abstract heterogeneities in interfaces across different platforms while binding to the underlying middleware stack. The other component, called 'M-Plugins', helps integrate MobiVine with existing tooling and deployment infrastructure. With the help of a prototype implementation for three platforms - Android, S60 and Android WebView, we evaluated the effectiveness of our approach based on various software engineering principles and performance metrics.

In the future, we would like to extend MobiVine implementation to cover other platform interfaces like those related to calendaring and contact list information, as well as include other platforms such as iPhone. We also plan to make the concept of proxy model broader by studying its applicability to other forms of mobile heterogeneity, for instance screen size and resolution. Finally, we wish to explore the usefulness of proxies in a converged network domain. For example, it would be interesting to create proxies for enabling voice access to Web applications.

Similarly, proxies can be created to interact with various Web-offerings based on the REST architecture [8].

References

1. P. Abrahamsson, A. Hanhineva, H. Hulkko, T. Ihme, J. Jlinoja, M. Korkala, J. Koskela, P. Kyllnen, and O. Salo. Mobile-D: An Agile Approach for Mobile Application Development. In *Proceedings of OOPSLA 04*, Vancouver, Canada, October 2004.
2. M. Bigrigg and J. Slember. Testing the Portability of Desktop Applications to a Networked Embedded System. In *Proceedings of the Workshop on Reliable Embedded Systems*, 2001.
3. K. Gajos, D. Christianson, R. Hoffmann, T. Shaked, K. Henning, J. J. Long, and D. Weld. Fast and Robust Interface Generation for Ubiquitous Applications. In *Proceedings of the Seventh International Conference on Ubiquitous Computing (UBICOMP'05)*, 2005.
4. P. Grace, G. S. Blair, and S. Samuel. A Reflective Framework for Discovery and Interaction in Heterogeneous Mobile Environments. *SIGMOBILE Mobile Computing and Communications Review*, 9(1):2–14, 2005.
5. Sun Java Toolkit for CDC. http://java.sun.com/products/cdctoolkit/.
6. J. Kreku, M. Hoppari, T. Kestila, Y. Qu, J. Soininen, and K. Tiensyrja. Application - Platform Performance Modeling and Evaluation. In *Proceedings of the Forum on Specification, Verification and Design Languages*, 2008.
7. L. Marucci and F. Paternò. Supporting Adaptivity to Heterogeneous Platforms through User Models. In *Proceedings of the 4th International Symposium on Mobile Human-Computer Interaction*, pages 409–413, London, UK, 2002. Springer-Verlag.
8. S. McFaddin, D. Coffman, J. Han, H. Jang, J. Kim, J. Lee, Y. Moon, C. Narayanaswami, Y. Paik, J. Park, and D. Soroker. Modeling and Managing Mobile Commerce Spaces using RESTful Data Services. In *Proceedings of the Ninth International Conference on Mobile Data Management*, 2008.
9. Resolving Java ME Device Fragmentation Issues Using NetBeans Mobility. http://www.netbeans.org/kb/60/mobility/javame-devicefragmentation.html.
10. D. Rajapakse. Techniques for De-fragmenting Mobile Applications: A Taxonomy. In *Proceedings of 20th Intl. Conf. on Software Engineering and Knowledge Engineering Conference (SEKE'08)*, July 2008.
11. H. Safa, H. Artail, and R. Shibli. An Abstract Model for Supporting Interoperability in Mobile Ad-hoc Networks. In *Proceedings of the IEEE International Conference onWireless and Mobile Computing, Networking and Communications*, 2006.
12. N. Trevett. OpenKODE: An Open Standard for Mobile Application Portability. http://www.khronos.org/files/openkode_whitepaper.pdf.
13. TWUIK: Powerful UI Technology for your JavaME Applications. http://www.tricastmedia.com/twuik/.
14. J. van Gurp, C. Prehofer, and C. di Flora. Experiences with Realizing Smart Space Web Service Applications. In *Proceedings of Consumer Communications and Networking Conference, (CCNC'08)*, January 2008.
15. The WebKit Open Source Project. http://webkit.org/.
16. WebSphere Everyplace Mobile Portal. http://www01.ibm.com/software/pervasive/ws_everyplace_mobile_portal_enable/.