A Framework for the Declarative Implementation of Native Mobile Applications

Patricia Miravet*, Ignacio Marin*, Francisco Ortin†, Javier Rodriguez*

June 28, 2013

Abstract

The development of connected mobile applications for a broad audience is a complex task due to the existing device diversity. In order to soothe this situation, device-independent approaches are aimed at implementing platform-independent applications, hiding the differences among the diverse families and models of mobile devices. Most of the existing approaches are based on the imperative definition of applications, which are either compiled to a native application, or executed in a Web browser. The client and server sides of applications are implemented separately, using different mechanisms for data synchronization. In this paper we propose DIMAG, a framework for defining native device-independent client-server applications based on the declarative specification of application workflow, state and data synchronization, user interface, and data queries. We have designed DIMAG considering the dynamic addition of new types of devices, and facilitating the generation of applications for new target platforms. DIMAG has been implemented taking advantage of existing standards.

Keywords: Device fragmentation, mobile applications, data and state synchronization, dynamic code generation, client-server applications

1 Introduction

The development of connected mobile applications is a promising field for companies and researchers. Around 6 billion cellular connection subscriptions (or 86 per 100 inhabitants) have been reached by the end of 2012 [1].

*Research and Development Department, CTIC Foundation, C/ Ada Byron 39, 33203, Gijon, Spain, {patricia.miravet, ignacio.marin, javier.rodriguez}@fundacionctic.org

†Computer Science Department, University of Oviedo, Calvo Sotelo s/n, 33007, Oviedo, Spain, ortin@lsi.uniovi.es

This means a potential market that exceeds that of the traditional market of fixed lines users (about 1,270 million subscriptions, by the end of 2012 [1]), so more and more software companies are looking for their slice of that global pie.

The heterogeneity of device models and versions, features, operating systems, and other elements of the software stack, makes it difficult to create applications for all the mobile users. This problem is known as *device fragmentation* [2] (or device diversity). A common approach to face that problem is to create from scratch different versions of the same software for different device models. This approach usually leads to different user experience for the same application in different devices, but it is also a source of problems in terms of software maintenance. The companies that develop software for mobile devices try to cover as many devices as possible. Thus, it is advisable to build platforms that allow developers to create applications once, and execute them in every mobile device.

A recent press release from IDC (*International Data Corporation*) indicating the market share of mobile operating systems for smartphones in the fourth quarter of 2012 [3] shows that Android is the most widely spread mobile operating system (70.1%), followed by Apple iOS (21%), BlackBerry RIM (3.2%), Windows Phone / Mobile (2.6%), Linux (1.7%) and others (1.3%). In order to cover this ample variety of operating systems, some software technologies allow developers to create applications once, and generate them for several software platforms. Titanium, Corona, Rhomobile and PhoneGap are examples of systems to create platform-independent mobile applications. Some facilitate the separation between presentation and behaviour based on Web development languages (e.g., HTML, CSS and JavaScript are used in PhoneGap and Titanium). However, all of them are based on imperative approaches for client-side application definition. To the knowledge of the authors, no approach seems to exist in the market that provides a declarative application definition to generate native client-server applications.

The main contribution of this paper is DIMAG, a Device-Independent Mobile Application Generation framework that provides a software tool for the dynamic generation of native client-server applications. Applications are defined by means of simple declarative specifications. DIMAG has been designed to facilitate the addition of new mobile platforms, supporting the reutilization of the components provided for this purpose. Besides, DIMAG follows the *Convention over Configuration* principle [4] to allow the dynamic addition of new target platforms using reflection. Currently, DIMAG generates client applications for Android, Java ME and Windows Mobile.

The rest of the article is organized as follows. Section 2 defines the different elements that comprise a DIMAG application. An example application,

the architecture and all the elements of the DIMAG framework are described in Section 3. Section 4 presents its design and implementation. The related work is described in Section 5, and Section 6 includes a qualitative evaluation. Conclusions and future work are presented in Section 7.

2 Specification of DIMAG Applications

DIMAG uses a declarative approach to define applications. This way, developers have to express *what* the application is meant to do, instead of expressing *how* to do it. The latter is the typical approach in the traditional programming languages used for building mobile applications (see Section 5).

We have considered a declarative approach because we think it has many benefits. First, it provides a higher abstraction level with sufficient expressive power to generate code for different target platforms. Second, software maintainability is improved because there is only one single implementation of each mobile application. We also think that a declarative approach facilitates the translation of applications to different languages and platforms, since different imperative strategies can be followed depending on the target platform (see Section 4.3). Finally, it may allow people without previous experience in imperative programming languages to create applications.

Applications in the DIMAG framework are conceived as distributed client-server programs. Therefore, two different parts of an application are generated: the server part, to be run in an application server; and the client part, to be downloaded and run in the mobile client. The server side is compiled for the Java EE platform, but the client side is generated and compiled for different mobile operating systems and software environments –depending on the actual device that demands the application.

DIMAG applications are divided into three modules:

- Workflow: An application is modelled as a state machine, made up of a set of states (one of them considered as the initial one) and transitions between states which take place when a given event happens.
- User interface: When the application requires displaying information to (or receive data from) the user, a view is associated to a specific state of the application.
- General information: This includes the application identification, its description, definition of data sources, synchronization policies between the client and the server, and the external resources required (e.g., images, audio and video).

3 The DIMAG Framework

3.1 A Motivating Example

The following example shows an example use of the framework at its current stage. The example simulates a simple online shop in which users can search products through different categories, and store the products in a shopping cart until the final completion of the purchase. Figure 1 shows the navigation flow, including the different synchronization points. In the client-server architecture followed (Section 3.2), the client implements a simple persistence layer that reflects part of the persistence system in the server side. Changes occur periodically in both sides of the application, and the synchronization policy defines when these changes are propagated between both sides. The data synchronization policy keeps the data layer up-to-date in both sides transparently, while state synchronization maintains the server side informed about the events triggered in the client side.

Data synchronization happens when the application data layer is accessed. In the example, it occurs when the user is authenticated and when the lists of categories and products are shown. Data synchronization also takes place when the user adds an item to the shopping cart, and when he/she decides to check out the selected items. It is worth noting that this kind of synchronization does not always imply real synchronization against the server. For example, when a user adds an item to the shopping cart, only the local data layer is modified. However, for simplicity (as in most cases both actions are directly related) and for legibility purposes (see Section 3.4), every data layer access in DIMAG is encapsulated with a data synchronization element.

State synchronization takes place when the user selects a specific category or product (Figure 1). Then, the selected item is transmitted to the server for statistical purposes, considering the categories and products commonly selected by the users.

3.2 Architecture

Figure 2 shows the architecture of the DIMAG framework. The left part of the figure shows the server side of DIMAG applications, whereas the right part presents the client side. Once the application definition is created, the server side is deployed in the application server and the client side is made available for remote users to download in their mobile devices.

The server side runs an instance of each application deployed in DIMAG, and implements the following four modules (Figure 2):

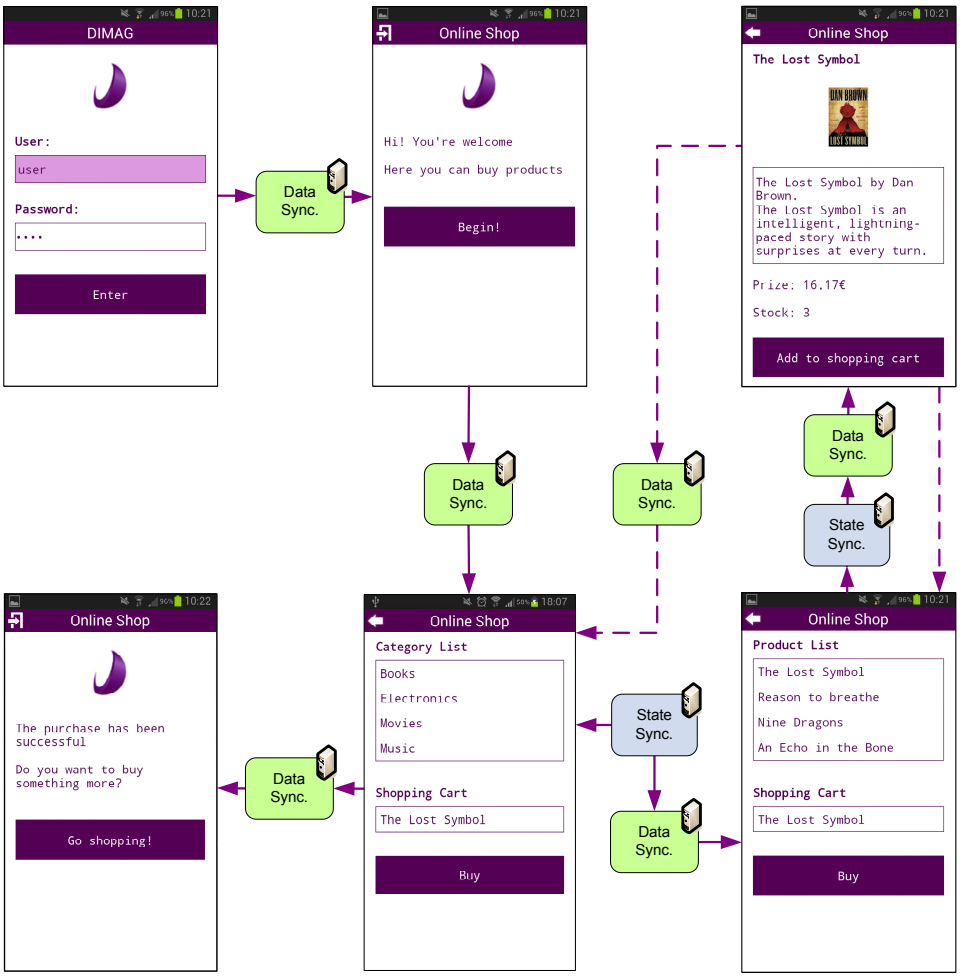

Figure 1: Visual description of an example DIMAG application.

– The communication layer, which listens to HTTP requests from clients. There are two types of requests: those initiated by the users when their mobile Web browser downloads a mobile application, and those HTTP requests performed by the mobile application logic. The DIMAG framework uses Servlets to provide client applications; for Web services, SOAP [5] and WS-I Basic Profile [6] are used to ensure Web services interoperability among different implementations.

– The device detection and application provision module receives information from HTTP requests when users download an application. Evidences are extracted from those requests in order to identify the client software platform, using a Device Description Repository (DDR). After identification, this module looks for the appropriate version of the application in its application repository, and sends it to the client de-

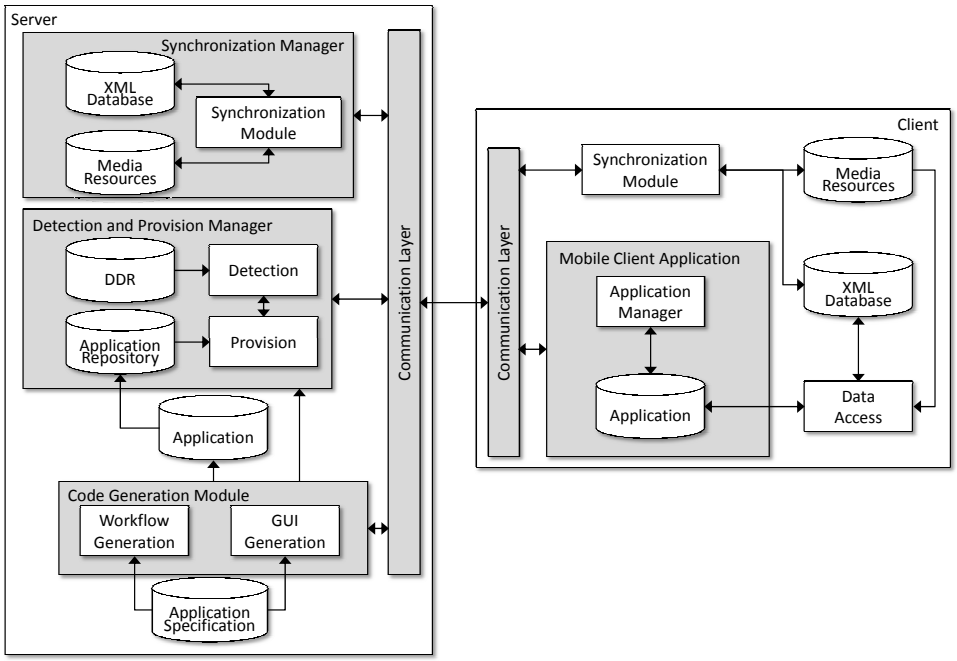

Figure 2: The DIMAG architecture.

vice for installation. If the repository does not have the appropriate version of the application, a new version is generated by the code generation module. The standard W3C DDR Simple API [7] has been used to design this module, choosing the DDR Simple API Reference implementation within the Morfeo Project [8].

– The code generation module dynamically generates the specific-platform client application selected by the user, as commanded by the device detection mechanism. If the DIMAG framework does not support code generation for a specific device, it will log information about the identification evidences. This is done for maintainability purposes, so that the administrator can identify the necessity to generate applications for unsupported platforms. This way, companies exploiting the DIMAG framework are reported about unsupported devices or device families willing to use specific mobile applications. The framework has been designed to facilitate the addition of new software platforms at runtime (Section 4.3).

– The synchronization module is aimed at synchronizing information between the server and the client. This includes the synchronization of application data and workflow state between both sides of applications. The main goal is to obtain a simple declarative system that enables tuning the synchronization level desired for each application.

For the sake of simplicity, data sources to be synchronized are simple files (e.g., XML documents and images). In the server side, there is a connector between these files and the relational database that transparently manages application persistence. When information is updated in any of the tables of the database, data files are accordingly updated. Similarly, changes in client data files also trigger updates of the relational database. DIMAG uses the SyncML [9] protocol and the Funambol open-source synchronization server [10] for client-server file synchronization.

The client side includes a communication layer to perform HTTP requests to the server (i.e., to invoke the remote methods of the server Web services), a synchronization module corresponding to that in the server, and the client-side of the application itself. The client application, in turn, consists of a manager responsible for managing the application lifecycle, and the specific application dynamically generated by the server. The implementation of the client side varies from one mobile platform to another. For example, a C# Windows Mobile application makes DIMAG generate code using the Windows Mobile API from Funambol, in order to synchronize information between both sides; for SOAP interchange, a SOAP client stub is generated from the WSDL Web service definition by means of the .NET Framework SDK tools; and `System.Net.HttpWebRequest` and `System.Net.HttpWebResponse` .NET Compact Framework classes are needed for more specific REST HTTP messages. On the other hand, for Java ME MIDlet-based applications, the DIMAG code generator module uses the Java ME API from Funambol, the JSR-172 [11] Web Services implementation, and the `javax.microedition.io.HttpConnection` class.

When deployment takes place, an instance of the server side is run and a new download URI is available in the Web server. Then, when users access that URI from their mobile Web browsers, a new version of the application client side is dynamically generated for their specific device.

3.3 Declarative Definition of Applications

The DIMAG framework is based on a set of declarative languages which are combined in order to define connected mobile applications. First, DIMAG-Root is an XML language to define the general aspects of a DIMAG application. It includes references to two external documents: user interface and workflow definition. The former is expressed with the DIMAG-UI language, a simplification of MyMobileWeb IDEAL [12]. This language separates the UI content from its style, referring to external CSS files. Workflow definition is expressed in SCXML. However, we propose several extensions to

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application SYSTEM "application.dtd">
<application xmlns='http://dimag.org/namespace/application'>
  <desc><appversion>1.8.0</appversion></desc>
  <flow><flowdir path="/dimag/resources/flow"/></flow>
  <ui>
    <xmldir path="/dimag/resources/xml"/>
    <cssdir path="/dimag/resources/css"/>
  </ui>
  <syncpolicy>
    <syncdataserver><URI>http://156.136.2.19:7777</URI></syncdataserver>
  </syncpolicy>
  <datamodel>
    <syncdir path="/dimag/resources/data"/>
    <entities>
      <entity id="Products" filename="products.data" defaultSync="preaccess"/>
      <entity id="Categories" filename="categories.data" defaultSync="both"/>
      <entity id="Users" filename="users.data" defaultSync="both"/>
      <entity id="ShoppingCart" filename="cart.data" defaultSync="postaccess"/>
    </entities>
  </datamodel>
  <server><URI>http://156.136.2.19:4567</URI></server>
  <resources>
    <lib path="/dimag/resources/externallib"/>
    <media path="/dimag/resources/media"/>
  </resources>
  <distribution><generatedcode path="/dimag/generatedcode"/></distribution>
</application>
```

Figure 3: Example DIMAG-Root document.

SCXML to provide all the elements required in the DIMAG workflow descriptions (Section 3.3.2). Synchronization information (data sources and data synchronization policies) is currently placed in the DIMAG-Root and the SCXML languages.

3.3.1 DIMAG-Root Language

The DIMAG-Root document that corresponds to the example described in Section 3 is shown in Figure 3. A DIMAG-Root document includes the following XML elements:

- `<desc>` describes the application version by means of its `<appversion>` child element. In DIMAG, different versions of client-server applications can be executed concurrently.
- `<flow>` holds the URI to an SCXML document describing the application workflow.
- `<ui>` provides a path to the DIMAG-UI documents defining the views of the application. There is one DIMAG-UI document for each screen

to be displayed by the client side of the application. The name of each document is built by appending the value of the `id` attribute of each view state in the SCXML document to the XML filename extension.

- `<syncpolicy>` allows the definition of data and state synchronization policies between the server and the client (`<datamodel>` and `<syncpolicy>` will be described in detail in the next section).
- `<datamodel>` defines the data sources to be used by the application. As mentioned, data sources are implemented by means of XML documents transparently synchronized with the server database (Section 3.4)
- `<server>` indicates the server endpoint used for data and state synchronization, and remote method invocation.
- `<resources>` provides URIs to the external resources used by the application, such as external software libraries (`<lib>`) or media files (`<media>`).
- `<distribution>` indicates information about how to distribute the application. Currently, it provides the path of the generated installers for the different mobile platforms.

3.3.2 SCXML

Workflow definition is expressed in the SCXML language created by the W3C Voice Browser working group. We have implemented the SCXML processor by reusing the Apache Commons SCXML open-source project [13].

The state machine in Figure 4 models the dynamic behaviour of the example application in Figure 1, and its corresponding SCXML document is shown in Figure 5. SCXML allows the definition of states by means of the `<state>` element, setting one of them as the initial one. It also allows defining the transitions between states (`<transition>`), and the actions to be performed when entering (`<onentry>`) or leaving (`<onexit>`) a state.

The explanation of some elements in the workflow related to synchronization (`<dimag:syncdata>` and `<dimag:syncstate>`) and data management (`<dimag:query>`) are omitted here because they are treated in depth in the next section. These XML elements and attributes have been created within the `dimag` namespace, allowing the SCXML language processor to differentiate between the SCXML lexical elements (to be handled by itself) and our language extensions. These extensions are handled by the additional modules we have specifically created for the DIMAG framework, which decorate the data structure representing the workflow state machine.

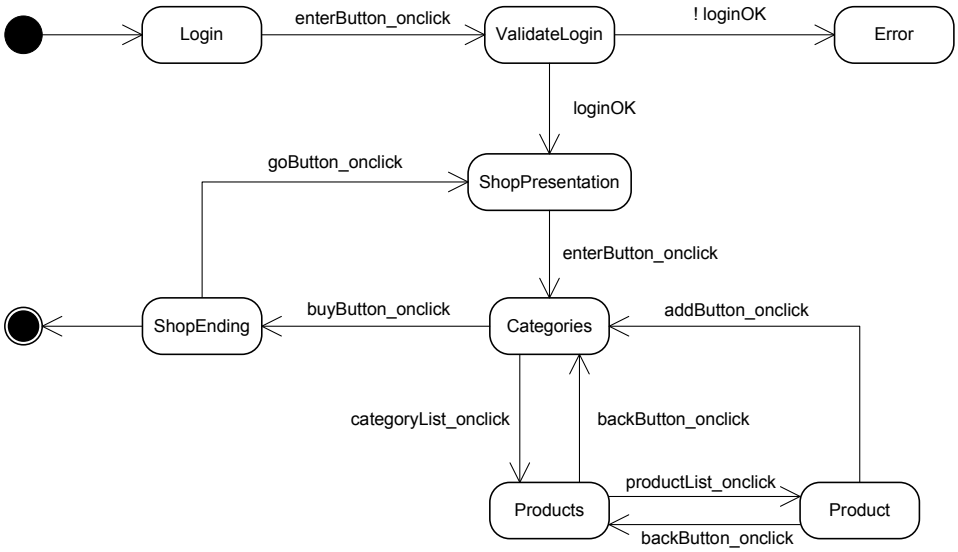

Figure 4: State machine of the example application presented in Section 3.1.

The new `<dimag:invokeMethod>` element represents actions associated to events. It has a `scope` attribute to differentiate between local and remote invocations. The `className` attribute is interpreted differently for local and remote method executions. If the scope is `local`, `className` provides the fully-qualified class name (e.g., `org.dimag.sample.Login`), comprising the namespace or package (`org.dimag.sample`) plus the name of class (`Login`). For `remote` invocations, the relative path is obtained first (`org/dimag/`), and then the name of the component that implements the remote call (the `sample.dll` .NET Compact Framework assembly, or the `sample.jar` package for Java ME and Android). In both scenarios, `method` indicates the method to be called.

For passing arguments and returning values, the framework uses context variables by means of a Java-like expression language (e.g., the `${login}` variable in Figure 5). Variables belong to the same scope and are available in all the states of the workflow, in the views of the user interface, and in the methods invoked with `<dimag:invokedMethod>`.

The transitions between states are triggered considering the `cond` attribute of the `<transition>` element. The conditions that activate a transition could be a user event (e.g., `enterButton_onclick`) or even the result of a method invocation (e.g., in the `validateLogin` method, the condition is the result of the method, kept in the `${loginOK}` context variable).

```
<scxml xmlns="http://www.w3.org/2005/07/scxml"
  xmlns:dimag="http://dimag.org/namespace/flow/custom"
  version="1.0" initialstate="applicationFlow">
<state id="applicationFlow" initial="login">
  <state id="login" category="view">
    <transition event="enterButton_onclick"
                target="validateLogin" />
  </state>
  <state id="validateLogin">
    <onentry>
      <dimag:syncdata id="Users" sync="preaccess"
              level="mandatory">
      <dimag:query>
        for $loginData in doc("users.xml")/users/user
        where $loginData/login="${textFieldLogin}" and
             $loginData/password="${textFieldPassword}"
        return $loginData
      </dimag:query>
      </dimag:syncdata>
      <dimag:invokeMethod scope="server"
            className="org.dimag.main.ValidateLogin"
            method="validateLogin" result="${loginOk}">
        <dimag:argument expression="${loginData}"/>
      </dimag:invokeMethod>
    </onentry>
    <transition cond="${loginOk == 'true'}"
                target=" shopPresentation" />
    <transition cond="${loginOk == 'false'}"
                target="error" />
  </state>
  <state id="shopPresentation" category="view">
    <transition event="enterButton_onclick"
                target="categories" />
  </state>
  <state id="categories" category="view">
    <onentry>
      <dimag:syncdata id="Categories" sync="preaccess"
                      level="optional">
      <dimag:query>
        for $categoryListData
        in doc("categories.xml")/categories/category
        return $categoryListData
      </dimag:query>
      </dimag:syncdata>
    </onentry>
    <transition event="categoryList_onclick"
                target="productList">
      <dimag:syncstate
                value="${categoryList_selectedItem}"/>
    </transition>
    <transition event="buyButton_onclick"
                target="shopEnding">
      <dimag:syncdata id="ShoppingCart"
                   sync="postaccess" level="mandatory"/>
    </transition>
  </state>
  <state id="products" category="view">
    <onentry>
      <dimag:syncdata id="Products" sync="preaccess"
                      level="optional">
      <dimag:query>
        for $productListData
        in doc("products.xml")/products/product
        where $productListData/category =
              "${categoryList_selectedItem.category}"
        return $productListData
      </dimag:query>
      </dimag:syncdata>
    </onentry>
    <transition event="productList_onclick"
                target="product" >
      <dimag:syncstate
                   value="${productList_selectedItem}"/>
    </transition>
    <transition event="backButton_onclick"
                target="categories" />
  </state>
  <state id="product" category="view">
    <onentry>
      <dimag:syncdata id="Products" sync="preaccess"
                      level="mandatory">
      <dimag:query>
        for $productData
        in doc("products.xml")products/product
        where $productData/id =
                    "${productList_selectedItem.id}"
        return $productData
      </dimag:query>
      </dimag:syncdata>
    </onentry>
    <transition event="addButton_onclick"
                target="categories">
      <dimag:syncdata id="ShoppingCart" sync="disabled"
                      level="optional">
      <dimag:query>
        update insert
        <product>
          <id>productList_selectedItem.id </id>
          <desc>productList_selectedItem.desc</desc>
          <prize>productList_selectedItem.prize</prize>
        </product>
        into //products
      </dimag:query>
      <dimag:query>
        update insert
        <product>
          <id>productList_selectedItem.id </id>
          <desc>productList_selectedItem.desc</desc>
          <prize>productList_selectedItem.prize</prize>
        </product>
        into //products
      </dimag:query>
      </dimag:syncdata>
    </transition>
    <transition event="backButton_onclick"
                target="products" />
  </state>
  <state id="shopEnding" category="view">
    <transition event="goButton_onclick"
                target="shopPresentation" />
  </state>
</state>
```

Figure 5: SCXML document defining the state machine in Figure 4.

3.3.3 DIMAG-UI Language

The declarative definition of a DIMAG application also requires the specification of the user interface. DIMAG uses a subset of the IDEAL language from the MyMobileWeb project [12]. Presentation styles are defined by means of CSS documents, referenced from DIMAG-UI. Figure 6 shows the definition of the `categories` view associated to the `categories` state in Figures 4 and 5. Note that an additional CSS document provides the presentation style of the UI controls.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="categories.css" type="text/css"?>
<dimag:presentation id="categories">
    <dimag:title>Online Shop Application</dimag:title>
    <dimag:head id="headApp">Category List</dimag:head>
    <dimag:list id="categoryList" value="${categoryListData}"/>
    <dimag:head id="headApp">Shopping Cart:</dimag:head>
    <dimag:list id="shoppingCart" value="${shoppingCartData}"/>
    <dimag:button id="buyButton">Buy</dimag:button>
</dimag:presentation>
```

Figure 6: User interface definition for the categories view.

Another important issue is the use of context variables in the presentation (e.g., `${categoryListData}`). Context variables are the communication mechanism between the application workflow and the user interface. In some cases, context variables take their values from user input in the application views. In some other situations, they take values from method invocations or declarative data queries performed during the execution of the application, defined in the SCXML document (Figure 5).

3.4 Synchronization Module

The synchronization module allows both data and state synchronization. Data synchronization provides data consistency between the local database in the mobile client and the relational database in the server side. State synchronization is used to inform the server about the new state of the client application, allowing clients to interchange their context variables with the server. This information can be used for different useful purposes, such as statistical reports of preferred user options.

Partially connected architectures are an important concern in mobile environments. Synchronization is performed when connectivity is available; otherwise, the `level` attribute of `<syncdata>` comes into play. This attribute can take two different values: `mandatory` or `optional`. The former implies that the synchronization is compulsory. In this case, if there is no connectivity, the application will show an error. If `level` value is `optional`, data synchronization will not be executed when there is no connectivity. The task will be queued and performed when connectivity is re-established.

3.4.1 Synchronization Policies in DIMAG-Root

Default synchronization configurations can be specified in the DIMAG-Root file (Figure 3), and specific fine-grained adjustments take place in the workflow definition file. Default synchronizations are specified in the

<syncpolicy> and <datamodel> elements. <syncpolicy> simply indicates the synchronization server name and port; <datamodel> specifies the directory that contains the data model, and the default synchronization policy for each single data entity (the defaultSync attribute).

3.4.2 Synchronization Policies in SCXML

Figure 5 shows how the <syncdata> element in SCXML workflow documents has three attributes. First, id provides the identifier of the entity. Then, sync admits four possible values: preaccess, postaccess, both and disabled. preaccess and postaccess indicate that synchronization takes place before and after the query, respectively; both performs the two previous synchronizations; and disabled indicates that data will only be stored in the local storage. The third attribute, level, indicates whether the synchronization is mandatory or optional.

Queries are executed during synchronization steps, saving their results in context variables specified with the return keyword. The values of these variables can be later used in the application workflow. The validateLogin state in Figure 5 shows an example where the result of the query is saved in ${loginData}. Afterwards, that variable is the argument of the ValidateLogin method invocation in the same state.

The XQuery expressions used in the workflow definition provide a simple and powerful mechanism to build declarative applications. They establish a relationship between the application workflow and the data model defined in DIMAG-Root files. Queries operate against the XML entities defined in the data model, and use context variables to intercommunicate the data layer, the application workflow, and the user interface.

Regarding state synchronization, the syntax is simpler than for data synchronization. It is only necessary to indicate the <syncstate> element and the context variable that the client wants to send to the server. For example, Figure 5 shows how to send the categoryList_seletedItem variable to the server in the categoryList_onclick transition of the categories state, using state synchronization (<dimag:syncstate>).

4 Design and Implementation

Figure 7 shows the different phases and elements used in the dynamic generation of client DIMAG applications. When an application is first demanded by a client, the program analysis phase takes the declarative description of a DIMAG program and represents each file as an abstract syntax tree (AST). These ASTs are traversed with different instances of the *Visitor* de-

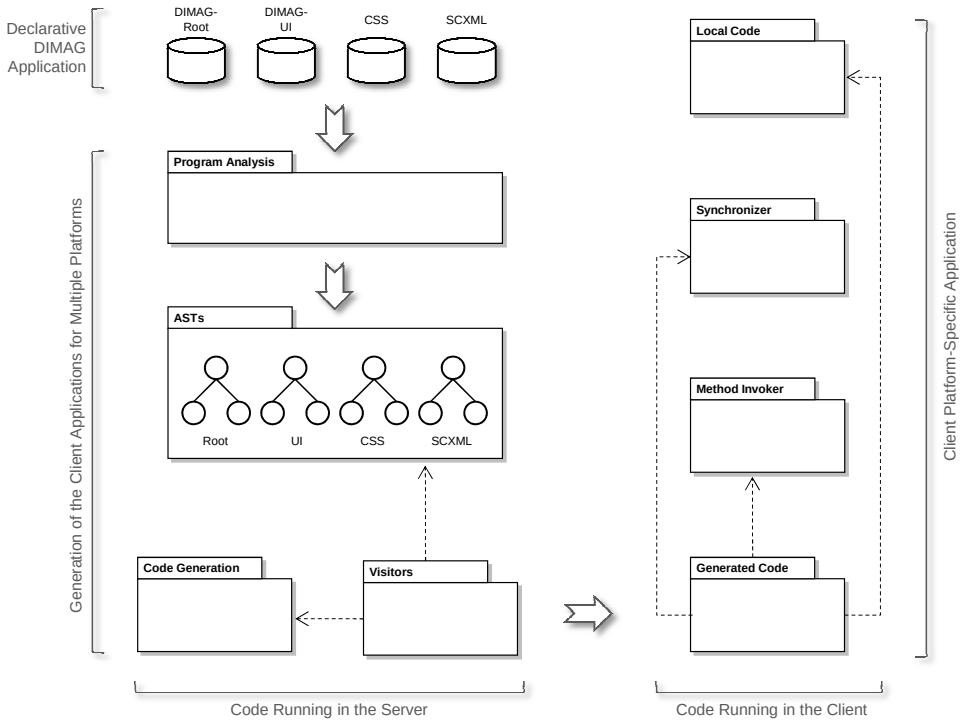

Figure 7: Dynamic generation of DIMAG applications.

sign pattern [14], generating the target code by using the services provided by the code generator package. The generated code is the client platform-specific application that will be deployed in the mobile device. This code requires the specific services of some packages implemented for each target platform: synchronization, remote and local method invocation, and some optional local method implementations. These packages and the generated code are downloaded and executed in the client. If this application is demanded by other clients with the same platform, the compiled application is simply downloaded. If one of the DIMAG specification files is modified, a new application is generated following the same process.

4.1 Program Analysis

The DIMAG-Root, DIMAG-UI, CSS and SCXML documents describing a DIMAG application are analysed before generating the target code. For each document, an AST structure is built in memory. An application may have multiple DIMAG-UI and CSS documents comprising the application presentation layer. All of them are stored in a dictionary of ASTs [15].

We have used the Apache Commons SCXML implementation to process the workflow documents. We have enhanced that tool with specific modules

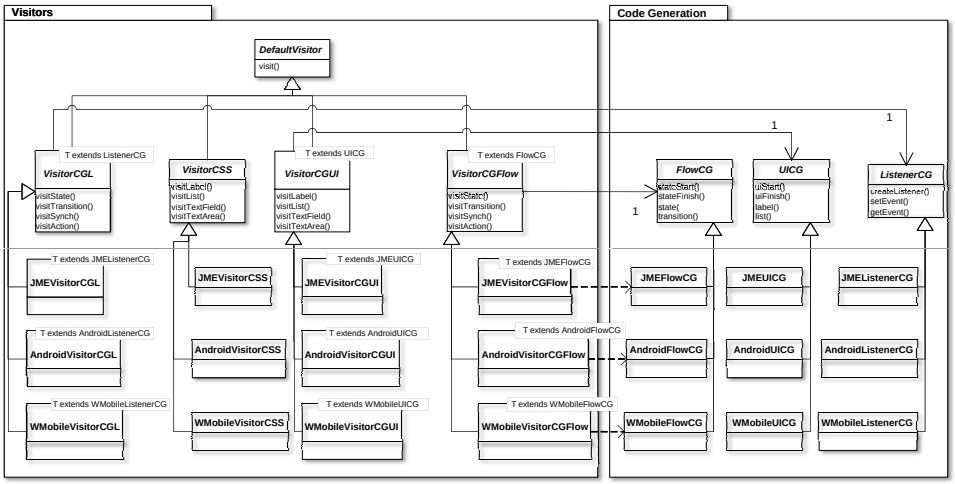

Figure 8: Code generation of DIMAG client applications.

that process the extensions we added to the language. These modules are executed when a `dimag` element is processed by the tool.

4.2 Client-Side Application Generation

The program analyser parses the files specifying a DIMAG application and creates a collection of ASTs in memory. These ASTs are then traversed following the *Visitor* design pattern [14]. Four types of ASTs are defined in DIMAG: user interface (`UI`), style information (`CSS`), application workflow taken from SCXML (`Flow`), and controllers in the MVC architectural pattern (`Listener`).

As shown in Figure 8, there is one visitor for each type of AST, all of them placed in the `Visitors` package: `VisitorCGFlow` takes the application workflow AST obtained from a SCXML document and generates the application model; `VisitorCSS` traverses the UI AST and decorates the tree with the information stored in the CSS AST (i.e., applies the appropriate styles to the UI, without generating code); `VisitorCGUI` generates the specific UI of the application, once the AST is decorated with the style information; and `VisitorCGL` generates the controllers (listeners) in order to map the events triggered by the UI components to the appropriate methods of the application workflow. All the visitor classes derive from `DefaultVisitor` to inherit a default dispatcher using introspection [16].

Figure 8 shows how each visitor hierarchy aimed at generating code (`VisitorCSS` just decorates the UI AST) is associated to a parallel hierarchy of the `CodeGeneration` package, following the *Parallel Hierarchies* design pattern [17]. The responsibility of each visitor class is traversing a specific

type of AST, delegating the work of generating code in the corresponding class of the `CodeGeneration` package. For instance, `VisitorCGFlow` implements the general algorithm for traversing workflow ASTs, calling the methods of the parallel `FlowCG` class that provides the abstract methods for generating the code of application workflows. Following the *Template* design pattern [14], the methods in `FlowCG` can be overridden in the derived `JMEFlowCG`, `WMobileFlowCG` and `AndroidFlowCG` classes that specify how to generate code for the concrete Java ME, Windows Mobile and Android platforms, respectively.

The *Parallel Hierarchies* design pattern also provides the capability of modifying the way an AST is visited for a particular platform. These modifications are implemented in the derived classes of each visitor class (e.g., in the three derived classes of `VisitorCGFlow`). As shown in Figure 8, these derived classes use constraint (or bounded) generics (i.e., the `T` type variable should `extend` another type). This allows specializing the type of the association between the two parallel hierarchies. For instance, the `JMEVisitorCGFlow` class is associated with an instance of type `JMEFlowCG` instead of `FlowCG`, permitting the access to all the specific services provided by the derived class. This use of bounded generics is represented with the dependency relation in Figure 8 between each pair of elements in the two parallel hierarchies[1]. A more detailed explanation of this design pattern is presented in [18].

4.3 Dynamic Addition of New Platforms

DIMAG allows the dynamic addition of new mobile platforms and devices, without restarting the execution of the application server [19]. When a Web browser requests an application, the device platform is identified using the server detection module (Figure 2). As shown in Figure 9, the `AbstractFactory` class loads and instantiates the appropriate visitor and code-generation classes for the selected platform. We use a naming criterion to follow the *Convention over Configuration* (CoC) principle [4], reducing the use of configuration files. For instance, when a DIMAG application is demanded by an Android client, the `createVisitorCGFlow` method is called, passing "`Android`" as the first parameter. An instance of the `AndroidFlowCGclass` is created and passed to the constructor of the `AndroidVisitorCGFlow` class, using Java reflection. The visitor obtained is used to generate the application workflow for the Android device, traversing the SCXML AST. This process is applied to the four visitors shown in Figure 8.

[1]For the sake of readability, we do not show the relationship between the classes derived from `VisitorCGL` and `VisitorCGUI` and the corresponding subclasses of `UICG` and

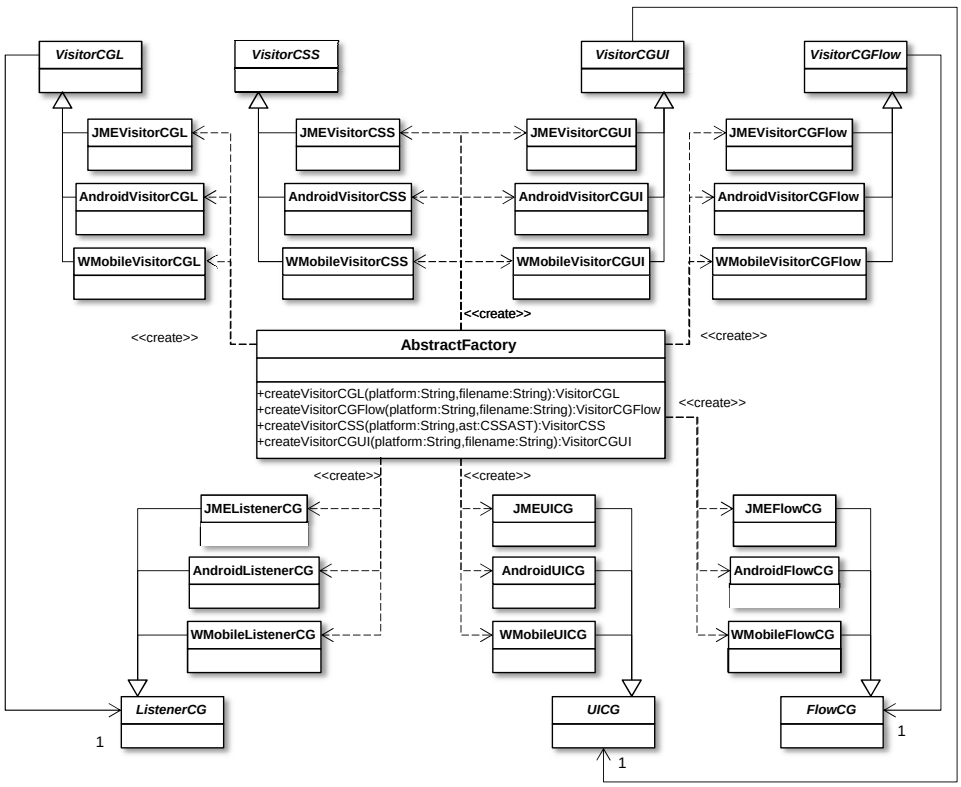

Figure 9: Reflective abstract factory to allow the dynamic addition of new mobile platforms.

In its first prototype, DIMAG generated code for the .Net Compact Framework and the MIDP Java ME platform, using Sun's LWUIT for the user interface [20]. We chose two different languages and platforms to test the language and platform neutrality of DIMAG. Considering the evolution of Smartphones in the last years, we have recently added the support for Android because it covers more than 70% of the market share [3].

DIMAG generates source (C# and Java) code that is later compiled to obtain the binary application. It is worth noting that code generation has to consider not only the different user interfaces, but also the syntactic and semantic differences among the target languages. The main difference to be considered between Java ME and the .Net Compact Framework was probably the way they both implement controllers: LWUIT follows the *Observer* design pattern [14], whereas .Net uses *delegates* to implement UI events. The introduction of Android applications implied more significant changes. Lifecycles of Java ME and Windows Mobile applications are quite similar (Windows Mobile provides an additional *not running* state besides the

`ListenerCG`.

paused and *active* ones), whereas Android defines two more states (*created* and *started*) and more transitions among them. This difference is controlled by the application manager module shown in Figure 2, which considers both the application lifecycle and workflow states. Another important difference we had to deal with was the user interface definition. In Android, view layouts and components are specified in XML files, and UI resources are accessed by the controllers using the `R` class.

4.4 Client Application Services

The visitor classes generate code for the application workflow, user interface and controllers (listeners). The generated code makes use of services that are added to the generated application upon client deployment (Figure 10), but do not need to be generated for each application. These services comprise synchronization, remote and local method invocation, and application-specific native code.

- `Synchronizer` provides the pre- and post-access data synchronization services using SyncML (Funambol). Those states that specify transparent data synchronization use this facility (e.g., `Categories` in Figure 5).
- `MethodInvoker` offers both remote and local method invocation. The `RemoteMethodInvoker` implements SOAP to perform remove invocation to those methods provided by the server as Web services (e.g., the `validateLogin` method used in Figure 5). Local method invocation concerns local native code (next element).
- `NativeCode`. In occasions, it is necessary to implement platform specific routines in a portable application. For example, to access the platform-specific services of the compass in a target device. In DIMAG, a unique interface of this code is defined, but all the implementations for each target platform should be provided (in the `LocalCode` package). As shown in Figure 10, `LocalMethodInvoker` calls the appropriate native implementation using reflection.

4.5 Server-Side Application Generation

The generation of the server side of DIMAG applications is much simpler than generating the client side. First, applications are not generated on demand; an explicit application generation process is run by the administrator. The process takes the SCXML application workflow specification and the

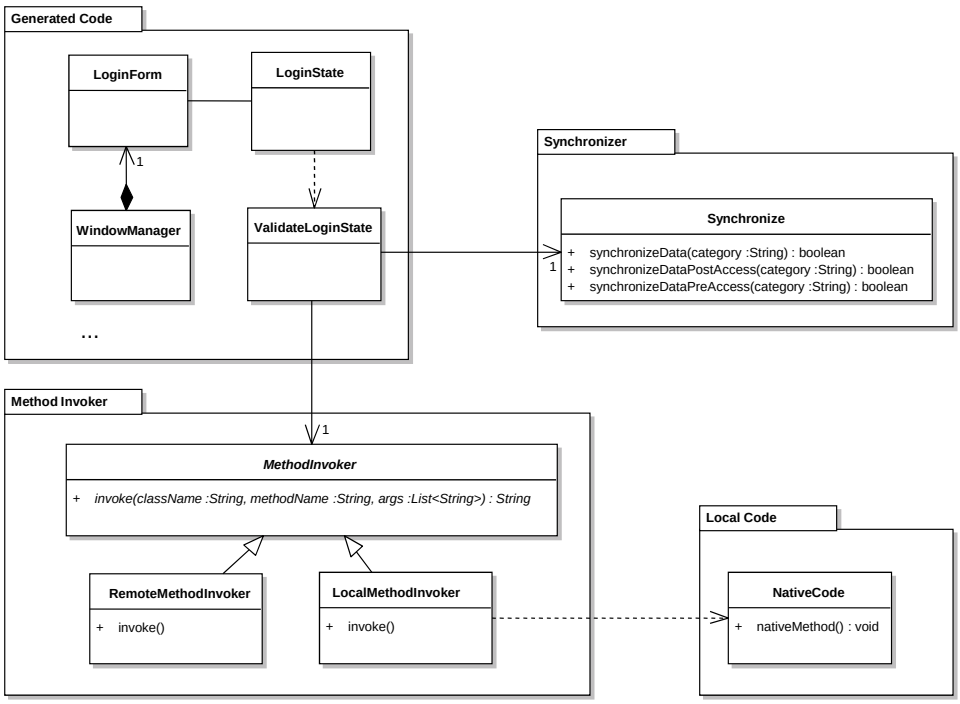

Figure 10: Elements of a client-side DIMAG application.

implementation of all the remote methods. The framework generates the Web services identified as remote methods in the SCXML document, associating their implementations to those provided by the programmer. Finally, a configuration file for the Funambol SyncML server also is created.

5 Related Work

In this section we analyse the existing works related to ours, analysing those systems aimed at the target-agnostic development on mobile devices. We only consider the cross-platform mobile development systems (XMTs) that generate compiled applications, excluding the Web-based approaches (e.g., Sencha Touch). The main differences between Web and native mobile applications are the use of native UI components, variations in user experience, easier access to specific hardware features, and runtime performance [21].

Appcelerator Titanium is an open, extensible development environment for creating mobile applications across different mobile devices and OSs including iOS, Android and BlackBerry [22]. Titanium offers an API to access system functions and to define the graphical user interface (GUI). Applications are imperatively implemented in JavaScript, and the code is interpreted in the client. With a compilation process, Titanium converts the

GUI specification into native views. For data synchronization they provide Titanium Cloud Services, a Mobile Backend as a Service (MBaaS), offering a transparent mechanism to interconnect mobile applications.

PhoneGap is an open-source framework to create mobile applications using standard Web technologies such as HTML, CSS and JavaScript [23]. PhoneGap runs on iOS, Android, Windows Phone, BlackBerry, Symbian and WebOS. Views are expressed in HTML and CSS, and they are executed in the context of an embedded Web browser instead of using the platform native components. Imperative applications are implemented in JavaScript, which is executed in the client device by a native engine. PhoneGap also provides a plug-in structure to allow accessing platform-specific code via native-to-JavaScript PhoneGap bridges.

Xamarin (formerly MonoTouch) is a commercial platform for creating cross-platform mobile applications using .NET technologies [24]. It provides a library that exposes a set of APIs for accessing common mobile device functionality across iOS, Android, and Windows Mobile. Xamarin is distributed with the Mono implementation of the .NET runtime, providing JIT-compilation, memory management and the .NET base class library. Applications are coded in the C# imperative programming language. Both the application flow and the GUI are (JIT- or AOT-) compiled to native binaries.

MobDSL addresses the problem of device fragmentation by defining a declarative Mobile Domain Specific Language called MobDSL [25]. This language is later processed to create platform specific applications. The proposed architecture identifies the tier representing applications written in MobDSL, the MobDSL language engine and libraries, and the target platform tier. MobDSL allows defining display elements, events, hardware features and widget containers. Each widget is a state machine that handles events by making a state transition to a new widget. Neither remote method invocation nor client-server development is considered. The current implementation is a Java interpreter prototype running on Android [25].

Corona is a mobile application development framework for iOS, Android, Kindle Fire and Nook [26]. Both the imperative applications and the display elements are implemented in Lua. The program is compiled obtaining a native application and GUI for the target platform. It provides an OpenGL rendering engine to allow full hardware acceleration of graphics. The Cloud Sync functionality of the Corona Cloud service allows synchronizing cloud-based application data across multiple platforms.

RhoMobile (formerly Rhodes) is an open-source XMT developed for building native mobile applications [27]. It follows the MVC architectural pattern. Views are written in HTML and controllers in the Ruby imperative

language. RhoMobile programs are compiled, generating native applications for iOS, Android, Windows Mobile, BlackBerry and Windows Phone. RhoConnect implements a synchronization engine that tracks the data on each device, identifying and synchronizing only the data that has changed on mobile devices and servers.

MoSync is a free open-source software development kit for mobile applications, integrated with the Eclipse development environment [28]. The MoSync framework produces native mobile applications using C, C++, HTML 5, CSS and JavaScript. MoSync applications can be compiled to Android, iOS, Symbian and Windows Phone. Applications can access the native GUI system of Android, iOS and Windows Phone devices. It also supports widgets for embedding Web pages and OpenGL views in applications.

The DIMAG framework presented in this paper is an improvement of a previous version [20]. In the first prototype, we covered the generation of Java ME (MIDP) and Windows Mobile (.Net Compact Framework). Due to the current market trend indicating that Android is the most widespread mobile operating system (70.1% at the end of 2012 [3]), we included Android OS to the new version. We have also added the support of XQuery as a declarative language to retrieve and update data expressed in XML. Furthermore, the SyncML protocol has been incorporated in DIMAG, providing a transparent way to synchronize data between the server and the client. Three synchronization policies have been defined for this purpose: `preaccess`, `postaccess` and `both`. We have redesigned the existing code generation module to add the support of Android. For this purpose, we have used the *Parallel Hierarchies* pattern described in Section 4.2, improving the dynamic generation of platform-neutral client applications. We think this design, which has been successfully used before to implement a retargetable compiler [29], will facilitate the future generation of applications for new platforms.

6 Qualitative Evaluation

We have qualitatively evaluated the systems analysed in the previous section. The objective of this evaluation is not to say which system is best, but to identify the main contributions and limitations of each approach. These are the features evaluated for each platform, along with the general assessment criteria used (results are presented in Table 1):

1. Scalability of the analysed system to support complex applications. DIMAG has only been used to create simple applications, and MobDSL

Feature	Titanium	PhoneGap	Xamarin	MobDSL	Corona	RhoMobile	MoSync	DIMAG
1. Scalability	●	●	●	○	●	●	●	◗
2. Support of new platforms	◗	◗	◗	◗	◗	◗	◗	●
3. Declarative workflow specification	○	○	○	●	○	○	○	●
4. Real applications	●	●	●	○	●	●	●	○
5. Transparent synchronization	◗	○	○	○	◗	◗	○	●
6. No performance overhead of app. generation	◗	◗	●	◗	◗	◗	◗	○
7. Native GUI	●	○	●	●	●	●	●	●
8. Declarative description of data	○	○	○	○	○	○	○	●
9. Number of target platforms	●	●	◗	○	◗	●	◗	◗
10. Compiled execution	○	○	●	●	●	●	●	●

Table 1: Comparison of the existing cross-platform mobile development systems (● represents that the feature is fully supported, ◗ means partial support, and ○ indicates no support).

does not seem to have a stable prototype yet. The rest of systems have been used to create complex programs.

2. Services for supporting new platforms. The analysed XMTs use their existing code when new platforms need to be supported. DIMAG has been designed to facilitate this process, providing an open code-generation approach, and following the *Convention over Configuration* principle to allow adding new platforms at runtime (Section 4.3).

3. Declarative specification of application workflow. Only DIMAG and MobDSL provide a declarative specification of application workflows; the rest of systems follow an imperative approach.

4. Use of the platform for the development of real applications. All the systems have been used to create production applications except DIMAG and MobDSL, which are research prototypes.

5. Transparent synchronization of data between client and server. DIMAG is the only approach that declaratively identifies the transparent synchronization of data in the application workflow. Titanium, Corona and RhoMobile provide different services for synchronizing data between the server and client applications.

6. No runtime performance overhead caused by the dynamic generation of applications. DIMAG is the only platform that dynamically generates an application the first time it is demanded for a specific platform. Although the binaries are not re-generated and re-compiled the following times the same version is demanded, the dynamic generation and compilation of client applications involves a runtime performance cost

in the DIMAG server. The rest of systems except Xamarin require the compilation of applications, but this compilation does not necessarily have to be done at runtime. Xamarin support JIT-compilation of applications in the client devices.

7. Native GUI. All the systems generate native GUIs except PhoneGap.

8. Declarative description of the data to be synchronized. Only DIMAG describes the data to be synchronized declaratively.

9. Target platforms. We have assigned the maximum score to those XMTs that generate code for the three most widespread platforms [3] (Android, iOS and BlackBerry): Titanium, PhoneGap and RhoMobile. Intermediate score is assigned to the platforms that provide more than 75% of the market share, according to IDC [3]: Xamarin, Corona, MoSync and DIMAG.

10. Compiled execution of the application workflow (no interpretation). All the systems but Titanium and PhoneGap generate a native implementation of the application workflows.

Compared to the related systems, the distinguishing feature of DIMAG is the declarative specification of application workflow and data that facilitates the transparent synchronization of application data and state. This feature, together with the use of flexible code-generation techniques, is used to reduce the cost of adding new platforms and devices, and to generate the native implementation of applications. Currently, its main disadvantage is that the platform has not been tested in the development of real and complex applications.

7 Conclusions

The DIMAG framework shows how the declarative specification of client-server mobile applications can be an appropriate approach to implement a platform-independent development system capable of generating both client and server sides of native applications. In this paper, we propose a specific declarative description of this kind of applications, the architecture and design of a flexible and extensible system to generate target applications, and the technologies and standards used for its implementation. All these elements make up DIMAG, a framework for creating native mobile applications for multiple software platforms. DIMAG facilitates the addition of new target platforms and devices at runtime. The feasibility of the proposed system has been tested with the implementation of three different target platforms: Android, Java ME and Windows Mobile.

We are currently working on generating iOS applications to cover more than 90% of the Smartphone OS market share [3]. The main differences between iOS and the existing target platforms (e.g., a different native programming language, defining views and transitions based on *storyboards*, and the generation of XIB files) entail a challenge to validate the design of the code generation module. We will also use DIMAG to define more complex real applications. These applications will be implemented with the XMTs analysed in Section 5, comparing the runtime performance, memory usage and battery consumption of those approaches and DIMAG. Finally, due to the involvement of the authors in the W3C MBUI working group, another future task will be the alignment of the user interface definition with the recommendations released by the W3C.

Acknowledgements

We would like to thank the anonymous reviewers for their detailed lists of indications, corrections and suggestions that have helped us to improve the article. This work has been partially funded by the European Commission's Seventh Framework Program under grant agreement number 258030 (FP7-ICT-2009-5; Internet of Services, Software and Virtualization STREP). We have also received funds from the Department of Science and Technology (Spain) under the National Program for Research, Development and Innovation: project TIN2011-25978 entitled *Obtaining Adaptable, Robust and Efficient Software by including Structural Reflection to Statically Typed Programming Languages*.

References

[1] International Telecommunication Union. Measuring the Information Society, The ICT Development Index; 2012. `http://www.itu.int/ITU-D/ict/publications/idi/material/2012/MIS2012_without_Annex_4.pdf`.

[2] Rajapakse DC. Fragmentation of Mobile Applications. Handbook of Research on Mobile Software Engineering; 2008.

[3] International Data Corporation (IDC). Android and iOS Combine for 91.1% of the Worldwide Smartphone OS Market in Fourth Quarter of 2012; 2013. `http://www.idc.com/getdoc.jsp?containerId=prUS23946013`.

[4] Thomas D, Hansson DH, Schwarz A, Fuchs T, Breed L, Clark M. Agile Web Development with Rails. A Pragmatic Guide. Pragmatic Bookshelf; 2005.

[5] Gudgin M, Hadley M, Mendelsohn N, Moreau JJ, Nielsen HF, Karmarkar A, et al.. SOAP Version 1.2; 2007. http://www.w3.org/TR/soap12-part1.

[6] Ballinger K, Ehnebuske D, Gudgin M, Nottingham M, Yendluri P. WS-I Basic Profile Version 1.0; 2004. 2.

[7] Rabin J, Fonseca JMC, Hanrahan R, Marin I. Device Description Repository Simple API, W3C Recommendation; 2008. http://www.w3.org/TR/DDR-Simple-API.

[8] Cantera JM. DDR Simple API, Java Reference Implementation; 2012. http://forge.morfeo-project.org/projects/ddr-ri.

[9] Open Mobile Alliance. SyncML Specifications; 2012. http://www.openmobilealliance.org/syncml.

[10] Funambol. The leading mobile cloud sync solution; 2012. http://sourceforge.net/projects/funambol.

[11] Bitterlich JY. JSR 172: J2ME Web Services Specification; 2011. http://jcp.org/en/jsr/detail?id=172.

[12] Cantera JM. The MyMobileWeb Project (TSI-020400-2010-118); 2012. http://mymobileweb.morfeo-project.org.

[13] The Apache Software Foundation. State Chart XML (SCXML); 2012. http://commons.apache.org/scxml.

[14] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series; 1995.

[15] Ortin F, Zapico D, Cueva JM. Design Patterns for Teaching Type Checking in a Compiler Construction Course. IEEE Transactions on Education. 2007 Aug;50(3):273–283.

[16] Ortin F, Garcia M, Redondo JM, Quiroga J. Achieving Multiple Dispatch in Hybrid Statically and Dynamically Typed Languages. vol. 206 of Advances in Intelligent Systems and Computing. Berlin, Heidelberg: Springer Berlin / Heidelberg; 2013. p. 703–713.

[17] Ortin F, Garcia M. A Type Safe Design to Allow the Separation of Different Responsibilities into Parallel Hierarchies. In: Maciaszek LA,

Zhang K, editors. International Conference on Evaluation of Novel Approaches to Software Engineering. ENASE 2011. SciTePress; 2011. p. 15–25.

[18] Ortin F, Garcia M. Modularizing Different Responsibilities into Separate Parallel Hierarchies. In: Communications in Computer and Information Science. vol. 275. Springer; 2013. p. 16–31.

[19] Ortin F, Redondo JM, Perez-Schofield JBG. Efficient Virtual Machine Support of Runtime Structural Reflection. Science of Computer Programming. 2009;74:836–860.

[20] Miravet P, Marin I, Ortin F, Rionda A. DIMAG: a framework for automatic generation of mobile applications for multiple platforms. In: Proceedings of the 6th International Conference on Mobile Technology, Applications, and Systems. Nice, France: ACM; 2009. .

[21] Corral L, Sillitti A, Succi G. Mobile Multiplatform Development: An Experiment for Performance Analysis. Procedia Computer Science. 2012;10:736–743.

[22] Appcelerator. Titanium Mobile Development Environment; 2013. `http://www.appcelerator.com/platform/titanium-platform`.

[23] PhoneGap. Easily create apps using the web technologies you know and love: HTML, CSS, and JavaScript; 2013. `http://phonegap.com`.

[24] Xamarin. Create iOS, Android, Mac and Windows apps in C#; 2013. `http://xamarin.com`.

[25] Kramer D, Clark T, Oussena S. MobDSL: A Domain Specific Language for multiple mobile platform deployment. In: Proceedings of the IEEE International Conference on Networked Embedded Systems for Enterprise Applications, NESEA 2010, November 25-26, 2010, Suzhou, China. IEEE; 2010. p. 1–7.

[26] Corona. Corona SDK, the ultimate 2D development platform; 2013. `http://www.coronalabs.com`.

[27] RhoMobile. RhoMobile Suite, develop applications for the next generation of business mobility; 2013. `http://rhomobile.com`.

[28] MoSync. App development made easy; 2013. `http://www.mosync.com`.

[29] Ortin F, Palacio DZ, Perez-Schofield JBG, García M. Including both static and dynamic typing in the same programming language. IET Software. 2010;4(4):268–282.