Using Standards to Build the DIMAG Connected Mobile Applications Framework

Patricia Miravet[a], Francisco Ortin[b], Ignacio Marin[a], Abel Rionda[a]

[a]*CTIC Foundation, Research and Development Department, c/Ada Byron s/n, 33203, Gijon, Spain*

[b]*University of Oviedo, Computer Science Department, c/Calvo Sotelo s/n, 33007, Oviedo, Spain*

Patricia Miravet, Francisco Ortin, Ignacio Marin, Abel Rionda. Using standards to build the DIMAG connected mobile applications framework. Computer Standards & Interfaces, volume 36, issue 2, pp. 354-367, February 2014, doi: 10.1016/j.csi.2013.08.007.

Using Standards to Build the DIMAG Connected Mobile Applications Framework

Patricia Miravet[a], Francisco Ortin[b], Ignacio Marin[a], Abel Rionda[a]

[a] *CTIC Foundation, Research and Development Department, c/Ada Byron s/n, 33203, Gijon, Spain*
[b] *University of Oviedo, Computer Science Department, c/Calvo Sotelo s/n, 33007, Oviedo, Spain*

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 hiding the differences among the diverse families and models of mobile devices. This work proposes the DIMAG software framework to generate connected mobile applications for multiple software platforms, employing a declarative description of applications. DIMAG has been implemented taking advantage of existing standards, giving importance to the connectivity of mobile applications. DIMAG applications are based on the traditional client-server paradigm. Server side applications are generated for the Java EE platform, and client side applications are dynamically produced for different mobile platforms. In addition, the possibility of occasional disconnections of the mobile device is taken into account. This problem is tackled with the definition of data and state synchronization policies between the server and the client, using XQuery as the language to access the synchronized data. IDEAL2 has been used to define abstract user interfaces, being rendered to the actual views of the specific target devices. Application workflow definition is the central element of a DIMAG application, defined as state machines serialized in SCXML documents. This language is enriched with several additional XML elements and attributes to add the expressiveness power needed in the proposed framework.

Key words: Device fragmentation, mobile applications, data and state synchronization, partially connected architectures, dynamic code generation

1. Introduction

Development of connected mobile applications is a promising field for companies and researchers. Around 6 billion cellular connection subscriptions (86% of the world population) have been reached by the end of 2012 [1]. 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 (screen size, resolution and color, data input and output mechanisms), operating systems, and other elements of the software stack,

Email addresses: patricia.miravet@fundacionctic.org (Patricia Miravet), ortin@lsi.uniovi.es (Francisco Ortin), ignacio.marin@fundacionctic.org (Ignacio Marin), abel.rionda@fundacionctic.org (Abel Rionda)

makes it difficult to create applications for all the mobile users. This problem is known as *device fragmentation* [2] (or device diversity) within the community of mobile developers and researchers. A common approach to face that problem is to create different versions of the same application for different device models and platforms. This usually leads to variable user experience among platforms. In addition, it multiplies software maintenance efforts, as the company must actually maintain distinct implementations of the same application. 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.

The creation of a development framework to write applications once and execute them in every mobile device is a complex task. The first factor to be taken into account is the huge amount of mobile software environments (operating systems and virtual machines) in the market. The software industry expects mobile application development frameworks to create applications at once for as many target platforms as possible. However, it is a challenging task for companies creating mobile application development frameworks to cover all the software platforms in the market. A reasonable approach seems to be the initial support for a reduced set of the most popular platforms, allowing covering an increasing number of them. Therefore, one of the most important features of mobile application development frameworks is the extensibility in terms of supported target platforms.

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. Examples of these tools are Titanium, Corona, Rhomobile, and PhoneGap. Titanium, Corona and Rhomobile create native applications, whereas PhoneGap is the main exponent for hybrid applications –native applications based on an embedded Web browser. Some facilitate the separation between presentation and behavior based on Web development languages (HTML, CSS and JavaScript in Titanium and PhoneGap). However, all of them are mostly based on imperative approaches for application definition, and they do not consider the transparent generation of distributed applications. To the knowledge of the authors, no approach seems to exist in the market that provides a declarative application definition to generate full native client-server applications.

The main contribution of this paper is DIMAG, a Device-Independent Mobile Application Generation framework, which provides the dynamic generation of client-server applications for any mobile device. Applications are specified with one single declarative specification for each application. DIMAG involves an important amount of technologies, reusing and refining existing standards as much as possible. Besides, extensions to existing standards are also proposed, as for some of the problems to be solved no standard has been proposed yet. Our work intends to be as flexible as possible, leaving several aspects of the proposed framework open in order to cover them in future revisions.

The rest of the article is organized as follows. Section 2 defines the different elements that comprise a DIMAG application, and Section 3 describes the existing standards to define those elements. Section 4 provides an overview of DIMAG, including an example application that facilitates the explanation of the framework modules. It also defines the DIMAG framework by explaining its architecture, and the declarative languages used to create DIMAG applications. Section 5 emphasizes the definition of state and data synchronization policies between the (mobile) client and the server sides, an aspect not sufficiently covered by existing standards. Section 6 explains implementation

issues. Related work is discussed in Section 7 and Section 8 presents the conclusions and future work.

2. Definition of a DIMAG Application

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 typical approach in the traditional programming languages used for building mobile applications). 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. 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 sides of an application are generated: the server side, to be run in an application server; and the client side, to be downloaded and executed in the mobile client. The server 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:

1. Application workflow. An application is modeled as a state machine, made up of a set of states (one of them considered as the initial one) and transitions between states, triggered when a given event happens.
2. 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.
3. 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. Existing Standards

As mentioned, we consider that the use of standards is a key point when creating our platform. Since many technologies need to be used and combined, we have tried to avoid reinventing the wheel, reusing as many existing technologies as possible. Standardized technologies are the result of the effort of organizations and research groups of different nature, with different goals. One of the main expectations of standardization bodies is the usage of the technology and the feedback from users to refine the standard. Therefore, in this paper we analyze the existing standards and propose soft extensions for some of those used to define DIMAG, providing new ideas to refine them.

Since *standard* is a concept that can be interpreted in different ways, we define the intended meaning of this term in this work:

- In the context of computer languages, a standard is a language supported by a standardization organization, an organization that releases recommendations, or any open-source community with a significant number of participants and users.
- In the context of existing software to be reused, a standard is a reference implementation of a language processor of a standard language, or a *de facto* software standard.

3.1. Workflow Definition

The definition of application workflows is a topic that has already been treated by both the research community and the software industry. Most of the existing works are focused on distributed workflow among heterogeneous systems, like the XML Process Definition Language (XPDL [4], which has been created to interchange business process models between different software tools) and the Web Services Business Process Execution Language (WS-BPEL [5], which uses SOAP as the means of communication between parts of distributed applications and services). More recently, the W3C has developed its own workflow definition language, SCXML (State Chart eXtensible Markup Language [6]), which has been created by the Voice Browser working group as a multimodal control language in the Multimodal Interaction Framework.

We have selected SCXML for describing the application workflows in DIMAG. The main reasons were its simplicity, the expressiveness power to define state charts, and the available open-source language processors. The last reason mentioned is especially important for our purposes, because existing SCXML language processors have facilitated the rapid implementation of the DIMAG framework, including the modification and addition of new elements to the W3C specification. We have used the Apache Commons SCXML implementation [7].

WS-BPEL has been discarded as it is directly associated to a specific technology, SOAP, whereas DIMAG has in mind independence of the actual communication technology. XPDL is an interesting technology, but a deep understanding of its definition and its extensions would have taken longer due to its greater complexity when compared to SCXML.

3.2. User Interface Definition

Regarding the declarative languages for user interface definition, several user interface definition languages have already been published in the standards domain. UIML [8] has been developed by the Organization for the Advancement of Structured Information Standards (OASIS). It is aimed at creating declarative definitions of user interfaces by means of six orthogonal pieces: definition of the UI parts, presentation (look/feel/sound) used for those parts, content (text, images, sound), behavior of the UI, mapping of the UI parts to controls in a given toolkit, and the business logic components the UI is connected to. There are implementations of UIML renderers [9, 10] that could have been reused to implement the user interface module of the DIMAG framework. However, the activity of OASIS on the development of the specification is stopped since 2008, and some of the participants in its definition have moved their efforts to other languages.

UsiXML [11] is one of the languages proposed by some of the former participants in the definition of UIML. As proposed in the CAMELEON project [12], UsiXML supports different abstraction levels when defining user interfaces: Abstract User Interface (AUI) for a high-level UI definition, independent of the interaction modes or target platforms; Concrete User Interface (CUI) for a UI definition that refines the AUI level, considering aspects devoted to interaction modes; and Final User Interface (FUI), which results from the conversion of the CUI to the format in which the application is executed in a target platform. AUI and CUI definitions would be supported by the

UsiXML specification, whereas FUI is the translation to the different execution platforms. Existing UsiXML processors have been analyzed in order to improve user interfaces in DIMAG [13, 14]. Unlike UIML, which is supported by a standardization organization (OASIS), UsiXML cannot be considered as a standard because it is defined by an industrial consortium advised by another research consortium. In addition, UsiXML focuses in the transformations between the AUI and the CUI, and does not describe the transformation process from CUI to FUI.

The choice for UI definition in the DIMAG framework has been IDEAL2 [15], the language used in the MyMobileWeb open-source project [16]. This project implements a modular and open-standards-based software platform that facilitates the development of mobile Web applications and portals, providing an advanced content and application adaptation environment. IDEAL2 uses concepts from W3C recommendations such as XHTML (e.g., content structuring and separation of content and style by means of CSS), and W3C technical reports, such as DIAL [17] for content selection and filtering depending on device features. Although IDEAL2 is not supported by any standards-developing organization, it has been created by consensus of several universities, technology centers, and companies that participate in the MyMobileWeb project [16]. In addition, the definition of IDEAL2 is currently being considered by the W3C MBUI working group for the definition of a CUI language for graphical interaction. The experience and participation of the authors of this article in the MyMobileWeb project and their knowledge of the IDEAL2 language processor have also motivated the choice of this language.

3.3. Synchronization

The definition of the datasets to be synchronized between the server and the client has been done with a standard language defined by the W3C Recommendation: XQuery [18]. However, no standards have been found in what regards to data and state synchronization. This is the reason why the authors have defined a declarative mechanism to specify the synchronization of data between the server and client sides of applications (detailed in Section 5.3).

XQuery is a declarative language designed to perform queries on data collections expressed in XML. Its main goal is the extraction of information from a dataset organized as a tree of elements, independently of the data origin. It is a functional language, so every query is an expression returning a result. Expressions can be combined in a flexible way, creating more complex and powerful expressions.

Traditional implementations of XQuery like Saxon [19] have a size of 3 MBs. This fact prevents developers from using them in mobile devices. For that reason, the developer community has created MXQuery [20], a lightweight but full-featured XQuery implementation with a reduced the size of 1 MB. We used MXQuery for Android OS, but it is still too big to be executed in low-end mobile devices. Therefore, for this kind of devices, we have developed an alternative implementation that supports a small set of features of the XQuery specification.

In this subsection, we have introduced the standards used to tackle the synchronization of information in the DIMAG framework. All the details concerning its design are discussed in Section 5.3.

4. Overview of the DIMAG Framework

There exist previous works in the generation of user interfaces for multiple software platforms [21], techniques to link declarative user-interface definitions to existing platform-dependent code [14], and authoring tools to create device-independent applications [21]. DIMAG raises the necessity to create a holistic solution to the problem of creating connected applications for mobile

devices, whereas previous research and development efforts have dealt with some of the different aspects related to the DIMAG proposal. Such a proposal must not only face the problem of generating the code of a distributed client-server application for different software platforms; it must also manage the lifecycle of applications (build process, server-side deployment, client software provision to mobile devices, installation, execution, and client and server version update), and the different types of runtime client-server interactions (e.g., data and state synchronization between client and server).

4.1. A Motivating Example

The following motivating example shows an example use of the DIMAG framework at its current stage. The example simulates a simple online shop in which users can search products through different categories, and store the selected products in a shopping cart until the final completion of the purchase. Figure 1 shows the navigation flow of the motivating example. Each screen is labeled with its identifier, and directed arrows suggest some of the possible transitions between them. These arrows are interrupted in Figure 1 by boxes indicating the information synchronized between the server and the client when the corresponding transition occurs.

In the client-server architecture followed (Section 5.1), the client implements a simple persistence layer that reflects part of the persistence system in the server side. Changes on data occur periodically in both sides of the application, and the synchronization policy defines when these changes are propagated between both sides of the application. The data synchronization policy keeps the data layer up-to-date in both sides seamlessly, 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, this 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 or 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 legibility purposes (see Section 5.3), 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. Then, the selected item is transparently transmitted to the server for statistical purposes, saving the categories and products commonly selected among the users.

5. Definition of the Framework

5.1. Architecture

Figure 2 shows the architecture of the DIMAG framework. The left part of the figure shows the client side of a DIMAG application, whereas the right part presents the server 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 (and install) in their mobile devices.

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

- 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

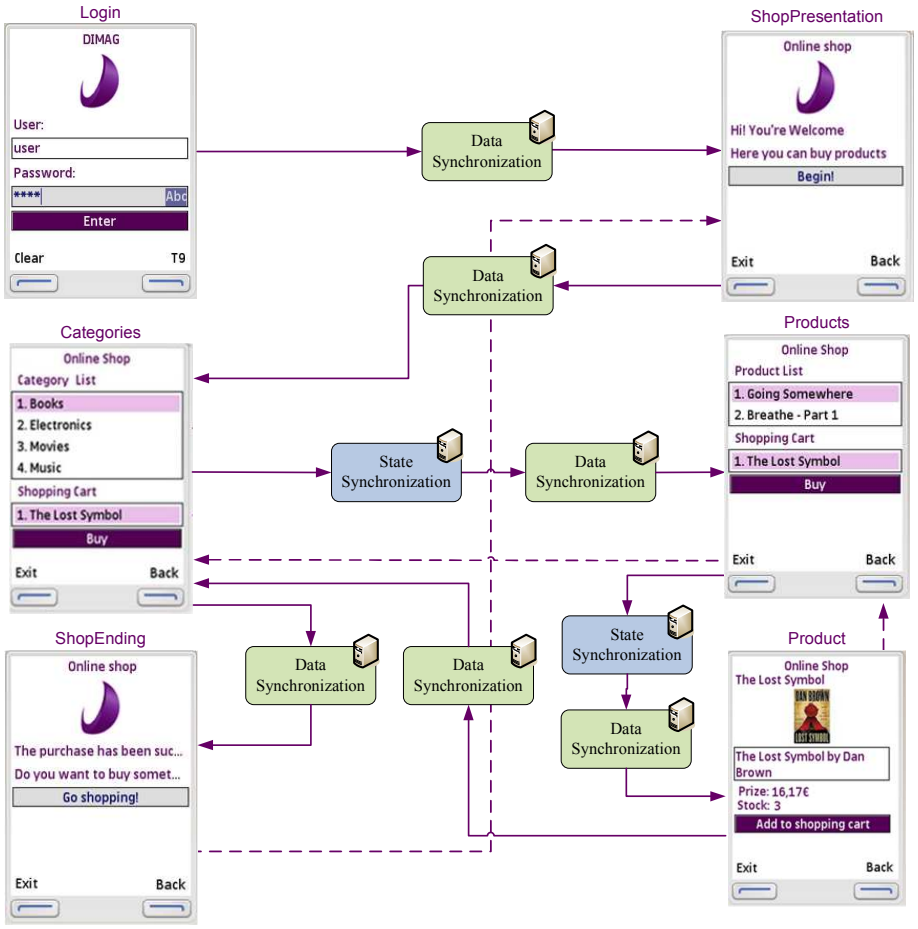


Figure 1: Visual description of an example DIMAG application.

application, and those HTTP requests performed by the mobile application logic. Regarding standards, the DIMAG framework uses the JSR-154 Java Servlet 2.4 Specification [22]. For Web services invocation, SOAP [23] and WS-I Basic Profile [24] are used. The latter is used to ensure Web services interoperability among different implementations, ready to work in devices with resource constraints.

– The device detection and application provision module, which receives information from HTTP requests when users try to download an application. Evidences are extracted from those requests in order to identify the software platform of the device, 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 device 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 Device Description Repository Simple API [25] has been used to design this module, choosing the DDR Simple API Reference

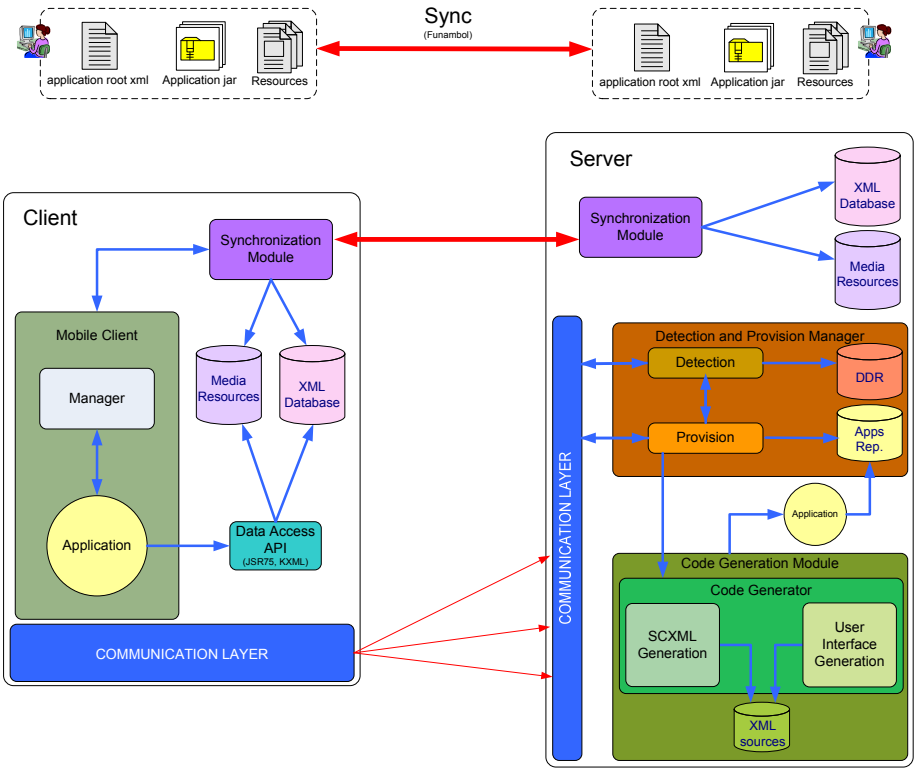


Figure 2: Architecture of the DIMAG framework.

implementation within the open-source Morfeo Project [26].

– The code generation module, which dynamically generates the platform-specific client applications selected by the user, as commanded by the second module. If the DIMAG framework does not support the implementation 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 applications. The framework is designed to be able to support code generation for new target software platforms at runtime, without needing to restart its execution.

– The synchronization module is aimed at synchronizing information between the server and the client. This includes data synchronization (between the server database and the local client storage) and synchronization of the running workflow state between both sides of the application. The main goal is to obtain a simple declarative system enabling the tuning of the level of data synchronization desired for an application. For the sake of simplicity, data sources to be synchronized are simple files (e.g., images and XML documents). 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 data files (as a consequence of changes in the mobile client side) also trigger updates of the relational database. DIMAG uses the SyncML [27] protocol and the Funambol open-source synchronization server [28] 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 in 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 initializing the application, and managing the specific features of the particular target platform. 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 [29] 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 becomes 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.

5.2. 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 XML language, a simplification of MyMobileWeb IDEAL2 [15]. This language fulfills the requirement of separating the content from the user interface and its style, referring to external CSS files. Although workflow definition is expressed in SCXML, we propose several extensions to the language (Section 5.2.2) to provide all the elements required in the DIMAG workflow description.

Synchronization information (data sources and data synchronization policies) is currently placed in the DIMAG-Root and the SCXML languages. This has been done by extending the previous definition of the DIMAG framework [30]. Future versions of DIMAG will probably separate data definition and synchronization in different modules.

5.2.1. DIMAG-Root Language

The DIMAG-Root document used in the motivating example described in Section 4 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`> indicates the URI to an SCXML document describing the application workflow.
- <`ui`> provides the 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

```
<?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.

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 defining data and state synchronization policies between the server and the client. In the example in Figure 3, it is only indicated the URI of the synchronization server.
- <`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 5.3).
- <`server`> provides the server endpoint used for data and state synchronization, and remote method invocation.
- <`resources`> specifies the URIs to the external resources needed 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.

5.2.2. SCXML

Application designers and engineers commonly propose a set of mockups (screen drafts) laid out in a diagram, linking them with arrows that suggest the possible transitions from one screen to another one. They usually annotate these transitions with comments that describe under which conditions the transition between two screens take place. This procedure is carried out either by hand or by means of mockup generation tools, such as Pencil [31] or Axure [32]. Such diagrams are

an informal way to depict application workflow similar to state chart diagrams. A state diagram provides an abstract description of the behavior of a system. More particularly, they facilitate the description of an application by representing it as a set of states, connected by means of transitions. Following the same approach, the workflow for each DIMAG application is defined as a state machine. Figure 4 depicts the state machine of the example application shown in Figure 1. Rounded-corner rectangles represent the different states of the application. Each state may imply the execution of actions. In some cases, the actions carried out when the application is in a specific state include the representation of a screen on the device display. This happens in the `Login`, `Error`, `ShopPresentation`, `Categories`, `Products`, `Product`, and `ShopEnding` states. This is the reason why they are marked with a `V` in the upper right corner of the representation of the corresponding states. In the rest of the cases, actions are performed without updating the device display (`ValidateLogin`).

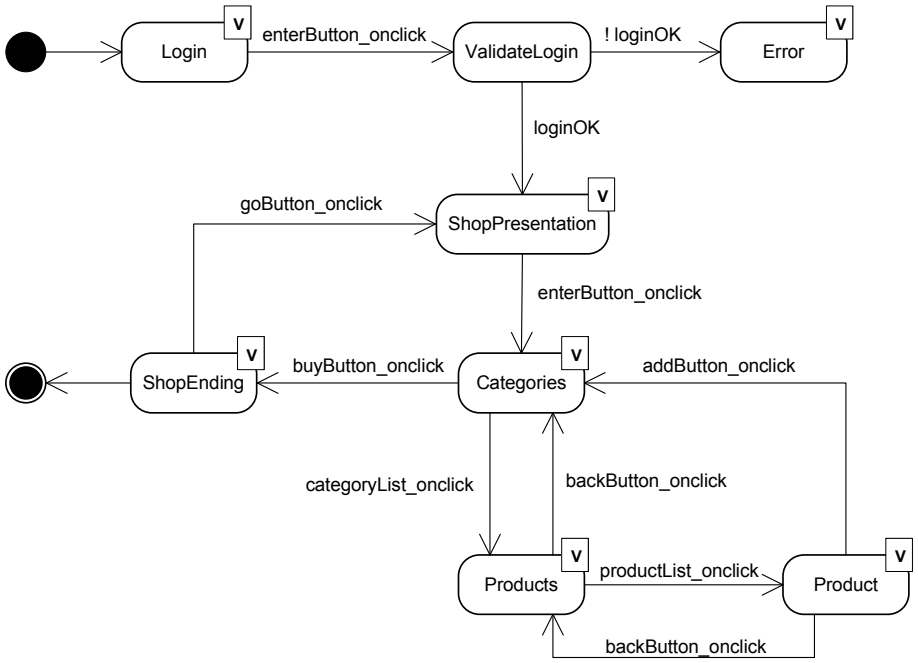


Figure 4: State diagram of the example application shown in Figure 1.

Figure 4 also shows how transitions between states are represented as directed arrows. The direction of the arrows implies that the state machine changes from a source state to a destination state. Special symbols mark specific states as initial (black circle) and final (black circle with surrounding white circumference) states. Arrows are labeled with the event or condition that triggers the associated transition. In the example application, most of the transitions are triggered by the event of clicking a button in the source state. In contrast, the mutually exclusive transitions from `ValidateLogin` to `Error` and to `ShopPresentation` depend on the value returned by the server after login and password submission, validating or rejecting user authentication.

This is a description of the application workflow state diagram shown in Figure 4:

– The application starts in the `Login` state, displaying the `Login` screen in Figure 1. In this

state, two variables (`user` and `password`) are set with the values introduced by the user in the `User` and `Password` text fields. When the `enterButton` button (captioned as "Enter") in the `Login` screen is clicked, the `enterButton_onclick` event is raised and, subsequently, the transition from the `Login` state to `ValidateLogin` takes place.

- The `ValidateLogin` state submits the login and password to the server. The returned value indicates whether the user is successfully authenticated. This value triggers the transition to either the `ShopPresentation` (true) or `Error` (false) states.
- The `Error` state (not shown in Figure 1 for the sake of clarity) is a view state showing an error message. It allows getting back to the previous state in which the error condition was detected.
- `ShopPresentation` shows the `ShopPresentation` screen, with a welcome message confirming the successful login. Clicking on the `enterButton` button (captioned as "Begin!") raises an `enterButton_onClick` event that triggers the transition to the `Categories` state.
- The `Categories` state renders the homonymous view and asks the server for the existing list of categories, and the products currently included in the shopping cart. The list of categories is shown in the "Category List" listbox. A `categoryList_onClick` event is triggered when the user selects a category, changing to the `Products` state. The selected category is passed as an argument in the transition.

  The "Shopping Cart" listbox shows the list of products selected by the user. When the "Buy" button is clicked, a `buyButton_onClick` event is raised, and the purchase process is considered as completed (a transition to `ShopEnding` takes place).
- The `Products` state is similar to `Categories`, but displaying all the items available for the selected category. The selection of any of the items in the product list triggers the `productList_onClick` event, involving a transition to `Product`.
- The `Product` state shows the detailed information of the product selected. The `Product` screen includes a "Add to shopping cart" button that serves as a confirmation for the inclusion of the item in the shopping cart.
- The `ShopEnding` state is reached after purchasing the products in the shopping cart. It allows starting another purchase process (clicking on "Go shopping!"), triggering a transition to `ShopPresentation`.
- All the screens provide both the "Exit" and "Back" buttons, allowing application termination and returning to the previous screen, respectively. Therefore, all the view states allow the transition to the immediately previous view state visited (and program termination).

DIMAG uses the SCXML language to express application workflow as state machines. SCXML was created by the W3C Voice Browser working group. We have implemented the processing of SCXML documents by reusing the Apache Commons SCXML open source project [7]. Figure 5 shows the SCXML document specifying the state machine illustrated in Figure 4. SCXML [6] defines states by means of the <`state`> element, setting one of them as the initial one (`initialstate`). It also allows defining the transitions between states, and the actions to be performed when entering (<`onentry`>) and leaving (<`onexit`>) a state.

```
<scxml xmlns="http://www.w3.org/2005/07/scxml"
 xmlns:mobapp="http://my.custom-actions.domain/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=" postaccess" 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:syncdata>
  </transition>
  <transition event="backButton_onclick"
          target="products" />
 </state>
 <state id="shopEnding" category="view">
  <transition event="goButton_onclick"
          target="shopPresentation" />
 </state>
</state>
```

Figure 5: Document defining the state machine in Figure 4.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="categories.css" type="text/css"?>
<dimag:presentation id="categories">
  <dimag:title>Online Shop</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.

Some elements and attributes of the application workflow description have been added to the SCXML standard, included in the `dimag` namespace. This new namespace permits the SCXML language processor to differentiate the standard SCXML lexical elements from our language extensions. We have implemented additional modules to process these new elements defined for the DIMAG framework. The new modules decorate the data structure generated by the SCXML implementation representing the workflow state machine.

The new <`dimag:invokeMethod`> element represents the actions associated to the transitions. <`dimag:invokeMethod`> has an attribute named `scope` to differentiate between local and remote invocations. The `className` attribute is interpreted differently depending on the value of `scope`. 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 global 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 carried out by 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).

5.2.3. DIMAG-UI Language

The declarative definition of a DIMAG application also requires the specification of the user interface. DIMAG uses a subset of the IDEAL2 language from the MyMobileWeb project [15]. 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 user interface controls. Another important issue is the presence of context variables in the presentations (e.g., `${categoryListData}`).

The context variables are the communication mechanism between the application workflow and the user interface. In some cases, context variables take their values from the user input in the application views. In some other situations, they take values from method invocations or data queries performed during the execution of the applications, as defined in the SCXML workflow

specification (Figure 5).

5.3. Synchronization Module

The synchronization module defines 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. Moreover, this state synchronization allows clients to interchange their context variables with the server. This information can be used for different useful purposes, such as statistical reports of the preferred user options.

As mentioned, partially connected architectures are an important concern in mobile environments. For this purpose, synchronization can be performed when connectivity is available; otherwise, the `level` attribute of <`syncdata`> element comes into play (Figure 7). This attribute can take two different values: `mandatory` or `optional`. The former implies that the synchronization is compulsory. Therefore, if there is no connectivity, the application will show an error, warning the user about the need for connection before continuing the application execution. For example, in the SCXML document in Figure 7, the user login and password are compulsorily sent to the server in order to authenticate the user.

In case the level attribute has the `optional` value, the data synchronization task will not be executed if there is no connectivity. The task will be queued and performed when connectivity is reestablished. In the `Product` state in Figure 7, the product selected by the user is optionally sent to the server when the `addButton` is clicked. In case there is no connectivity, this information is stored locally and updated afterwards, when the connection is restored.

```
<scxml xmlns="http://www.w3.org/2005/07/scxml"
  xmlns:mobapp="http://my.custom-actions.domain/CUSTOM"
  version="1.0" initialstate="applicationFlow">
<state id="applicationFlow" initial="login">
  ...
  <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>
  ...
  </state>
```

```
...
  <state id="product" category="view">
    ...
    <transition event="addButton_onclick"
            target="categories">
      <dimag:syncdata id="ShoppingCart"
                  sync=" postaccess" 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:syncdata>
    </transition>
    ...
  </state>
...
</state>
```

Figure 7: <`syncdata`> element in workflow definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application SYSTEM "application.dtd">
<application xmlns='http://dimag.org/namespace/application'>
    ...
    <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>
    ...
</application>
```

Figure 8: Synchronization elements in DIMAG-Root.

5.3.1. Synchronization Policy in DIMAG-Root

The default synchronization configuration is specified in the DIMAG-Root file (Figure 3), and specific fine-grained adjustments take place in the workflow definition file. Default synchronization in DIMAG-Root is specified in the <`syncpolicy`> and <`datamodel`> elements (Figure 8). <`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). This attribute admits four possible values: `preaccess`, `postaccess`, `both` and `disabled`. `preaccess` and `postaccess` indicate that synchronization takes place before and after the query, respectively; `disabled` indicates that data will be updated in the mobile device only; and `both` represents `preaccess` plus `postaccess`.

5.3.2. Synchronization Policy in the SCXML Document

Figure 7 shows how the <`syncdata`> element in a SCXML workflow definition document has three attributes. First, `id` provides the identifier of the entity which is accessed. Then, `sync` is the particularization of the `defaultSync` attribute explained in 5.3.1 (`preaccess`, `postaccess`, `both` or `disabled`). If it was not included in the declaration of the <`syncdata`> element, the default value specified in the DIMAG-Root document will be used. The third attribute, `level`, indicates whether the synchronization is mandatory or optional.

The data to be synchronized is declaratively specified by means of XQuery expressions, which are executed upon synchronization. The results of all the queries are stored in the application context using the `return` keyword, and they can be later used in the application workflow. An example is the `ValidateLogin` state in Figure 5. The result of the query is saved in the `${loginData}` context variable. 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. The XQuery features used in the DIMAG framework

are compatible with the W3C XQuery 1.0 recommendation.

The syntax of state synchronization is simpler than for data synchronization. It is only necessary to indicate the <`syncstate`> element and the context variable that the client will send to the server. For example, Figure 5 shows how simple it is to send the `categoryList_selectedItem` variable to the server in the `categoryList_onclick` transition of the `Categories` state, using state synchronization (<`dimag:syncstate`>).

5.3.3. The Synchronization Process

As mentioned, DIMAG performs client-server file synchronization using the SyncML protocol and the Funambol open-source synchronization server. A client application interacts mainly with two entities of the Funambol Client API: the `SyncManager` and the `SyncSource`. The `SyncManager` is in charge of hiding the complexity of the synchronization process, providing a simple interface to the client application. It handles communications, including the underlying protocol. The `SyncSource` represents the collection of items stored in the local repository. It represents both the client items to be sent to the server, and those obtained from the server. The client interacts with `SyncSource`, while the `SyncManager` performs the transparent synchronization.

Figure 9 shows a sequence diagram of the synchronization flow, where the entities that participate in the synchronization process have the following meaning. `ClientApplication` represents any program using the Funambol SyncML API (the client mobile application in DIMAG). A `SyncManager` represents the SyncML API synchronization engine, and `SyncSource` corresponds to the interface of the data source. `SyncListener` provides a mechanism to monitor the synchronization process and `SyncServer` symbolizes the SyncML server.

As shown in Figure 9, the synchronization process is made up of an initialization, the synchronization performed when data is modified, and the end of the synchronization session. The client application starts the synchronization process providing a `SyncSource` to be synchronized (`source`). Then, the `SyncManager` takes control of the synchronization process. Whenever an element in the `SynSource` is modified, either by the client or the server, the data is transparently synchronized. A further description of the synchronization process is described in [28].

6. Implementation

The dynamic generation and deployment of DIMAG applications consist of two phases: program analysis and code generation.

6.1. Program Analysis

The DIMAG-Root, DIMAG-UI+CSS and SCXML documents describing a DIMAG application are analyzed before generating the target code. For user interface and workflow documents, an Abstract Syntax Tree (AST) structure is built to guide the code generation phase.

A DIMAG application is commonly defined with multiple views comprising the application presentation layer. Therefore, the analysis of the DIMAG-UI documents creates an independent AST for each DIMAG-UI document. Afterwards, each AST is decorated with the style information provided by the corresponding CSS documents.

We have used the Apache Commons SCXML implementation to process the workflow documents. We have enhanced that tool with specific modules that process the extensions we added to the language. These modules are executed when a `dimag` element is processed by the tool, building the corresponding AST in one step. A single AST is generated for the whole application workflow [33].

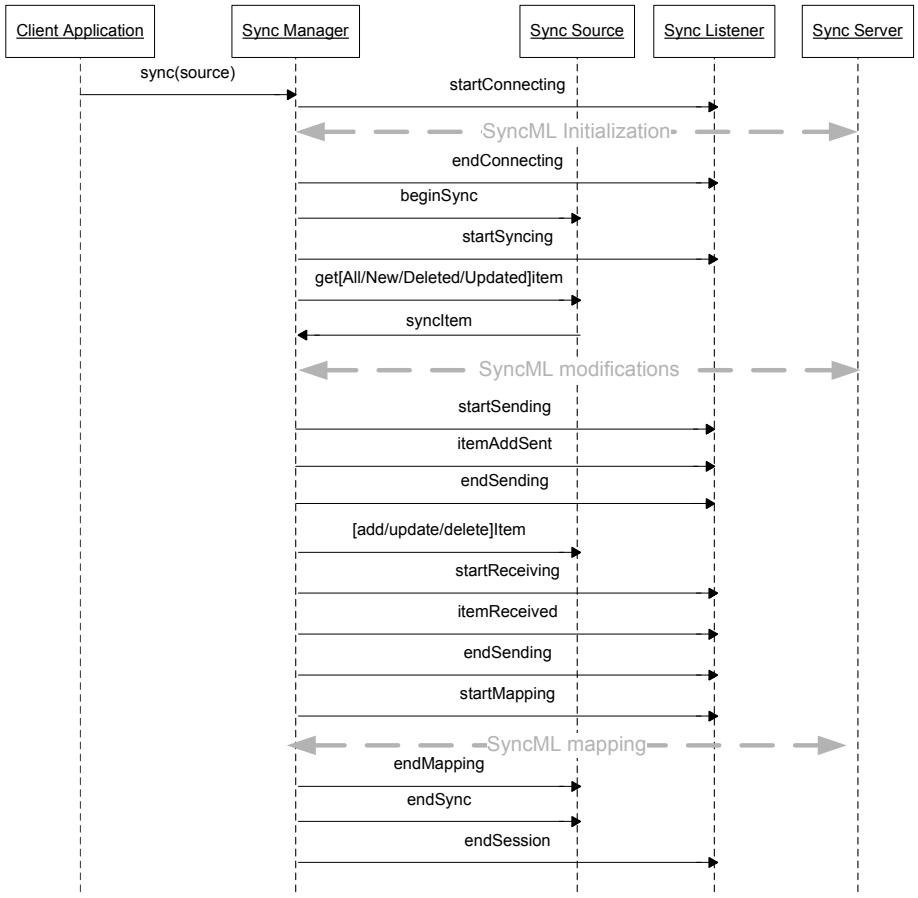


Figure 9: Synchronization flow in Funambol.

6.2. Code Generation

Once the AST has been created in the program analysis phase, DIMAG uses the class diagram shown in Figure 10 to generate application code for a specific platform. The classes involved in the code generation process are divided into four modules:

- `codegenerator`: This module writes the target code for a specific DIMAG application. In Figure 10, only the `CodeGenerator` class is shown, but this class is the root of another hierarchy shown in Figure 11. It holds the path where code is generated, a buffer for temporal storage of the class being generated, and a `write` method to generate the target code.
- `synchronizer`: This module performs the state synchronization with the server, as described in Section 5.3.3. Using SyncML, it synchronizes the data when there are changes in the dataset at the server or client side, considering the synchronization policies described in the application specification. If there is no connectivity, the mobile client displays a warning message and waits for the server acknowledgment to advance. This module is added to the generated application.

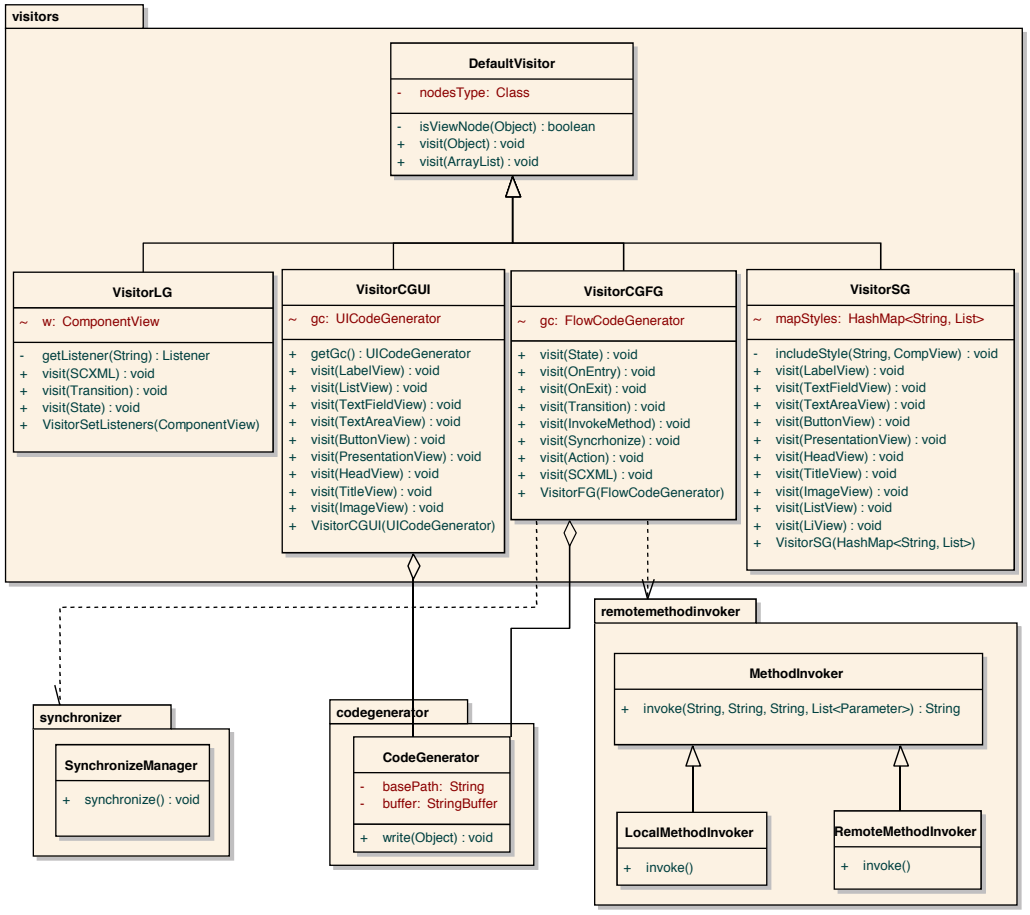


Figure 10: Classes used to dynamically generate DIMAG applications.

– `remotemethodinvoker`: This is the module devoted to perform local and remote method invocations. For remote invocations, the current implementation uses SOAP Web Services provided by the `RemoveMethodInvoker` class. In contrast, `LocalMethodInvoker` allows calling local methods implemented for a particular target platform. This module is added to the generated application.

– `visitors`: A module that includes different instances of the *Visitor* design pattern [34] to traverse the different ASTs representing a DIMAG application, calling to the code generator. The `DefaultVisitor` class guides the process of code generation, invoking the rest of the modules using reflection. The `VisitorCGUI` class generates user interface code and `VisitorCGFG` the application workflow; `VisitorSG` decorates the user interface AST with CSS annotations; and `VisitorLG` connects the application workflow with the application views, telling the interface elements what to do when an event is triggered. This module, together with the `codegenerator`, converts the ASTs into executable code.

Figure 11 details the internal structure of the `codegenerator` module. It has been designed following a component-oriented approach, so that the programmer can dynamically add new code generation modules, without restarting the framework [35]. Those new modules would provide code generation for software platforms that were not supported yet, following the *Parallel Hierarchies* design pattern described in [36].

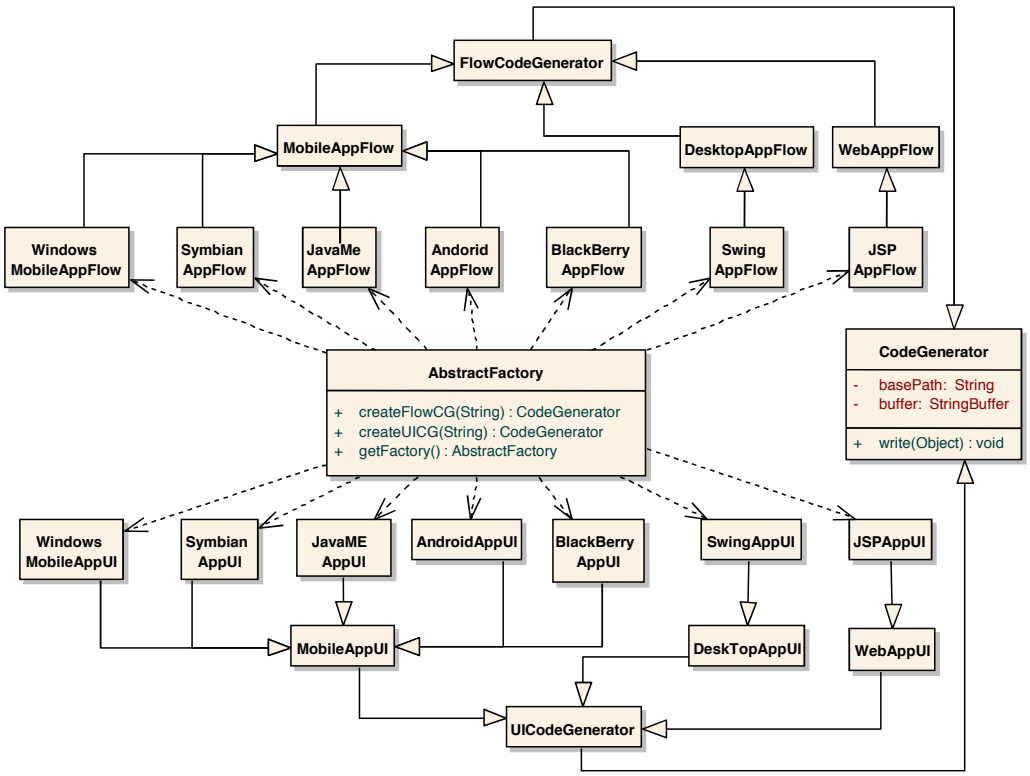


Figure 11: Class hierarchy of the `codegenerator` module.

When a mobile Web browser requests an application and the device is identified, the required classes for generating the application for that particular platform are dynamically loaded. The criterion used to name these classes is very important, because they are dynamically loaded considering a fixed naming convention. Class names are made up of the target platform name (e.g., `BlackBerry` or `Android`) plus the application element to be generated (e.g., `AppUI` or `AppFlow`) –see Figure 11. This naming convention is used to follow the *Convention over Configuration* (CoC) principle [37], facilitating the dynamic extension of the framework, and reducing the number of configuration files [37].

The current version of DIMAG generates code for Android, the .NET Compact Framework, and the MIDP Java mobile platform using Oracle's LWUIT for the user interface. LWUIT has been created to reduce user interface fragmentation in Java ME; the same UI can be defined for CLDC/MIDP and for CDC/PBP/PP. DIMAG generates (C# and Java) source 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 (e.g., the life cycle of a Midlet is quite different to that of a .NET application).

7. Related Work

7.1. Cross-Platform Mobile Development Tools

Many existing commercial cross-platform mobile development tools (XMTs) have appeared on the market over the last years. They commonly follow different approaches to facilitate application definition and code generation for a specific set of mobile target platforms. Focusing on the development tools for building native applications, the following paragraphs provide a brief description of the most popular XMTs [38].

Xamarin [39] provides two ports of the Microsoft's .NET Common Language Runtime (CLR) to Android and iOS, and an Integrated Development Environment (IDE) named Xamarin Studio. Xamarin provides imperative application creation for Android and iOS in C#. In order to create a multi-device application, a project needs to be started for each different platform (Android, iOS and, directly, on Windows Phone), with the possibility of sharing source code among projects. The C# classes required to program an application vary depending on the target platform. Therefore, the *Write Once, Run Anywhere* principle is not fully followed, although Xamarin's technology provides a single programming language to develop applications for three mobile platforms. Reusing code among platforms is managed by the developer in a manual way.

The Corona SDK [40] is a software tool that allows application definition in the imperative Lua programming language. Corona generates code for iOS and Android (and two other platforms which are actually specific Android-based eBook device families: Kindle and Nook).

Appcelerator Titanium [41] is an IDE that allows the generation of applications for Android and iOS. Recently, it has started providing support for generation of mobile Web applications. Application definition is carried out programmatically, in JavaScript.

Unity [42] is a visual IDE mainly focused on the creation of multi-platform video-games. It supports iOS and Android, in the mobile domain, but also for other desktop and video-game console platforms. Additionally, it provides the definition of generic applications using C#, JavaScript or Boo.

Finally, PhoneGap [43] permits single application definition by means of HTML, CSS and JavaScript. It executes the application by interpreting the definition files in a Web view embedded in a native application. Therefore, application view (UI content and styling) is defined in a declarative way, by means of HTML and CSS. On contrast, application model and controller are imperatively defined in JavaScript.

7.2. Existing uses of SCXML

Considering the relevance of the SCXML language in the definition of the DIMAG framework, we have analyzed how other works have effectively used this language to define application workflow. SCXML is the proposed language for Controller Document in W3C's Multimodal Architecture and Interfaces [44]. This recommendation is the technological basis for software products facilitating multimodal interaction between information services and users, such as Convergys Media Exchange with HomeZone [45] and Intelligent Voice Portal [46]. SCXML is used in the W3C Multimodal Architecture as the application's interaction manager. Related to voice-based applications, W3C is using SCXML as the language to handle dialog management in the next version of VoiceXML [47] (version 3.0).

Apache Commons SCXML [7], the language processor used in the DIMAG framework, is also used by other Apache projects, such as Reusable Dialog Components and Apache Shale. Outside the Apache community, the Software Demo [48] platform for online live testing of software uses SCXML to manage software/hardware infrastructure for managing resources, such as virtual machines, storage devices and virtual desktop components.

MyMobileWeb [16] is an open source, standards-based software framework that simplifies the rapid development of mobile Web applications and portals. MyMobileWeb uses SCXML to specify the control of mobile application flow generated by this framework.

8. Conclusions

Computer standards entail an important tool to develop new software products that integrate multiple existing technologies. DIMAG faces the complex problem of device fragmentation, proposing a framework to declaratively build connected mobile applications. We have divided this problem into simpler subproblems, and studied the suitability of existing standards to solve them. In DIMAG, applications are defined with three different specifications: workflow definition, user interface, and application information. Different standard languages have been chosen to define these modules. Mainly, IDEAL2 is used to define the user interface and SCXML to specify the application workflows. XQuery has been the selected language to declaratively specify the data to be synchronized between the server and the client. Software standards have also been used to implement DIMAG: W3C DDR Simple API for device identification, SyncML for data synchronization, and JSR 154, SOAP 1.1 and WS-I Basic Profile for client server communications.

The DIMAG framework shows the feasibility of creating a platform to open new opportunities in the research and development of connected mobile applications, based on the *Write Once, Run Anywhere* approach. It is meant to give support to many of the issues related to mobile application generation, considering the specifics of target mobile devices, the dynamic generation of applications, and the client upgrade when a new version of the application is released.

Future work will refine the different modules that make up the framework. The code generation process of mobile clients will be augmented, considering new specific controls of mobile devices. We will also take into account other additional factors regarding user interface adaptation, such as screen size and color depth. Integration with the context is another issue of interest for the authors of the DIMAG framework. Future versions of the platform will support transitions between states caused by events raised by the proximity of a wireless network, specific cells in a cellular network, or Bluetooth devices. Finally, after the involvement of the authors in the W3C MBUI working group, another future improvement of DIMAG is the alignment of user interface definition with the recommendations released by the W3C.

Acknowledgments

We would like to thank the anonymous reviewers for their 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.*

This work has also been supported by the Intermark and Telecable Companies, and the Government of the Principality of Asturias by means of its Science, Technology and Innovation Department.

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] Shapiro R. Process Definition Interface – XML Process Definition Language. Workflow Management Coalition Specification; 2008.

[5] Jordan D, Evdemon J. Web Services Business Process Execution Language. OASIS Standards; 2007.

[6] Barnett J. State Chart XML (SCXML): State Machine Notation for Control Abstraction. W3C Working Draft 16; 2012. http://www.w3.org/TR/scxml.

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

[8] Phanouriou C. UIML: A Device-Independent User Interface Markup Language. Virginia Polytechnic Institute and State University; 2000.

[9] Luyten K. UIML.Net: a UIML renderer for .NET; 2012. http://research.edm.uhasselt.be/kris/projects/uiml.net.

[10] jUIML. Open source implementation of UIML specification for JavaSE and JavaME; 2012. http://sourceforge.net/projects/juiml.

[11] Limbourg Q, Vanderdonckt J, Michotte B, Bouillon L, Lopez-Jaquero V. UsiXML: A Language Supporting Multi-path Development of User Interfaces. vol. 3425 of Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin / Heidelberg; 2005. p. 134–135.

[12] ISTI. The CAMELEON Project: plasticity of user interfaces; 2012. http://giove.isti.cnr.it/projects/cameleon.html.

[13] Michotte B, Vanderdonckt J. GrafiXML, a Multi-target User Interface Builder Based on UsiXML. In: Proceedings of the Fourth International Conference on Autonomic and Autonomous Systems. ICAS '08. Washington, DC, USA: IEEE Computer Society; 2008. p. 15–22.

[14] Trindade FM, Pimenta MS. RenderXML - a multi-platform software development tool. In: Proceedings of the 6th international conference on Task models and diagrams for user interface design. TAMODIA'07. Berlin, Heidelberg: Springer-Verlag; 2007. p. 293–298.

[15] Cantera JM, Diaz JL, Rodriguez C. IDEAL Core Language Working Draft 13; 2010. `http://files.morfeo-project.org/mymobileweb/public/specs/ideal2`.

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

[17] Smith K. Device Independent Authoring Language (DIAL), W3C Working Group Note 29; 2010. `http://www.w3.org/TR/dial`.

[18] Boag S, Chamberlin D, Fernandez MF, Florescu D, Robie J, Simeon J. XQuery 1.0: An XML Query Language, W3C Recommendation 14; 2010. `http://www.w3.org/TR/xquery`.

[19] Kay MH. SAXON, the XSLT and XQuery Processor; 2011. `http://saxon.sourceforge.net`.

[20] MXQuery. A lightweight, full-featured XQuery engine; 2012. `http://mxquery.org`.

[21] Simon R, Wegscheider F, Tolar K. Tool-supported single authoring for device independence and multimodality. In: Proceedings of the 7th International Conference on Human Computer Interaction with Mobile Devices & Services. MobileHCI '05. New York, NY, USA: ACM; 2005. p. 91–98.

[22] Mordani R. JSR 154: Java^{TM}Servlet 2.4 Specification; 2007. `http://jcp.org/en/jsr/detail?id=154`.

[23] 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`.

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

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

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

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

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

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

[30] 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. .

[31] Evolus. Pencil Project; An open-source GUI prototyping tool that's available for all platforms; 2013. `http://pencil.evolus.vn`.

[32] Axure. Arure RP; make interactive HTML prototypes of websites & apps; 2013. `http://www.axure.com`.

[33] 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.

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

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

[36] 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.

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

[38] Mobile V. Cross-platform development tools 2012; bridging the worlds of mobile apps and the Web; 2013. `http://www.visionmobile.com/product/cross-platform-developer-tools-2012`.

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

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

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

[42] Unity. Unity 3D: mobile deployment free.; 2013. `http://unity3d.com`.

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

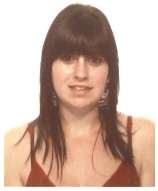
[44] Barnett J, Bodell M, Dahl D, Kliche I, Larson J, Porter B, et al.. Multimodal Architecture and Interfaces, W3C Recommendation 25; 2012. `http://www.w3.org/TR/mmi-arch`.

[45] Convergys. Media Exchange for Home Zone; 2013. `http://www.convergys.com/products/media-exchange-home-zone/index.php`.

[46] Convergys. Intelligent Voice Portal; 2013. `http://www.convergys.com/products/contact-center-software/intervoice-voice-portal.php`.

[47] McGlashan S, Burnett DC, Akolkar R, Auburn R, Baggia P, Barnett J, et al.. Voice Extensible Markup Language (VoiceXML) 3.0, W3C Working Draft 16; 2010. `http://www.w3.org/TR/voicexml30`.

[48] SoftwareDEMO. SoftwareDEMO, Boost your sofware sales; 2013. `http://www.softwaredemo.com`.



Patricia Miravet (Oviedo, Spain, 1984). BS in Computer Science from the Technical School of Computer Science of the University of Oviedo (2006). MS in Web Engineering from the Technical School of Computer Science of the University of Oviedo (2009). PhD student developing a thesis entitled "DIMAG: Device-Independent Mobile Application Generation Framework". She works as a researcher at the unit of Device Independence and Mobility of the R&D Department of CTIC Foundation since 2007. Her main research interests are device independence, declarative approaches for application definition, and ubiquitous and mobile computing. She has participated in several national projects funded by the Governments of Spain and the Principality of Asturias.

Francisco Ortin (Oviedo, Spain, 1973). Computer Scientist from the Technical School of Computer Science of the University of Oviedo (1994). Computer Engineer from the Technical School of Computer Science of Gijon (1997). PhD from Computer Science Department of the University of Oviedo (2002) with a thesis entitled "A Flexible Programming Computational System Developed over a Non-Restrictive Reflective Abstract Machine". Working as an Associate Professor at the Computer Science Department of the University of Oviedo, his main research interests are computational reflection, dynamic languages, object-oriented abstract machines, meta-level systems and meta-object protocols, and aspect-oriented software development. He can be reached at `http://www.di.uniovi.es/~ortin`.

Ignacio Marin (Gijon, Spain, 1973). BS in Computer Science from the Technical School of Computer Science of the University of Oviedo (1995). MS in Computer Engineering from the Technical School of Computer Science of the University of Oviedo (1999). PhD student developing a thesis entitled "Optimization of Interactive Multimedia Services through Server Configuration and Tuning". He works as the Head of the unit of Device Independence and Mobility of the R&D Department of CTIC Foundation since 2005. His research interests include device independence, multimedia systems, and ubiquitous and mobile computing. He has participated in several projects funded by the Government of Spain and the European Union.

Abel Rionda (Gijon, Spain, 1981). BS in Computer Science from the Technical School of Computer Science of the University of Oviedo (2003). MS in Computer Engineering from the Technical School of Computer Science of the University of Oviedo (2006). PhD student developing a thesis about adaptation mechanisms in device-independent mobile Web applications, making use of context information. Working as a researcher at the unit of Device Independence and Mobility of the R&D Department of CTIC Foundation since 2005, his research interests are service and application adaptation to the delivery context, and ubiquitous and mobile computing. He has participated in several projects funded by the Government of Spain and the European Union.