A Case Study in DSL Re-targetability

Master’s Thesis

Nami Nasserazad
A Case Study in DSL Re-targetability

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Nami Nasserazad
born in Tehran, Iran

TU Delft
Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl
A Case Study in DSL Re-targetability

Author: Nami Nasserazad
Student id: 4039416
Email: n.nasserazad@student.tudelft.nl

Abstract

Domain-specific language is a computer programming language of limited expressiveness focused on a particular domain [23]. By focusing on domain concepts, there is a little need to care about non-essential details during software development process. One non-essential aspect that developers are mostly willing to dispose of is a target platform on which the application is to be executed. Ideally, the task of porting a software system to different platforms should be completely hidden from the developer's point of view. Consequently, the DSL compiler is responsible for generating codes in different target languages that are compatible with different platforms. This problem is known as DSL re-targetability. The goal of this thesis is to explore DSL re-targetability challenges to find out how the characteristics of a DSL associated with the target platform can determine the degree of portability. To achieve this goal, mobl [5] (a DSL for mobile web applications) has been chosen as a source DSL. As a new target platform for mobl, iOS native applications especially for iPad have been selected. Afterwards, a compiler to transform mobl to Objective-C (which is Apple recommended programming language to develop iOS applications) has been designed and implemented. Now, a great number of applications written in mobl for web backend can be compiled to Objective-C backend with minimal changes in their source code. The resulting iOS applications meet the iPad applications standards suggested by Apple. However, to have real iPad style applications, some adjustments in the source code are recommended.

Thesis Committee:

Chair: Dr. E. Visser, Faculty EEMCS, TU Delft
University supervisor: Drs. Zef Hemel, Faculty EEMCS, TU Delft
Committee Member: Dr. M. Pinzger and Dr. J. Hidders Faculty EEMCS, TU Delft
First of all, I would like to thank my supervisor, Eelco Visser, for suggesting me this interesting project in which I could realize my ideas. Also I would like to thank Zef Hemel, who created mobl, for his kind help and guidance during preparing my thesis. Moreover, I would like to thank my great friends: Katayoun, Siamak, Armin, Pejman, Nadjla and Siavash who make Delft like my hometown. Finally, a big thanks to my brother and my parents especially my mother who has supported me in all stages of my life.

Nami Nasserazad
Delft, the Netherlands
September 20, 2011
Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1
   1.1 Terminology and Basic Concepts 1
   1.2 MDSD for Re-targetability or Re-targetability for MDSD? 4
   1.3 Research Questions 5
   1.4 DSL Re-targetability Case Study: Tablets 6

2 DSL Re-targetability: State of the Art 9
   2.1 User Interface Markup Language (UIML) 9
   2.2 eXtensible Interface Markup Language (XIML) 11
   2.3 eXtensible Application Markup Language (XAML) 11
   2.4 MyMobileWeb 13
   2.5 DIMAG 14
   2.6 XMLVM 18
   2.7 Applause 21
   2.8 MobDSL 22
   2.9 PIL 23

3 mobl: A DSL of the Mobile Web 25
   3.1 Technology 25
   3.2 Code Generation Phases 25
   3.3 Structure 26

4 iOS Native Applications as a New Target for mobl 47
   4.1 iOS Architecture 48
   4.2 iOS: a New Target for mobl 52
## CONTENTS

5 Evaluation 83  
  5.1 Shopping List . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 83  
  5.2 A Comparison between Different Approaches . . . . . . . . . . . . . . . 92  

6 Conclusions and Future Work 95  
  6.1 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95  
  6.2 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95  
  6.3 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98  

Bibliography 101  

A Glossary 105  

B Shopping List Source Code 109
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>UIML sample user interface declaration</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>XAML sample user interface declaration</td>
<td>12</td>
</tr>
<tr>
<td>2.3</td>
<td>XAML code behind</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>IDEAL user interface specification</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>IDEAL CSS specification</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>IDEAL final layout</td>
<td>15</td>
</tr>
<tr>
<td>2.7</td>
<td>SCXML sample workflow declaration</td>
<td>17</td>
</tr>
<tr>
<td>2.8</td>
<td>DIMAG-ui user interface declaration</td>
<td>18</td>
</tr>
<tr>
<td>2.9</td>
<td>A method to calculate factorial in Java</td>
<td>19</td>
</tr>
<tr>
<td>2.10</td>
<td>The XML that is generated using XMLVM tool</td>
<td>20</td>
</tr>
<tr>
<td>2.11</td>
<td>Cocoa UIButton wrapped by Button class</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>A sample screen definition in mobl.</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>the output of the screen declaration</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>A sample control definition in mobl.</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>The output of the control declaration</td>
<td>30</td>
</tr>
<tr>
<td>3.5</td>
<td>A sample HTML control definition in mobl.</td>
<td>30</td>
</tr>
<tr>
<td>3.6</td>
<td>The output of the HTML control declaration</td>
<td>31</td>
</tr>
<tr>
<td>3.7</td>
<td>A sample of using elements in mobl.</td>
<td>32</td>
</tr>
<tr>
<td>3.8</td>
<td>The output of the control built using &quot;elements&quot;</td>
<td>32</td>
</tr>
<tr>
<td>3.9</td>
<td>A sample of a nested control declaration</td>
<td>33</td>
</tr>
<tr>
<td>3.10</td>
<td>The output of the nested control declaration</td>
<td>33</td>
</tr>
<tr>
<td>3.11</td>
<td>A sample of data binding declaration for an HTML control</td>
<td>34</td>
</tr>
<tr>
<td>3.12</td>
<td>The output of the HTML control data binding declaration</td>
<td>35</td>
</tr>
<tr>
<td>3.13</td>
<td>A sample of &quot;when-else&quot; usage in mobl.</td>
<td>36</td>
</tr>
<tr>
<td>3.14</td>
<td>The output of the &quot;when-else&quot; clause.</td>
<td>36</td>
</tr>
<tr>
<td>3.15</td>
<td>A sample of &quot;list&quot; usage in mobl.</td>
<td>37</td>
</tr>
<tr>
<td>3.16</td>
<td>The output of the &quot;list&quot; clause.</td>
<td>38</td>
</tr>
<tr>
<td>3.17</td>
<td>A control variable sample</td>
<td>39</td>
</tr>
<tr>
<td>3.18</td>
<td>The control variable sample output.</td>
<td>40</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

3.19 A sample of "style" usage in mobl. ........................................... 40
3.20 The resulting output with its customized style. .......................... 41
3.21 A sample of function declaration in mobl. ................................. 41
3.22 A sample of "type" declaration in mobl. .................................... 42
3.23 A sample of built-in generic type declaration. ............................ 43
3.24 A sample of entity declaration and entity instantiation. ................. 44
3.25 A sample of a collection instantiation for an entity. ..................... 44
3.26 Two programming styles in mobl in order to use data retrieval APIs. . 45

4.1 iOS architecture overview [4]. .................................................. 48
4.2 A UITableView sample. .......................................................... 49
4.3 A UITabBarController sample. .................................................. 50
4.4 A UISplitViewController sample. ............................................. 51
4.5 The domain model of mobl-ios. .............................................. 53
4.6 mobl to Objective-C control transformation. .............................. 56
4.7 iOS default button. .............................................................. 57
4.8 iOS standard button. ............................................................. 57
4.9 A group sample in mobl. ....................................................... 58
4.10 The web mobl output of group control ...................................... 59
4.11 The iOS mobl output of group control. .................................... 60
4.12 The iOS default split view (bottom image) vs. mobl-ios split view (top image). 61
4.13 A custom control sample. ..................................................... 62
4.14 A sample usage of the control declared in figure 4.13 .................... 63
4.15 A modal view sample code. ................................................... 64
4.16 The output of the code shown in figure 4.15 ............................. 65
4.17 Scope class diagram. ........................................................... 66
4.18 A callback sample. ............................................................. 66
4.19 A sample of using callback in a loop. .................................... 67
4.20 The iOS output of the code shown in figure 4.19 ......................... 68
4.21 A derived variable sample. ................................................... 70
4.22 The iOS output of the code shown in figure 4.21 ......................... 71
4.23 A control data binding sample. ............................................. 71
4.24 The iOS output of the code shown in figure 4.23 ......................... 72
4.25 The iOS output of the code shown in figure 3.13 ......................... 73
4.26 The iOS output of the code shown in figure 3.15 ......................... 74
4.27 The iOS output of the code shown in figure 3.17 ......................... 75
4.28 The mobl-ios styling concepts. ............................................ 76
4.29 Objective-C code generated for Person entity declared in figure 3.24. 81

5.1 The web (top image) and iOS (bottom image) layouts of the root screen of the Shopping list application. ........................................ 84
5.2 The web (top image) and iOS (bottom image) layouts when the selected category is changed. .................................................. 85

viii
5.3 The web (top image) and iOS (bottom image) layouts for searching through the
shopping items of the selected category. ........................................ 87
5.4 The web (top image) and iOS (bottom image) layout for searching through all
shopping items. ................................................................. 88
5.5 The web (top image) and iOS (bottom image) layouts to create a new shopping
item. ............................................................... 89
5.6 Using <h2> HTML tag in mobl source code. .......................... 90
5.7 The web mobl context menu. ............................................. 91
5.8 Android context menu. .................................................. 92
5.9 Master part with section and ”Add Item” screen as dialog. ............ 93
6.1 UITableView in delete mode. ........................................... 97
Chapter 1

Introduction

Nowadays, software production market is extremely competitive and companies are looking for innovative approaches to speed-up their productivities. Hiding the irrelevant details of software systems and focusing on problems can reduce the amount of repetitive tasks and thus speed-up the software development process. Model-driven software development (MDSD) by employing domain-specific languages (DSL) is a promising solution to increase productivity of companies.

On the other hand, ubiquitous computing that its idea was first proposed in [44] is spreading out rapidly. Computation is going to be embedded in a large variety of devices such as smart phones, tablets etc. Users are expecting their desktop applications to be executed on their portable devices. Each of these devices has its own characteristics such as screen size, I/O device, CPU and storage limitation etc. Several adaptations are needed for an application to become useful on these devices and work properly. Thus, nowadays, re-targetability of a software system plays a prominent role in the software development process.

Combining above concepts results in another concept called DSL re-targetability which is the topic of this thesis. The aim of DSL re-targetability is to bring the capability of platform portability for applications which are developed by DSLs. In this chapter, first, the basic concepts related to DSL re-targetability will be described. Afterwards, main research questions of this thesis will be introduced. A brief description about the case study of this thesis will come next. Finally, the scheme of the remaining chapters will be presented.

1.1 Terminology and Basic Concepts

1.1.1 Model-driven engineering

Model-driven engineering (MDE) is a software engineering approach that aims to improve software development process by employing high-level, domain-specific models in every cycle of a software system development such as implementation, integration, maintenance, and testing [21]. The models used to describe the system should be automatically transformed to implementation. The main goal of using models in this approach is to provide a higher level of abstraction by hiding irrelevant details.
1. Introduction

1.1.2 Model

Different definitions of model have been proposed in the literatures. According to Kuehne definition of model in [29], "A model is an abstraction of a (real or language based) system allowing predictions or inferences to be made". Considering the terms used in this definition, although a model should describe a system in a higher level of abstraction, this description should not result in vagueness. It means that by using model, a correct overview about the final system should be achieved and resulting software system should meet what has been expected from the initial model. Based on OMG definition, a model of a system is "a description or specification of that system and its environment for some certain purpose" [33]. As can be seen, unlike Kuehne definition, the term of abstraction is not used in this definition and this denotes that models can be specified in different levels of abstraction. Thus, some models can be more abstract and some others can be more concrete to cover more details. Consequently, the model-driven development can be done in multiple phases and resulting models of those phases can be transformed to each other.

1.1.3 Domain and Domain Model

Models can be specialized for a family of systems with considerable set of commonalities. Idiomatically, the general subject to which these systems belong is called domain. The models that specialized for a domain called domain-specific models (DSMs). The basic constructors of a DSM can be defined based on the basic concepts of its associated domain. For example, if we consider database as a domain, table, tuple, relation, constraints etc. become the essential components of this domain. A model to describe a database system can use these components as its basic constructors.

1.1.4 Domain-specific Language

Domain-specific models can be expressed or more precisely realized by domain-specific languages (DSLs). In the other word, DSLs make DSMs executable by computers. Based on the definition proposed by Martin Fowler in [23], "DSL is a computer programming language of limited expressiveness focused on a particular domain". Several important key elements can be deduced from this definition. Firstly, DSL is a programming language. It means that although it has a structure that makes it easy for humans to understand, it still should be executable by computers. Secondly, DSL has limited expressiveness. Obviously, a DSL supports only a set of features needed for its domain and it does not support lots of capabilities that we often expect from a general purpose language (GPLs). The third key point of Fowler’s definition of DSL is the domain focus. DSL is built using the high level concepts of its associated domain and other aspects of the system should be hidden from the developer point of view. On the one hand, by being close to high level concepts of its domain, the DSL will be easier to understand by humans and on the other hand, by hiding irrelevant details, it will help developers only to think about the domain problems and not about the different software development aspects.

2
1.1.5 Target Platform and Re-targetability

One irrelevant detail of which developers are willing to dispose is the software multi-targeting. Ideally, a developer expects that the software system developed by her automatically has the capability of execution on different platforms. Compilers can make this process automatic by having the ability of generating code for multiple platforms. Generally, an attribute of compilers to generate code for more than one platform is called re-targetability.

Based on the definition proposed in [18], a platform is a set of subsystems and technologies that provide the capabilities needed to support the execution of software. Basically, a platform presents the device and operating system on which the software is executed. For example, 64-bit Windows operating system running on Intel processor can be considered as a target platform. The source code of a software system, in order to be executed on this platform, should be compiled to the machine architecture specific instructions of this platform. Clearly, developing a special compiler per platform is expensive and makes the portability of the software systems difficult. Therefore, building the platform-independent software systems has always been taken into consideration by many general purpose languages. Using intermediate languages is one common approach to achieve the platform independency. For example, the Java compiler generates bytecode instead of machine code. Java bytecode is an abstraction of machine languages in order to provide a unique set of instructions for various machine architectures. Java bytecode is executed by Java Virtual Machine (JVM). JVM maps bytecode to machine instructions based on the platform on which JVM is running. JVMs are available for many hardware and software platforms and consequently, a Java application can be easily executed on those platforms. As can be seen, in Java programming language, re-targetability is handled at the virtual machine level instead of the compiler level and the Java compilers are only responsible for generating bytecode.

In model-driven engineering context, the researches are focused on DSLs and re-targetability is examined for the DSLs compilers. Rather than generating the executable machine code or low-level intermediate source code, the DSL compilers typically generate source code in high level languages such as Java, C# and Python [25]. Obviously, generating source code in a high level language is far easier than generating machine instructions and this makes the DSL compilers easier to produce. Moreover, a DSL compiler may benefit the abstraction level of its destination language. For example, if Java is chosen as the target language, the DSL automatically becomes platform-independent because Java itself is platform-independent.

By moving from machine instructions to high level source code, DSLs re-targetability challenges switch to software platforms. Generally, according to the context, the ingredients of a platform can be adapted. For example, in the user interface design, the set of widgets that operating system provides can also be considered as a part of target platform [19]. Similarly, in the DSL context, platform is abstracted to the software platform. A software platform consists of one or more programming languages with a set of libraries and frameworks, deployable on one or more operating systems [25]. For example, Microsoft .Net is a software platform. .Net framework and its libraries beside all the supported programming languages like C#, VB, J# etc. construct this software platform.

According to these concepts, DSL re-targetability can be formulated as follows:
1. INTRODUCTION

**Definition 1** DSL re-targetability is the capability of DSL compiler to generate source code for different software platforms.

Imagine a DSL for developing Android mobile native applications. The Android application framework and Java programming languages are the elements of the Android platform. In this case, the application written in the DSL should be compiled to a source code in Java programming language based on Android application framework. If we decide to use the software developed by this DSL on iPhone or iPad, then we have to define iOS\(^1\) as a new target platform of our DSL. Therefore, the DSL compiler should have the ability of generating both Java source code and Objective-C source code for the applications programmed in the DSL.

1.2 MDSD for Re-targetability or Re-targetability for MDSD?

MDSD and re-targetability are two interrelated topics. On the one hand, in many contexts, MDSD has been proposed as a promising solution for re-targetability [19, 22, 38, 40, 34]. These solutions try to hide the details of the target platforms by introducing a set of abstract constructors. For example, in the model-based user interface design, most of approaches (like UIML [19], XForms [42], XIML [39], and UsiXML [30]) provide a set of abstract user interface widgets to produce the abstract presentation model (APM). These abstract widgets will be mapped to the concrete widget set of destination targets. It means that in these contexts, the goal is re-targetability and an approach to achieve this goal is MDSD. In these approaches, first, various target platforms are studied and based on the commonalities of them, a set of abstract constructors are presented which will be used as the basic constructors of the abstract modeling.

On the other hand, many existing DSLs are willing to support multiple platforms. For example, Hibernate Query Language (HQL) (which is an object oriented DSL for database queries) provides multiple back-ends for different database management systems. The direction of the approach used in HQL differs from what has been used in the model-driven user interface design. In the HQL case, first HQL as a DSL has been proposed to provide the higher level capabilities to facilitate database querying. Afterwards, the multi-platform support has been added to this DSL to make it compatible with various database management systems. However, in the model-based user interface design approaches, supporting multiple targets has been introduced as a goal and UI DSLs have been employed to achieve this goal. The focus of this thesis is re-targetability for DSLs which means that we are going to evaluate the challenges of adding a new target to an existing DSL.

In the next section, these challenges will be enumerated and according to them, some research questions will be formulated.

\(^1\)iPod, iPhone, and iPad operating system
1.3 Research Questions

As mentioned before, the main direction of this thesis is to explore the DSL re-targetability challenges. Mapping the constructors of a source DSL to a destination GPL may have several difficulties. Each programming language, either domain-specific or general purpose, provides its own programming style. The programming language style can be either at the syntactic level or at the semantic level (the computation model of the language). The closer source and destination programming language in style, the easier to transform. For example, many programming languages (especially DSLs) are functional. Thus, they may realize different concepts of functional programming style such as global functions, closures etc. If the destination language does not support these concepts, the compiler developer should emulate them in an efficient way and obviously this needs some efforts.

In addition to the language style, most of DSLs, because of their high level nature, declaratively support very complicated concepts. These concepts should be translated to a target language which mostly has imperative style. As a simple example, in XSLT which is a functional DSL to transform a XML to HTML, equal sign (=) has a complex high level meaning. Based on the definition, “If both object to be compared are node-sets, then the comparison will be true if and only if there is a node in the first node-set and a node in the second node-set such that the results of performing the comparison on the string-values of the two nodes is true”. As can be seen, in XSLT, equal sign is a declarative syntactical representation of XML node sets comparison. CSS is an excellent example to demonstrate these complexities. CSS is highly declarative and therefore, it is tricky to understand and also to realize. As mentioned in [23], in CSS, instead of “do this, then do that” structure, the interaction designer simply declares matching rules for HTML elements. CSS highly declarative nature makes it difficult to realize and this is the reason why still some browsers such as Internet Explorer cannot render it properly.

Basically, one of the goals of DSL design is to go from an imperative structure to a declarative structure. The DSL compiler should pass this way backwards. Obviously, if the DSL is more declarative, the transformation of that to an imperative model needs more efforts.

In DSL re-targetability context, although the above challenges still have their own impacts, sometime they get a new dimension. In a DSL design process, sometimes, first, a target language is assumed and several constructors of the DSL are built based on them. The tendency to one target language in DSL design process is mostly common in the user interface abstract languages. For example, XAML, which is Microsoft representation for user interface design, is XML based and inspired by HTML. Obviously because of close characteristics of XAML and HTML, supporting HTML back-end for XAML is easier than supporting Windows form back-end for that. In these cases, the DSL is not abstract enough for efficient re-targeting. To solve this deficiency, several approaches like what proposed in [32] have been introduced.

Based on above discussion, the research questions of this thesis can be formulated as follows:

- What are the main challenges and difficulties in DSL re-targetability?
1. INTRODUCTION

- Which characteristics of DSL do have influences over its degree of portability?
- What are the techniques to reduce the efforts needed to map the DSL constructors to the target language constructors?

The focus of above topics in this thesis is mostly at software platform level. However there are a lot of rooms to research at device platform level. Device attributes like screen size, I/O device etc. and operating system attributes like garbage collecting, multi-tasking etc. may have great influences on retargeting a DSL. Porting software to a new platform with different properties brings the need to adaptation. For example, imagine a hotel reservation system having both web version and iPhone application. The user expectations of these two versions are not the same. Thus, if there is a DSL for developing ticket reservation systems, in order to be portable between different platforms, a set of facilities are needed to make the resulting applications adaptive. For example, in tablets, because of the smaller size of the screen, many pages should be divided into multiple screens. Additionally, to facilitate form fillings for mobile users, some information can be retrieved from the context such as the user name or even the account number. The software adaptation can be imagined even in one single platform. For example, iPhone and iPad provide re-orientation in their screens. A user friendly application running on these devices should appropriately adapts itself with current orientation. These adaptations can be declared at the design level using DSL constructors or can be done automatically at the code generation level. Above subjects can be investigated under the following research question:

- What are the principles of platform adaptive code generation?

However, this topic is not in the scope of this thesis and can be researched as the future work.

1.4 DSL Re-targetability Case Study: Tablets

Ubiquitous computing is one of the most important topics in software industry in recent years. The term of “ubiquitous computing”, first has been used by Weiser in 1993 in [44]. Ubiquitous computing implies on embedding computation in many different kinds of devices which are greatly used in our daily life. Smart phones, tablets, pocket PCs etc. have become largely popular and they play an important role in our life. On the other hand, the user expectations of these devices are growing vastly. Typically, users expect their tablets to have the capability of executing all the applications of their desktop computers. Consequently, the software vendors to satisfy their customers should publish different versions of their products compatible with different devices. For example, Skype users, in addition to their desktops, are willing to use it on their tablets. Based on this need, iPhone, iPad and Android versions of Skype have been developed. Differences in software platforms used in

\[\text{For simplicity, the term of tablet is used for any kind of portable computers such as smart phones, pocket PCs etc.}\]
devices on the one hand and differences in the device attributes such as screen size, quality of the device accessories etc. on the other hand force the vendors to produce special software per platform which is clearly inefficient.

From another perspective, developing a mobile application itself can be difficult and time consuming. Although the SDK provided by different tablet brands like iOS SDK or Android SDK considerably simplify development process, still there are much rooms for abstraction. Different capabilities of a tablet should be usable through high level, declarative environment to facilitate software development process in this competitive market. Using document reader, using camera, using GPS, persisting and retrieving local data, calling services, sending Bluetooth etc. are those environment specific capabilities which should be available for developers easily.

In order to increase the level of abstraction, DSLs can be employed. Additionally, an appropriate DSL for developing mobile applications can facilitate multi-targeting by providing a unique syntactic and semantic interface for all platforms.

The popularity of mobile applications on the one side and the need for re-targetability to be compatible with different platforms on the other side encourage us to choose a DSL for mobile application development as our DSL re-targetability case study. Therefore, "mobl" [5] which is a functional DSL to create mobile web applications have been chosen. The target language of mobl is HTML5, JavaScript and CSS. Although the web applications developed using HTML5 can be portable to different platforms, the users of specific tablet mostly prefer to use applications that follow application development standards and guidelines of that specific tablet brand. For example, iOS users get used to interact with iOS widgets using their special swipe and tap behaviors. The style and the interaction way of the widgets on another platform like Windows Phone are quite different and the customers of each brand have bias to switch between them. Consequently, the mobile web applications cannot satisfy users completely. However, it is possible to provide different mobl control libraries for different platforms. These special libraries can look like platform widgets but these libraries should evolve by platform widgets evolution. For example, Kindle Cloud web application is an Amazon product to allow users to buy books directly from Amazon and not from Apple App Store. This web application has been built using HTML5 and it is quite similar to an iOS native application.

Although making HTML5 widgets similar to iOS native widgets is possible, but still developers prefer to use original widgets of a platform. Moreover, establishing connection to the internet is not always possible and this is one of the disadvantages of the web approach. Additionally, the web-browsers on the mobile devices have some limitations which can lead to inconsistencies in application functionality between different platforms. Lack of APIs for different device components is one of these deficiencies [28]. As a result, we have decided to implement an iOS backend for mobl especially for iPad. To achieve this goal, a mobl to Objective-C compiler have been developed. This experiment helps us to find out concrete answers for our research questions introduced in the previous section. With this approach, the back-ends for other platforms like Android or Windows Phone can also be produced.

The remaining of this thesis is organized into five chapters. The next chapter will be dedicated to related works in DSL re-targetability. mobl structure will be explained in
1. **Introduction**

Chapter three. Chapter four will explain the whole process of building the Objective-C back-end of mobl. Different technical and conceptual challenges with which we have dealt in this process in addition to our proposed solutions for them will be described there. The result of our work will be evaluated in Chapter five. In this chapter, the portability of a sample application from web mobl to Objective-C mobl will be evaluated. Conclusions and future work will come in the last chapter.
Chapter 2

DSL Re-targetability: State of the Art

In this chapter, the related works done in DSL re-targetability are briefly described. Additionally, we try to enumerate the advantages and disadvantages of them.

2.1 User Interface Markup Language (UIML)

UIML [17, 37] is a declarative XML-based language for the user interface specification. The main goal of UIML is to make cross-platform user interface design more efficient by highlighting the common features of different platforms user interfaces. A user interface defined in UIML is formed by a set of widgets which can be beautified using style attributes. These widgets can be either generic or platform-specific. When a widget is expressed in a generic term, it means that it is common among different platforms and in fact, it is an abstract interface object (AIO). Otherwise, when a widget is specified for a special target, it means that only in that target the widget should be realized. These kinds of widgets are called concrete interface objects (CIO). This process is the same in the beautification phase. It is possible that for different platforms, the widgets are styled separately. Furthermore, multiple user interfaces can be declared for even one single target platform. The user interface can be adapted to each of these user interfaces according to the current context. Generally, a UIML user interface declaration consists of four following parts:

- Structure: A hierarchical structure of the user interface elements (parts) either abstract or concrete is declared in this part. Widgets can be defined conditionally based on the target platform. The target platform is indicated using some prefix in the name of the widgets.

- Style: As mentioned earlier, the elements of this part specify the graphical properties of the user interface like size, position, color etc.

- Content: Providing a clean separation between the user interface content and the user interface structure is the main goal of this part. UIML supports different types of
2. DSL Re-targetability: State of the Art

```xml
<template name="CreditCard">
  <part name="CreditContainer" class="CreditDialog">
    <style>
      <property name="title">Credit Card Entry Form</property>
    </style>
    <part name="CreditNum" class="number" />
    <part name="AcceptNum" class="Accept" >
      <style>
        <property name="content">Accept</property>
      </style>
    </part>
  </part>
</template>
```

Figure 2.1: UIML sample user interface declaration

content such as text, sound, image etc. The content section can be useful for resource management especially for multi-lingual user interface design.

- **Behavior**: The connection between the user interface and the rest of the application is declared in this part. This part comprises a set of conditions and associated actions to change the value of the properties, to call APIs, to fire events, and to restructure the user interface.

The new version of UIML has some additional components to improve the modularity of the language. These components can be found in [10]. Figure 2.1 which is from [16] shows a sample user interface specified using UIML.

In the rendering process, first, using model to model transformations, all the generic structures and styles are transformed to platform-specific ones based on the target platform. Then the actual user interface is created according to the resulting model. Transformation-based Integrated Development Environment (TIDE) [29] is a tool developed to map the UIML specification to the actual user interface. This transformation has been done in multiple phases. Based on the commonalities of the target platforms, they are categorized in families. First, the user interface model is mapped to a more concrete model belonging to a special family. Afterwards, in the next phase, the actual user interface will be created. Dividing the transformation into multiple phases increases the extensibility of the transformer.

UIML declaratively supports re-targetability. Additionally, without any need to platform-specific widgets, the cross-platform user interfaces can be created using only generic widgets. However, the resulting user interface may not support the target platform standards. One UIML weak point is that collecting the generic and platform-specific parts of the user interface in one document makes the specification unreadable. On the other word, UIML does not separate the user interface specification from its rendering information.
tionally, UIML does not address context data. Binding the user interface elements to data objects cannot be clearly specified.

2.2 eXtensible Interface Markup Language (XIML)

XIML [39, 37] is a declarative XML-based DSL to describe the user interface. Representing a complete user interface ontology is the main aim of XIML. The user interface elements in XIML are categorized into a set of components. Task, domain, user, presentation and dialog are five predefined components in XIML. However, the designer is free to define his own components to provide a new perspective to the user interface.

Relation is the main element in XIML to generate the design knowledge about the user interface. According to definition proposed in [37], relation is a statement or definition which relates two or more elements. For example, ”Name attribute of person domain object is shown by text field A" is a relation which relates a domain element to a presentation element. As can be seen, using the domain component and relations, XIML has solved the user interface data binding problem which is not addressed in UIML. Using the semantics provided by XIML vocabulary, rich user interfaces can be described.

The user interface elements described in the presentation components can be declared in different abstraction level. In order to solve the re-targetability issue, XIML provides an intermediate presentation component consisting of a set of AIOs. Additionally, corresponding to different platforms, different collections of CIOs are also provided. AIOs are mapped to CIOs using relations. This means that the designer has acceptable degree of freedom to customize the retargeting strategies. Furthermore, XIML provides a set of semantics by which the user interface can be adapted to the current context. The target platform characteristics (e.g. screen size, I/O device etc.) shape the context properties.

Due to extensibility and flexibility, XIML is a good candidate to be a universal language for the user interface specification. However, XIML is only a specification and no systematic approach has been introduced in order to understand and realize the semantics provided by XIML specification. Some sample XIML user interface specification can be found in [12].

2.3 eXtensible Application Markup Language (XAML)

XAML [11] is a declarative markup DSL proposed by Microsoft to lay out the user interfaces. XAML is a part of Windows Presentation Foundation (WPF) and the main goal of that is to separate the UI definition from the run-time logics [11]. The user interface behaviors implemented in code-behind files are joined to markup through partial classes.

XAML has direct tie to Common Language Runtime (CLR) classes. Like Java Virtual Machine (JVM), CLR is .Net framework special runtime environment to execute the intermediate code generated by the compiler. Using this capability of XAML, the applications developed in different languages supported by .Net framework such as VB.NET or C# can share a single user interface specification. Additionally, Microsoft provides a set of tools to convert XAML specification to HTML or the other way around.
2. DSL Re-targetability: State of the Art

```xml
<Window x:Class="xamlExample.Hello"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="Hello World">
  <WrapPanel>
    <Button Click='HowdyClicked'>Howdy!</Button>
    <Button>A second button</Button>
    <TextBox x:Name='_text1'>An editable text box</TextBox>
    <CheckBox>A check box</CheckBox>
    <Slider Width='75' Minimum='0' Maximum='100' Value='50' />
  </WrapPanel>
</Window>
```

Figure 2.2: XAML sample user interface declaration

```csharp
public partial class Hello : Window {
    public Hello() {
        InitializeComponent();
    }
    private void HowdyClicked(object sender, RoutedEventArgs e) {
        _text1.Text = "Hello from C#";
    }
}
```

Figure 2.3: XAML code behind

The strength of XAML is that XAML is bound to code-behind files. It means that the developer can easily communicate to the user interface elements as class instances and consequently these elements can be modified based on the application logic. Figure 2.3 which is from [23] shows a sample user interface defined in XAML.

A XAML document logically defines a class [23]. The behavior can be added in another partial class. Figure 2.3 shows a corresponding partial class to wire the behavior to the specification shown in the previous figure.

XAML is tightly coupled with .NET framework and consequently, it is difficult to use it in other platforms. If one wants to do so, it can be only used as a specification language and all the rendering mechanism should be re-implemented.
2.4 MyMobileWeb

MyMobileWeb [26] is an open source software platform to create multi-platform mobile web application. MyMobileWeb is based on a declarative, platform-independent, XML-based language to describe the user interface specification which is called IDEAL. IDEAL is a markup language consisting of a set of AIOs for user interface specification. The designer can declare the structure of the user interface using these abstract elements to be compatible with all platforms. On the bottom of this specification, the user interface can be specialized for different platforms using CSS. It means that the target platform adaptation of the user interface is controlled by CSS. Basically, IDEAL provides a set of predefined adjustments according to the delivery context. But, these predefined settings can be overridden by the designer’s custom CSS styles.

In addition to AIOs and styling module, IDEAL consists of other modules as follows:

- Containers and layout managers: In order to group AIOs, two generic containers can be defined in IDEAL called "p" and "panel". The former is a simple container that only contains AIOs and does not support nested structures. On the other hand, "panel" similar to "tab", is for sub-screen switching. These containers can be styled using different kinds of layout managers by means of CSS. For example, AIOs grouped by "p" can be visualized as a grid, horizontal list or vertical list. The separation of the containers and their layout style facilitates the delivery context adaptation.

- Binding Module: Binding the context data to AIOs is done by using this module. The binding direction can be one-way, two-way or one time.

- Validation Module: According to the target platform capabilities, validations can be done either server-side or client-side. If the device supports client-side scripting, the validation scripts are created by this module and sent to the device. Otherwise, this module is responsible to do the validation on the server side.

MyMobileWeb uses JavaServer Pages (JSP) technology to create web pages. Pre-generation of the web pages for different platforms speeds up the loading process. However, for unknown devices, the page is created upon the request.

Figure 2.4\(^2\) shows a user interface definition in IDEAL. In addition, three different CSS specifications for three different targets can be shown in figure 2.5. Figure 2.6 shows the resulting layout in three target devices.

MyMobileWeb software platform provides a good degree of flexibility for the cross-platform user interface design. However, the layout can only be adjusted using CSS. In many cases, different high level concepts are realized differently according to delivery context. For example, the master-detail concept should be visualized through multiple screens in the devices with smaller screen size such as smart phones. However, in devices with larger screen like iPad, both the master and detail can be shown in one screen. These conceptual differences cannot be captured by CSS. Moreover, because of the web nature, the

\(^1\)Interface DEscription Authoring Language

\(^2\)http://mobiforge.com/developing/story/mymobileweb-an-open-source-platform-developing-mobi-compliant-applications
2. DSL Re-targetability: State of the Art

application created using MyMobileWeb platform cannot be accessed when the internet connection is not established. The limitation of the browsers to connect to different device components is another problem from which the mobile web applications suffer.

2.5 DIMAG

Device Independent Mobile Application Generation (DIMAG) [31] is a framework to generate cross-platform mobile applications by means of high level application description languages. DIMAG is based on three declarative languages:

- DIMAG-ui: This is a declarative language based on MyMobileWeb’s IDEAL. Although IDEAL is designed for the web user interfaces, DIMAG uses the IDEAL
Figure 2.5: IDEAL CSS specification

Figure 2.6: IDEAL final layout
specification to describe the native user interfaces.

- State Chart eXtensible Markup Language (SCXML): SCXML [7] is a declarative language to describe state machines by means of states, state transitions and conditional actions. SCXML currently is a working draft published by W3C. By using SCXML, it is possible to describe the workflow of application. Each scenario in an application is described as a set of states that can be transited to each other through events. Additionally, different actions can be set as the target of those events. These actions can be done through local method invocation or remote calls. To declare an action, a fully qualified name of its class and the method name should be provided. For example, if $x$ is provided as the assembly name of class $c$ containing method $m$, in a Windows Mobile device, $x.dll$ should be loaded while in an Android device, $x.jar$ should be found.

- DIMAG-root: DIMAG-root provides references to the user interface and workflow definitions.

DIMAG framework follows client-server architecture. The server component has four major layers. The first one is the communication layer which is a HTTP request listener. It forwards the request information to another layer called device detection and application provision. This layer uses a repository called Device Descriptor Repository. Using this repository, the request information is decoded and the exact type of the device which has sent the request is extracted. Based on this information, appropriate version of the application is selected and returned to the device for installation. If there is no appropriate version is available, the third module which is code generator will be invoked. If the device type is already supported, the code generator compiles the user interface and workflow declaration to the actual code. This new application is added to the repository to speed up the responding process of the future requests. The fourth module is synchronization module to synchronize offline data.

In the client component, in addition to client application which is downloaded from the server, there are a communication layer and a synchronization layer. The former is designed to request appropriate version of the application or to invoke the remote services. Similar to the server, the client synchronization component is responsible for managing data in offline and online modes. Figure 2.7 shows a workflow declaration sample in SCXML. Figure 2.8 shows the corresponding user interface declaration in DIMAG-ui. These figures are from [31].

DIMAG has solved the connectivity problem of MyMobileWeb by generating native applications. Additionally, by using the high level declarative languages to describe the application user interface and workflow, DIMAG facilitates the development of cross-platform applications. However, there is no explanation in [31] about the way used to compile the declarations. For example, the DIMAG approach to transform IDEAL user interface declaration to Android or Windows Mobile native user interface is not clear.

DIMAG suffers from not having an integrated DSL. Working with different declarative languages might be annoying for the developers. Additionally, the logic of the application which contains the class definitions, method implementation etc. should be developed us-
  <state id="applicationFlow" initial="login">
    <transition event="linit" target="menu"/>
    <state id="login" category="view">
      <transition event="loginButton_MOUSE_CLICKED" target="validateLogin"/>
    </state>
    <state id="validateLogin">
      <onentry>
        <dimag:invokeMethod scope="client" className="org.dimag.sample.Login" method="validateLogin" result="${loginOk}">
          <dimag:argument expression="${login}"/>
          <dimag:argument expression="${password}"/>
        </dimag:invokeMethod>
      </onentry>
      <transition cond="${loginOk=='true'}" target="menu"/>
      <transition cond="${loginOk=='false'}" target="login"/>
    </state>
    <state id="menu" category="view">
      <transition event="lsearch_MOUSE_CLICKED" target="search"/>
    </state>
    <state id="search" category="view">
      <transition event="searchButton_MOUSE_CLICKED" target="results">
        <dimag:invokeMethod scope="server" className="org.dimag.sample.Search" method="executeSearch" result="${list}">
          <dimag:argument expression="${criteria}"/>
        </dimag:invokeMethod>
      </transition>
      <transition event="lback_MOUSE_CLICKED" target="menu"/>
    </state>
    <state id="results" category="view">
      <transition event="lback_MOUSE_CLICKED" target="search"/>
    </state>
  </state>
</scxml>

Figure 2.7: SCXML sample workflow declaration
2. DSL Re-targetability: State of the Art

XMLVM [36, 35, 13] is a toolchain to facilitate the cross-compiling between different software platforms. Converting Java applications to .NET applications or Android applications to iOS applications are samples of this cross-compiling. In this cross-compiling process, first, the program developed in the source language is represented as an XML. Instead of using the source code, XMLVM derives the XML representation from the intermediate code which is generated by the source language compiler. For example, Java byte code and .NET Intermediate Language (IL) program are used to generate XML representation. Figure 2.9 which is from [36] shows a very simple class with a static method to calculate the factorial of an integer in Java. The resulting XML representation can be seen in figure 2.10.

As can be seen, corresponding to each bytecode instruction, an XML tag is generated in the resulting XML. Consequently, the generated XML does not provide any information more than the Java class files.

The next step of the cross-compiling in XMLVM is to generate the program in the
public class Math {
    public static int factorial(int n) {
        return n == 0 ? 1 : n * factorial(n - 1);
    }
}

Figure 2.9: A method to calculate factorial in Java

destination programming language. For this purpose, XMLVM uses XSL stylesheets [14] to transform the XML to the source code in the target language. Different XSL templates have been defined to transform the XML to the code in the destination language using pattern matching. As can be seen, since Java VM (and also .NET CLR) is a stack-based virtual machine, to generate the target code, a stack-based machine should be emulated in the destination programming language. Unlike JVM, Dalvik (which is the virtual machine of Android) is register-based. Therefore, to cross-compile an Android application to another platform, a register-based machine should be mimicked in the target platform.

XMLVM mainly focuses on retargeting the smart phones applications. For this purpose, XMLVM provides a toolchain to cross-compile Android applications to iPhone and also Palm Pre applications. In order to port Android applications to iPhone, first, a complete set of Cocoa Touch APIs\(^3\) have been re-implemented in Java. All the iOS widgets are simulated using Java 2D. Additionally, an iPhone emulator has also been developed to test the resulting applications. Java based Cocoa Touch provides the same APIs interfaces with its Objective-C ones. The program developed using Java based Cocoa Touch can be cross-compiled to Objective-C and thus can be executed on iPhone or generally iOS.

Now, it is time to port an Android application to iPhone. In order to achieve this goal, Android APIs should be mapped to Cocoa APIs. XMLVM uses compatibility library to do that. This compatibility library uses Java based Cocoa Touch APIs but offers the Android APIs. It means that the Cocoa Touch APIs are abstracted using wrapper classes that are compatible with the official published Android APIs. Figure 2.11[36] shows a Cocoa UIButton which is wrapped by Button class. Button has the same interface with Android native Button widget. Now, the resulting application is completely compatible with Android APIs and by changing the references from native APIs to compatibility APIs, it can be cross-compiled to an iOS application.

XMLVM provides a powerful toolchain to port the applications to new platforms. Using XMLVM, an application is developed and maintained in only one language. Additionally, since it does bytecode level cross-compiling, supporting new target platforms is easy.

As mentioned in [36], XMLVM is mostly appropriate for those groups of applications which use the native widgets less. Since, as mentioned earlier, mapping the widgets should be done at the conceptual level and not at the API level, XMLVM cannot perform the widgets mapping appropriately. Thus, XMLVM is more appropriate for porting the game

\(^3\)A set of APIs to build iOS applications
Figure 2.10: The XML that is generated using XMLVM tool.
applause

package android.widget;

public class Button extends View {
    protected UIButton button;

    public void setOnClickListener(OnClickListener l) {
        final OnClickListener theListener = l;
        button.addTarget(new UIControlDelegate() {
            @Override
            public void raiseEvent() {
                theListener.onClick(Button.this);
            }
        }, UIControl.UIControlEventTouchUpInside);
    }
}

Figure 2.11: Cocoa UIButton wrapped by Button class

applications which use the full-screen space with few widgets. One another disadvantage of XMLVM is that it does not address higher level programming. Although programming in one language may ease the development process, DSLs can make this even easier.

2.7 Applause

Applause [20, 1] is a multi-platform DSL to develop data-driven native mobile applications. Applause targets different platforms such as iOS, Android etc. The Applause DSL is based on a set of typed views. Detail view, table view etc. are samples of these views. Additionally, there is a set of abstract widgets which can be defined inside views. In the compiling process, these widgets are mapped to actual widgets of the target platforms. The Applause DSL provides entity constructors to define the application entities in order to hold content data. Furthermore, different data sources such as XML files, RSS feeds etc. can be mapped to these entities. Using the entity properties name, the mapping is done automatically. Applause has been developed using Xtext [15] which is a language development framework integrated with Eclipse IDE. Corresponding to each target platform, a code generator module is available to generate the source code in the destination language. There is also an extension of Applause called Applitude [2] which is specialized for iPhone.

Applause has been specialized for the content rendering and not for the content editing. The applications built by using the Applause DSL have the capability to connect to existing web resources, retrieve the data asynchronously, and present them using the standard widgets of the target platform [20]. One of the disadvantages of Applause is lack of flexibility. Applause has a set of limited widgets and built-in functions. The custom logics cannot be
implemented easily using Applause DSL. Moreover, the developer is not free to compose
different widgets to create re-usable composite controls. These limitations force the de-
velopers to manipulate the generated code in the target platform which brings maintenance
difficulties.

A set of sample applications developed using Applause can be found on [1].

2.8 MobDSL

MobDSL [28] is a DSL to develop cross-platform mobile applications. However, it is
mostly intended to be used as a tool to evaluate the DSLs that support mobile applica-
tions. As mentioned in [20], MobDSL is presented as a mobile application calculus. It
means that, the essential requirements of an application in mobile environment have been
studied and based on them, an extended version of $\lambda$-calculus to cover those requirements
has been proposed. Consequently, MobDSL can be used as an intermediate language to
which other DSLs that support mobile applications compile. This intermediate language
can be executed by using platform specific virtual machines.

In order to clarify the MobDSL structure, we are going to explain its general syntax in
brief. The syntax of MobDSL has the following features:

- **Basic $\lambda$-calculus**: MobDSL has been built based on $\lambda$-calculus to achieve a high
  level of expressiveness. Variables, constants, applications and first-class functions
  are the basic elements of $\lambda$-calculus. Considering functions as the first-class data
  values makes the computational model of the language more flexible and extensible.
  Supporting the first-class functions means that the functions can be passed to other
  functions as arguments, returned from other functions as return value, assigned to
  variables, and stored to data structures. To make the language more flexible, conditionals
  have been added as a $\lambda$-calculus extension.

- **Data extensions**: Lists and records are the MobDSL data structures. The record data
  structure has the same meaning as tuple data structure in mobl which will be described
  later.

- **Commands**: Commands are used to read and update values in memory. A command
  sequence is described as an ordered set of read and update instructions in addition to
  a return value.

- **Widgets**: A widget is a representation of an AIO which consists of a type, a record
  as its state, and a set of handlers to handle its events. A collection of widgets can be
  composed in a hierarchical structure. Every events of a widget should be handled in
  MobDSL. If no handler is found for an event in the widget declaration, the container
  tree of the widget is traversed to find one. The event handlers are defined as a record of
  functions which are responsible for updating the states and returning the superseded
  widget to event receiver. This definition of event handler makes the state transition
  easier. The mobile user interfaces mainly use state transition to navigate between
  screens.
MobDSL provides a powerful calculus to develop mobile applications. However, as mentioned in [28], because of the complicated syntax, it is mostly appropriate to use MobDSL as an intermediate language. In this case, MobDSL can be executed directly using the virtual machines. Some vendors like Apple do not allow the virtual machines to be installed on their devices. Therefore, alternative solutions such as XMLVM have been thought.

MobDSL does not provide any approach for API mapping. Especially in the user interface context, the MobDSL approach to map abstract widgets to platform-specific widgets is not clear.

A sample application developed in MobDSL can be found in [28].

2.9 PIL

Platform Intermediate Language (PIL) [25] is an intermediate language in order to fill the large semantic gap between DSL and the target platform code. PIL achieves this goal by providing a layer of abstraction between DSL and the destination platform.

PIL has been built based on a subset of Java. Most of the high level GPLs provides a set of capabilities that are only useful for developers. These facilities lose their importance when the code is generated. For example, the modifiers (e.g. private, public etc.) are useful to hide the details from the developer but when the code is generated using code generator, they are no longer useful. This kind of language features are discarded in PIL. PIL supports a subset of object oriented programming features. Supporting basic features of the object oriented languages increases PIL portability degree. At this lower level of abstractions, most of the target software platforms are very similar and thus, the re-targetability is cheaper.

On the other hand, there is a set of abstractions that are mostly useful when the code is generated. For example, the partial class and partial method concepts prevent code generator to deal with large classes and methods. In order to achieve these abstractions to facilitate the development process of a DSL to PIL transformer, an abstraction layer called PIL/G has been built on top of PIL.

Developing a DSL compiler on top of PIL is quite cheaper than building a DSL compiler which compiles DSL directly to the target platform. The advantage of this approach is that the different back-ends of PIL are developed only once, and after that, every DSL which is built on top of PIL is automatically re-targetable to the supported platforms. Additionally, because of the basic structure of PIL, implementing DSL compilers on top of that is cheap. Moreover, if a feature is added to the DSL, only the DSL to PIL compiler should be updated. In the case of using a compiler which directly transforms the DSL to the target platform, if multiple platforms are supported, all the corresponding compilers must be updated. This redundant task makes the evolution of the DSL expensive.

One of the major challenges that PIL faces with is the variety of APIs in different platforms. In many cases, although different platforms offer APIs with different interfaces, their APIs have the same behavior. Therefore, it is possible to wrap them behind a set of general interfaces. These interfaces are represented through external classes in PIL which will be mapped to platform-specific APIs during the platform-specific code generation. For
example, the user interface APIs can be implemented in this way although currently there is no specific support for the user interface.

Sometimes, a particular API is available in one platform but is not available on another platform. One solution for this problem is to use PIL as a language to port API to the platform which does not support that API. For example, as mentioned in [25], since there is no suitable object-relational mapper (ORM) for Python, a set of APIs in order to objects persistence and retrieval have been developed using PIL. Then a specific implementation of these APIs has also provided in Python. It should be mentioned that since the code in the target platform is generated, all the capabilities of an ORM which only facilitate the development process and do not have any value when the code is generated are discarded. Thus, a small set of APIs are needed to implement.

PIL provides a scalable method for re-targetability. By hiding the target platforms differences, the re-targetability of a DSL compiler is achieved in a cheaper way. PIL is suitable for the object-oriented target platforms. Building a code generator to transform PIL to a functional language is expensive. Obviously, using the multi-phase compiling may slow down the compiling process. However, the transformation in each phase is simple. PIL currently supports Java and Python as its back-ends. Building other back-ends has been planned as the future work.

As can be seen, there are different approaches proposed to achieve the software systems re-targetability. Some of them address the higher level programming by introducing DSLs such as UIML, XIML, Applause etc. Some other approaches use the different platform code conversion like XMLVM. In this thesis, our goal is to explore the challenges of the DSLs re-targetability. DSLs can hide many irrelevant details of software development process and highlight the essential semantics of their corresponding domains. However, retargeting a DSL to different software platform has own challenges. One major problem is that since a DSL defined on top of different platforms only highlights the commonalities between them, it is possible that the resulting application in the target platform becomes banal. It means that the application does not use special capabilities of a particular platform and thus, it is not acceptable by the users of that platform. This problem is also imagined in XMLVM although it does not provide a common language among different platforms. The compatibility library approach proposed by XMLVM to map the source APIs to destination APIs cannot capture special features of a platform. To some extent, this problem is solved in PIL using API porting approach. Although, all the features of a platform especially those related with the hardware cannot be ported to another platform and consequently, cannot be available through a general interface. As a result, the ideal solution should capture the commonalities and also provide facilities to capture the specialties of a platform. This goal is one of the things that we are willing to achieve in this thesis.

In the next chapter, we will describe the structure of "mobil" which is a DSL to develop mobile web applications. "mobil" has been developed by TU Delft Software Engineering Research Group (SERG) and like MyMobileWeb, targets the web to solve the multi-platform application development in the mobile devices.
Chapter 3

mobl: A DSL of the Mobile Web

mobl is a DSL to develop mobile web applications. The target language of mobl is HTML5, JavaScript, and CSS. mobl aims to provide an integrated language for different aspects of a web application such as the user interface, logic, data persistence, service consumption etc. in the mobile context. mobl creates complete client-side HTML5-based mobile web applications and it has no dependency on server-side technology [5].

In this chapter, the structure of mobl is described. In addition, the programming concepts behind them will be explained.

3.1 Technology

mobl has been developed using Spoofax [27]. Spoofax is a language workbench based on Eclipse [3] to provide an integrated environment to define grammar, declare transformation rules, and specify the editor services. In Spoofax, Stratego [9, 41] has been used as a program transformation language. The aim of Stratego is to provide a language to develop transformation systems in order to support wide variety of program transformations. The language grammar is defined using the syntax definition formalism (SDF) [8, 24] in Spoofax.

After defining the DSL grammar and the editor services like syntax highlighting, code completion etc. the developer will be able to use Eclipse as an IDE for that DSL.

3.2 Code Generation Phases

The target platform code generator of mobl like other DSLs code generators consists of two major parts: front-end and back-end.

In the front-end component, based on the language grammar, the abstract syntax tree (AST) is produced which is used for semantic checking. In the next phase, using model-to-model transformation, AST is transformed to a simplified version. This phase is known as desugaring. In this phase, all the abstraction layers which are only useful for the developers and do not have any value when the code is generated are removed from the model. In fact, using this phase, AST is transformed to a reduced set of core DSL constructs [25].
Transforming "If" statements to "if-else" statements with empty "else" block, filling the omitted optional parameters with their default values etc. are samples of transformations performed in the desugaring phase. These simplifications make the code generation process easier.

Now, the resulting model is ready to be used by back-end component in order to generate the target platform code. In the case of mobl, a simple framework has been built using JavaScript. Using the APIs offered by this framework, the back-end generator generates less code and this makes the compiler maintenance easier.

### 3.3 Structure

Basically, mobl is a functional language. However, mobl supports some object oriented concepts. An application developed using mobl consists of a set of modules. The modules bring modularity to a mobl application like "packages" in Java and "namespaces" in .Net. Global variables, functions, user interface components, and entities can be defined within a module. There is also a particular module which acts as an entrance module and it is defined using "application" keyword.

As mentioned before, mobl aims to provide an integrated language to cover different aspects of an application. In order to achieve this goal, mobl provides a set of language constructors for the user interface specification. In addition, by using functions, the logic of an application can be implemented. A function can be invoked as an initial expression of a user interface variable or can be called when a special event of a user interface component is fired. Furthermore, mobl supports a set of built-in data structures for different types of data. The custom data structures can also be defined using a subset of object-oriented programming concepts. Moreover, the persistable types are defined using entities. An embedded ORM handles the persistence and retrieval of these entities. Additionally, a SQL like notation has been linguistically integrated with mobl to retrieve the entities. This integration helps a developer by compile time checking of her queries.

As can be seen, a mobl application consists of three major parts:

- User interface
- Application Logic
- Persistence

In the upcoming sub-sections, these parts will be described. It should be mentioned that we mostly focus on the essential features of these parts in addition to those features which need more efforts in the retargeting process.

#### 3.3.1 User Interface

One important goal of mobl is to simplify the user interface specification for the mobile applications. For this purpose, the user interface widgets can be defined, bound to data, gathered together to construct the composite widgets, and shown inside a screen. Two fundamental components of a user interface specification in mobl are "screen" and "control".
These two components form the user interface context. Inside the user interface context, other user interface elements can be used to specify the behavior of the user interface.

**Screen**

Since mobl is specialized for the mobile devices, the user interface specification in mobl is adapted to this target. The user interface of a mobile application is constructed based on a set of full screen views. Since the screen size in the mobile devices is small, a limited set of information can be shown in one screen view and therefore, the navigation between screen views is one common behavior in the mobile applications. It should be mentioned that, our focus is mostly on the data centric applications and not on the other kind of applications like games, players etc. It is obvious that in such applications, mostly one full screen view is used and thus, the navigation between the screens is less needed.

The user interface specification in mobl also consists of a set of screens representing the full screen views. A screen definition sample developed in mobl can be seen in figure 3.1. The resulting layout can be seen in figure 3.2.

Although screen is defined like a function, it is not similar to an imperative function. Instead of having a set of statements executed sequentially, a screen declares the structure of the user interface. It means that the structure of the user interface and the data shown by the user interface can be changed through the state changes at anytime [5].

The user interface adaptation based on the data changes are mostly hidden from the developer. Consequently, the developer deals with a screen like a regular function. Thus, a screen can be invoked using a function call which results in a screen navigation. Like any other functions, a screen accepts arguments and also has an optional return type. When

```plaintext
screen root(){
    header("Tasks"){
        button("Add", onclick={
            //some logic
        })
    }
    group{
        item{
            label("Task #1")
        }
        item{
            label("Task #2")
        }
    }
}
```

Figure 3.1: A sample screen definition in mobl.
Corresponding to each screen, a function is created in JavaScript. When the screen function is called, the current visible screen will be hidden and the new screen will be shown instead. Using different action listeners and event handling mechanism, the state management of the user interface is implemented in the generated code.

Control

Another major component of the user interface specification in mobl is control. A control is a reusable user interface widget which can be defined using the combination of other controls and other user interface elements. Basically, the user interface model of mobl is based on the composition of controls. To realize this model, mobl is flexible in combining different controls in order to create more complex ones. Some platforms like iOS do not support this degree of flexibility. Instead, the widgets support multiple styles and can be customized to these styles based on the requirements. The differences between mobl user interface model and iOS user interface model will be explained later.

Similar to a screen, a control is also defined as a function (but not as an imperative function) which accepts a set of arguments. However, it does not have any return type and it has a control type which will be described later. A control definition sample in mobl can
control listControl(items : Array<String>){
    header("Items")
    group{
        list (i in items){
            item{
                label(i)
            }
        }
    }
}

Figure 3.3: A sample control definition in mobl.

be seen in figure 3.3. If this control is invoked within a screen, the output will be what you can see in figure 3.4.

In the web user interface design, the controls are built using HTML tags. In addition, JavaScript is also used to handle the client-side behavior of controls. mobl aims to provide the capability of defining the web controls completely in the mobl program. With this approach, there is no need to use the target platform user interface APIs in the DSL. This approach increases the extensibility of the DSL by allowing the user to develop her own control set. For this purpose, mobl includes HTML tags and JavaScript language in its syntax. As a result, the developer can combine mobl, HTML, and JavaScript facilities to create various kinds of web controls. However, as mentioned before, embedding the target platform languages into DSL decreases its level of abstraction and thus, retargeting the DSL to new platforms may face with problems. These problems will be described in the next chapter. A sample control built using HTML tags is shown in figure 3.5. The output of this declaration can be found in figure 3.6.

In mobl, the controls can also be defined externally. In this case, using external keyword, only the signature of the control is declared. It means that the implementation of the control can be located somewhere else. The external controls can be used for those widgets that implemented in the target language. Using this mechanism, mobl application can use the target platform user interface APIs by API wrapping technique. As a result, it is possible to retarget the application to another platform and use the standard APIs of that platform.

A set of predefined controls have been embedded in mobl. However, all of them are built inside mobl. These predefined controls are categorized based on the different target platforms. Although mobl targets the web applications, the web controls can also be adapted regarding to the destination platform. For example, ios library provides a set of controls that their layouts are adapted for iOS platform.

Similar to the screens, a control is called through a function call. When a control is called, as a default, its body will be rendered next to the last control which has been previously called. Clearly, the location of the controls can be customized using the CSS attributes.
3. MOBL: A DSL OF THE MOBILE WEB

Figure 3.4: The output of the control declaration.

```plaintext
control HtmlSample(){
  <div>
    "Hello World!"
  </div>
}
```

Figure 3.5: A sample HTML control definition in mobl.
When a control is called to be rendered, a set of additional user interface elements can be also passed to it. In fact, these elements construct a special control called "elements". Elements are passed automatically to the control as a function variable. This function variable can be invoked wherever in the implementation of the control. A usage of elements in mobl is shown in figure 3.7. The resulting output can be seen in figure 3.8. As can be seen, a "group" control is passed to the custom control as its elements.

In mobl, a control can be implemented inside another control. Generally, mobl supports closure. A closure (also known as lambda) is a block of code that can be represented as a first-class data structure and placed seamlessly into the flow of code by allowing it to reference its lexical scope [23]. A closure may occur when a function is defined inside another function. In this case, the inner function, in addition to its local variables, can access to its outer scope variables called upvalues. For simplicity, the local variables and the outer scope variables of a closure are called the closure context variables. As mentioned before, a control in mobl, structurally (not behaviorally), is a function and thus, it can be defined as a nested control. In the code generation phase, since JavaScript also supports closure, the nested controls are translated to the nested functions. In this case, the similarity between the DSL and the target language simplifies the code generation process. However, retargeting to another platform which does not support closure may face with some challenges. A nested control declaration can be seen in figure 3.9. The output of this declaration can be found in figure 3.10.
3. **mobl: A DSL of the Mobile Web**

```java
control elementsSample()
    header("Sample")
    elements()
}
screen root()
    elementsSample()
    group{
        item{
            label("Item1")
        }
        item{
            label("Item2")
        }
    }
}
```

Figure 3.7: A sample of using elements in mobl.

Figure 3.8: The output of the control built using "elements".
control contextControl()
    header("Context")
control nestedControl()
    label("I am inside")
// invocation of the nested control
nestedControl()

Figure 3.9: A sample of a nested control declaration.

Figure 3.10: The output of the nested control declaration.
3. mobl: A DSL OF THE MOBILE WEB

```javascript
control htmlDataBind()
{
    var txt = "This is a text"
    <input type="text" databind=txt/>
}
```

Figure 3.11: A sample of data binding declaration for an HTML control.

**Derived Variables and Data Binding**

As mentioned above, the structure and the content of the user interface defined in mobl, can be changed any time through the state changes. This is a realization of a programming paradigm called **reactive programming**. Based on this paradigm, data flows should be expressed easily in the programming languages. For this purpose, the execution model should be able to propagate the changes using the data flow. Reactive programming is common in spreadsheets such as Microsoft Excel. In Excel, a value of a cell can be set either by direct value assignment or by using a formula. For example we can define a formula as follows:

\[ f := \text{sum}(b_1 : b_{10}) \]  

(3.1)

Then it is possible to bind this function to a particular cell like c1. Any change in b1, b2,..., b10 notifies c1 to re-evaluate \( f \) and update its value.

Reactive programming is one of the core programming models of mobl. Derived variables act an important role to realize this paradigm. In mobl, it is possible to bind an expression to a variable called **derived variable**. It means that any change in the variables that are included in that expression notifies the derived variable to re-evaluate the expression and update itself. A derived variable can be defined only in the user interface context and after being initialized by an expression, new value cannot be assigned to that explicitly.

In desugaring phase, if an expression is passed to a control as an argument, that argument is transformed to a derived variable to be notified about changes. This mechanism causes the propagation of changes to the controls in any depth of the controls hierarchical structure. However, if a simple variable (not an expression) is passed to a control, it will not be changed to a derived variable since the notification here is meaningless.

As mentioned earlier, the user interface widgets can be defined as HTML tags. In this case, in order to bind data to a control, the keyword "databind" should be used. Using databind, the HTML control which should be notified about changes is added to the listeners of all the variables included in its bound expression. Consequently, the HTML control will be notified if those variables are changed. If the HTML control is editable, instead of one-way data binding, a two-way data binding will be established. It means that in addition to value-to-control direction, the change propagation is done in control-to-value direction. In this direction, by editing the control value, the bound variable will be updated. A sample data binding of an HTML control is shown in figure 3.11. The resulting output can be found in figure 3.12.
If a control is defined externally, the data binding should be handled in the target platform. Our approach for this purpose will be described in the next chapter.

**when-else Clause**

A particular part of the user interface can be rendered conditionally. For this purpose, mobl provides ”when-else” clause. A sample of ”when-else” usage can be seen in figure 3.13.

Unlike traditional imperative ”if-else” which is executed once, ”when-else” clause is reactive. It means that, the condition expression is bound to the clause and ”when-else” acts similar to a control with a condition expression as its argument. Therefore, ”when-else” listens to all variables included in the condition expression. If any of them is changed, the expression is re-evaluated and based on its new value either ”when” block or ”else” block will be rendered and the other one will be hidden. In the sample shown in figure 3.13, if the ”Change” button is clicked, the value of the ”change” variable is logically complemented and since it is used in the condition of the ”when-else” clause, the content of this clause will be re-rendered. The resulting output for both cases of the ”change” variable can be shown in figure 3.14.
screen root()
{
  var change = false
  header("Screen"){
    button("Change", onclick={
      change = !change;
    })
  }
  when (change){
    label("Change is true")
  }
  else{
    label("Change is false")
  }
}

Figure 3.13: A sample of "when-else" usage in mobl.

Figure 3.14: The output of the "when-else" clause.
screen root(){
    var items = ["item1", "item2"]
    group{
        list {i in items}{
            item{
                label(i)
            }
        }
    }
}

Figure 3.15: A sample of "list" usage in mobl.

list Clause
The user interface controls can be located on the screen through a loop constructs. For example, in order to represent the elements of an array, "list" construct can be used. Figure 3.15 shows a sample of "list" usage. The output of this sample code can be seen in 3.16.

If a "list" iterates over "Collection", it will be reactive and thus, if an element is added to or removed from the collection, the "list" block will be re-rendered. "Collection" will be described later in this chapter.

Control Variable
As mentioned before, mobl supports first-class functions. Therefore, a function can be assigned to a variable. Similarly, in mobl, a control can also be assigned to a variable and can be passed as a value. Control variables are also reactive. Figure 3.17 shows a sample of using a control variable. In this sample, whenever the "Change" button is clicked, "isEdit" variable is logically complemented. Based on the "isEdit" value, "controlVar" which is a control variable is set either to "edit" control or "view" control. Since this variable has already been invoked in the screen declaration, any change on its value will result in its re-rendering. Figure 3.18 shows the behavior of the code demonstrated in figure 3.17. As can be seen, if the "Change" button is clicked, the text field will be changed to a label or the other way around.

Styling
Another feature of mobl in the user interface context is styling. Inspired by Sass [6], mobl provides an abstraction layer on top of CSS in order to facilitate the beautification process of the user interface. Styling in mobl has some features as follows:

- Styling Variables: A styling variable is a variable representing a value for a particular styling attribute. Styling variables can be used to customize the application theme.
3. MOBL: A DSL OF THE MOBILE WEB

For example, mobl generic library widgets used a shared styling variable for their color called $baseColor. This variable can be overridden by the developer and thus, the color of all the generic controls can be changed to the new value. It is also possible to do some calculations on the values assigned to the styling variable. For example, these calculations can be used for lightening and darkening of controls.

- **Custom Styles**: The mobl widgets can accept some additional styling argument. These styles can be defined through the custom styles. A custom style is defined as a pure CSS code supporting some extra features. One of these features is that in a custom style definition, the styling variables can be used as the value of a particular style attribute. Another feature is "mixin styles".

- **Mixin Styles**: Mixin styles remove the redundant style definitions by parameterizing the styles. The common style attributes can be located in a mixin style and the differing values can be passed as an argument to that mixin style. Mixin styles can be called from the custom styles.

A custom style declaration and its usage can be seen in figure 3.19. The resulting output is shown in 3.20.
control edit(s : String){
    textField(s)
}
count view(s : Strong){
    label(s)
}
screen root(){
    var content = "This is a text"
    var controlVar = edit //This is a control variable
    var isEdit = true

    header("Screen"){
        button("Change", onclick={
            isEdit = !isEdit;
            if (isEdit)
                controlVar = edit;
            else{
                controlVar = view;
            }
        })
    }
    controlVar(content) //Control variable invocation
}
3. **mobl: A DSL of the Mobile Web**

![Figure 3.18: The control variable sample output.](image1)

```plaintext
style sampleStyle{
    border: 1px solid #000;
    background-color:#ff0000;
}
screen root(){
    button("Sample Button", style=sampleStyle)
}
```

![Figure 3.19: A sample of "style" usage in mobl.](image2)

function concat(str1 : String, str2 : String) : String{
    return str1 + str2;
}

Figure 3.21: A sample of function declaration in mobl.

As can be seen, there is a special programming style is needed to achieve the asynchronous invocation which makes the code hard to read.

mobil wraps asynchronous JavaScript APIs using external functions. Similar to external controls, external functions are those functions that implemented somewhere else. External functions can be used to wrap the target platform APIs. In mobil all the external functions are assumed to be executed asynchronously. But there is no difference between calling an asynchronous function and a synchronous function. In fact, mobil hides callback functions from developer point of view and makes it automatic. As a result, even for asynchronous APIs, synchronous programming style is sufficient. Offering asynchronous APIs using synchronous coding style is implemented using continuation-passing style (CPS) transformation. Conceptually, in CPS, a procedure accepts an extra argument representing what should
3. MOBL: A DSL OF THE MOBILE WEB

```javascript
type Person{
    firstName : String
    lastName : String
    age : Num
}
```

Figure 3.22: A sample of "type" declaration in mobl.

be done next with the function’s calculation results. This extra argument is called the procedure *continuation*. The CPS is applied automatically by mobl compiler. It means that, all the code come after the external function invocation is considered as the continuation and passed to that function as a callback function. As a result, when the calculations of the function are done, the continuation will be called and consequently, the program continues normally. As can be seen, with this mechanism, it is possible to call asynchronous JavaScript APIs using synchronous programming style. Furthermore, to execute a block of code in parallel, mobl provides "asynch" block. The code located inside this block is conceptually executed in a different thread.

**Typing System**

As mentioned before, mobl is a statically-typed language with type inference. It means that it is not necessary to explicitly express the type of a variable when it is declared. Instead, the compiler draws conclusions about the type of variables. For example, if a variable is initialized by concatenation of two strings, the compiler can conclude that the type of that variable is string. As a result, the type checking can be done at the compile time. This enables IDE to provide different features such as error reporting, code completion etc.

A custom type can also be defined using "type" keyword. A set of object-oriented programming features such as inheritance, type methods etc. are supported in mobl. When a type declaration is translated to JavaScript, it is defined as normal JavaScript object definition.

In fact, types are in memory objects and they have not been designed to be persisted. All the mobl built-in types such as Object (the base type of all types), Num (any kind of numbers), String, Bool etc. have been defined as types. Figure 3.22 shows a type declaration sample.

In addition to simple types, mobl supports generic types. A generic data type is a parametric data type which can be instantiated at many different types [43]. These types are passed to the generic data type as arguments. Comparing to using base type, generic data type reduces the number of type casting and increases readability of the code. Figure 3.23 shows a generic type declaration sample which has been defined in mobl libraries.

At this time, mobl only supports its built-in generic types and defining a custom generic type is not possible. mobl compiler, like Java, uses *type erasure* way in order to generate JavaScript code for generic types. It means that the generic type information is removed.
Figure 3.23: A sample of built-in generic type declaration.

external type Array<T>{
    length : Num
    function get(n : Num) : T
    function one() : T
    //.....
}

when the code is generated. For example, Array < String > will be compiled to a general JavaScript array.

As mentioned before, in mobl, because of considering functions as a first-level data structure, controls can be assigned to variables. Since mobl is statically-typed with type inference, a static data type should be assigned to these control variables. For this purpose, a set of generic types based on the controls arguments number are provided in mobl. For example, similar to Scala programming language\(^1\), if a control which accepts two string arguments is assigned to a variable, the type of that variable will be Control2 < String, String >. This mechanism allows a control to be passed as an argument. This story is the same for mobl functions.

3.3.3 Persistence

mobil provides entities as the persistable objects. Entities have the same structure with mobl types but in spite of types which are in-memory data, entities are stored in the local database. The persistence of entities is done transparently. As a result, entity-relation mapping is done automatically. Figure 3.24 shows an entity declaration sample and its instantiation.

In the web target, mobil uses client-side storage. Basically, the client-side storage can be done in three ways. Using session storage is the most common way in which the data are stored in cookies as the key-value pairs. The second way is using local storage. The local storage is functionally identical to the session storage, but the local storage is a long-term storage. Moreover, the local storage is accessible by all the browser windows while the session storage can be only accessed through the browser window in which it was created. The third storage is the database storage which is provided by HTML5. Using database storage, instead of key-value pairs, data can be stored in a structured manner. HTML5 provides a set of APIs called Web SQL Database API for data persistence and retrieval in the local database. Different browsers have their own implementation of these APIs but most of them use SQLite as their underlying database. mobil uses HTML5 Web database API in order to store the entities. Its mechanism is that first, the compiler traverses all the entity definitions and based on them, appropriate JavaScript code to create the database and its tables are generated. When the page is loaded, if the storage has not been created yet,

\(^1\)http://www.scala-lang.org/
those scripts are executed and thus, storage is created. Additionally, mobl provides a set of general APIs to add a new entity and delete an existing one. Updating the entities is done transparently by pushing in-memory entities into the storage in certain time intervals.

In order to access the stored data, mobl provides collection type which is a virtual collection of items. Collection type is inspired by “query” class of Google App Engine\(^2\). Collection is a generic type which can be instantiated on any entity type. Figure 3.25 shows how a collection can be instantiated for an entity.

The main APIs provided by collection type is as follows:

- **filter(property: String, operator: String, value: Object)**: This function filtered the collection items by some predicates and returns a new collection. Returning a new collection helps developer to define compound predicates as follows:

  \[
  \text{var col} = \text{Task.all().filter(”date”, ”<”, now()).filter(”done”, =, false)} \quad (3.2)
  \]

- **order(property: String, ascending: Bool)**: This function sorts the collection items in either ascending or descending way and returns a new collection.

\(^2\)http://code.google.com/appengine/docs/python/datastore/queryclass.html
function Retrieve1() : Collection<Person>
{
    return Person.all().filter("age", ">", 40).order("firstName", true);
}
function Retrieve2()
{
    return Person.all() where age > 40
        order by firstName asc;
}

Figure 3.26: Two programming styles in mobl in order to use data retrieval APIs.

- list(): As can be seen, above APIs do not retrieve any entity from the storage. They only hold some query metadata. The "list()" API based on this metadata retrieves appropriate entities from the database. In fact, it executes the query represented by the collection type and returns an array of entities.

In addition to these APIs, the collection type provides several other APIs which can be found in mobl documentation.

In order to linguistically integrate the SQL query language into mobl, these APIs are syntactically sugared to SQL like syntax. For example, Figure 3.26 shows a sample code developed in mobl using two programming styles. Although they are identical ("Retrieve1" and "Retrieve2"), the second function has a SQL like query which is more familiar and it can be validated at compile time.

In this chapter we explained the major parts of mobl. Our focus was mostly on those parts that need more efforts in the retargeting process. In the next chapter, our approach to port these features to iOS platform will be described.
Chapter 4

iOS Native Applications as a New Target for mobl

By using the web approach, the management of a large portion of the retargeting issues has been delegated to the browsers. This is the reason why mobl and MyMobileWeb have chosen the web approach in the mobile context. However, as mentioned before, the web approach has its own disadvantages. First, the browsers cannot use all the capabilities provided by a device. For example, the browsers have some limitations in order to connect with different device components. The second major problem is the connection between the client and the server which should be established in order to access to the application. Clearly, the constant connection is not possible all the time. Another problem is that each target platform has its own standards for the applications running on that. For example, each platform provides a set of patterns to design the user interface of an application. These standards are something beyond the appearance style and they can be related to the behavior of the widgets, the patterns to locate the widgets on the screen etc. Consequently, in MyMobileWeb in which the layout of an application can only be adjusted using CSS, it is not possible to provide a complete user interface adaptation based on the target platform. mobl has solved this problem in another way by providing a new set of widgets per platform. In this way, the web version of the target platform standard widgets should be developed. Although by using this solution, the resulting application is more closely resemblance to the native applications, because of using the platform graphical APIs, the native user interface widgets have better appearance quality. Additionally, the web control libraries should be evolved by the evolution of the native widgets to keep the similarity. Obviously, this process is expensive.

These disadvantages motivated us to build a new back-end for mobl to support the mobile native applications. For this purpose, we have chosen iOS native applications. iOS is the operating system of Apple portable tablets such as iPod touch, iPhone and iPad. In this thesis, we mostly focus on iPad device. The reason of this decision is since iPad has a wider screen, the user interface model of that has some differences with iPod and iPhone. These differences make the portability more challenging. In this chapter, first we briefly describe the infrastructure of iOS. Afterwards, our approach to build an iOS back-end for mobl will be explained.
4. **iOS Native Applications as a New Target for mobl**

4.1 **iOS Architecture**

iOS is the operating system running on iPod touch, iPhone and iPad. In addition to manage the device hardware, iOS provides the technologies needed to develop the platform native applications. An iOS application can be developed using iOS SDK. The iOS SDK is all of the interfaces, tools and the resources needed to build an iOS application. Xcode IDE, Interface Builder (IB), iOS simulator etc. are some of the tools which are provided by the iOS SDK. Objective-C is the primary language to build an iOS application. Objective-C was created in the early 1980s as a thin layer on top of C in order to provide the object oriented features. However, Apple has extended Objective-C by adding some higher-level features such as blocks which will be described later.

iOS has four major layers. Figure 4.1 shows the overall architecture of iOS.

![iOS Architecture Overview](image)

**Figure 4.1: iOS architecture overview [4].**

4.1.1 **Cocoa Touch Layer**

The essential frameworks needed to build an iOS application are provided by Cocoa Touch layer. Multitasking, gesture recognition, standard widgets and their corresponding controllers etc. are some of the features provided by this layer. In addition to these fundamental facilities, iOS provides a set of higher level frameworks. Using the interfaces offered by these frameworks, the features that mostly needed in the mobile context can be easily added into an application. Address Book UI framework for managing the contacts, Event Kit UI framework for managing the calendar-related events and tasks, iAd framework for adding advertisements etc. are some of these high-level frameworks. One of the most important frameworks of Cocoa Touch Layer is UIKit framework. UIKit framework manages the iOS applications user interface by incorporating different design patterns. The user interface built using UIKit framework is based on the Model-View-Controller (MVC) pattern. Every widget is presented through a view. As a result, from now, we use the `view` term instead of the `widget` term. The behavior of a view is handled by its corresponding `controller`. Loading the view, handling its event, managing its state transition, adapting the appearance based on the device orientation, and releasing the view are those tasks that are managed by the controller. The underlying data which should be presented and manipulated using the view is...
represented by model. For example, "UITableView" is one of the most common views in iOS applications. "UITableViewController" is the predefined controller of this view which is responsible for managing its behavior. Binding a model object (data source) to the table, customizing the presentation of data source items in the table cells, changing the state of the table from the view mode to the edit mode and the other way around are some of the tasks that are handled by this controller. Figure 4.2 shows a "UITableView" sample.

In order to provide a concrete view about UIKit framework, we continue with the description of the user interface structure of an iOS application. Basically, an iOS application user interface has a special element called UIWindow which provides an area for displaying the views and distribute the events to them. Typically, every iOS application has only one window but it may have several different screens. UIWindow accepts a root controller as a container of the screen elements. Other views are added to the root view. Figure 4.3 shows an iOS application using a tab control as its root view. For this purpose, the UIWindow root controller has been set to UITabBarController.

By using the screen navigation mechanism provided by a special controller called "UINavigationController", it is possible to navigate between different screens of an application. As
can be seen, similar to other user interface frameworks, UIKit provides the user interfaces with the hierarchical structure. However, iOS has placed several restrictions for combining the views in order to force the developers to follow the iOS user interface guidelines.

Because of the wider screen of iPad, UIKit provides a set of special controllers for that. One of these controllers is "UISplitViewController". Using this controller, the screen can be divided into two sections which are appropriate to present master-detail data. Figure 4.4 shows a sample of UISplitViewController. As can be seen, By selecting any item in the master part, the corresponding detail information will be shown in the detail part.

The user interface of an iOS application can be designed either by Interface Builder or programmatically. Interface Builder (IB) is a designing tool that comes with the iOS SDK in order to graphically design the user interface. Using IB, the design output is an XML
file called .nib\(^1\) file. When the application is launched, the objects defined in this file will be unarchived and joined to the application binary. In fact, IB serializes the user interface objects into a .nib file which should be desterilized when the application starts. The user interface can also be created programmatically by using UIKit framework APIs. We have used the latter approach in order to create the user interface views since we did not want to deal with two different target languages (XML and Objective-C).

### 4.1.2 Media Layer

Media Layer provides the graphics, audio and video technologies for the multimedia mobile applications. In this thesis, our focus is not on this kind of applications.

### 4.1.3 Core Services Layer

The basic system services and technologies used directly or indirectly by all applications are gathered in this layer. Some of these technologies are as follows:

- **Block Object**: a block object is a language construct in order to create closures. By using this construct, it is possible to create anonymous functions and treat them as a

\(^1\)NeXT Interface Builder; As of Interface Builder version 3.0, the format of the .nib file is changed from bundle to flat file and since then it has been called .xib file.
first-level data structure. A block object can share and modify the variables defined in its context (outer scope) even after the context is destroyed. As can be seen, this is a good option to implement mobl closures. However we did not use this iOS feature. The reason of this decision and our alternative solution will be described later.

- **Grand Central Dispatch (GCD):** GCD is a technology to provide asynchronous programming. Handling the thread for parallel computations is the main responsibility of GCD.

- **Core Data Framework:** Core Data framework is a technology to manage the data model of an iOS application. In fact, Core Data framework is an extended ORM for data persistence and retrieval. Core Data uses SQLite as its underlying database. Using Core Data APIs, the application objects can be stored, retrieved, filtered, sorted, and grouped. The object-relation mapping needed for this features is done transparently and all the modifications are propagated properly to keep the consistency of the relationships between objects. We have used Core Data framework in order to manage mobl entities.

- **Core Foundation Framework:** Core Foundation framework is the set of C-based interfaces for basic data management. Collection data types, string management, date and time management etc. are some of the features that are provided by this framework.

In addition to these technologies, Core Services layer provides several other facilities which are not used in our implementations. The detail of these features can be found in iOS documentations.

### 4.1.4 Core OS Layer

Core OS layer provides a set of low-level features upon which other technologies are built. Accelerate framework for mathematical computations, Security framework for security management etc. are some of these features. Generally, Core OS APIs are not used directly from an iOS application and mostly used by the other layers of iOS.

After this brief description about the iOS architecture, we continue with the explanation of our approach to build a new mobl back-end in order to generate iOS native applications.

### 4.2 iOS: a New Target for mobl

In this section, the approaches used in this thesis to build a mobl-to-Objective-C compiler are described. In order to build this compiler, we have employed the same toolchain that has been used to build the web mobl. As a result, the transformation rules have been developed using Stratego under Spoofax language workbench. As mentioned before, in the front-end component of the compiler, the AST of the mobl program is created. Afterwards, during desugaring phase, using a set of model-to-model transformations, the AST is reified to a simpler version consisting of the core mobl constructs. The resulting model of the desugaring phase is the source model to implement different back-ends of mobl. Thus, Objective-C
Figure 4.5: The domain model of mobl-ios.

code generation phase is done by employing this model in our approach. However, before pretty printing the model to the target language, we also apply a set of model-to-model transformations. The reason of these transformations will be described later.

**mobl-ios Framework.** In order to decrease the amount of the code generated by the code generator, an extensive framework has been built on top of iOS. We called this framework *mobl-ios*. Figure 4.5 shows a domain model representing the structure of this framework. For clarity, only the most important components of mobl-ios have been highlighted in this diagram.

As can be seen, mobl-ios has two layers: user interface and core services. The implementation of the external controls defined in mobl, handling the data binding of the controls, managing the style of the controls, and handling the controls events which are represented in mobl as callbacks are some of the responsibilities of the user interface layer. On the other hand, the implementation of the external data types, simulating the lexical scopes to handle the closures and providing notifiable objects in order to propagate the variable changes are
4. iOS Native Applications as a New Target for mobl

handled through core services.

In the upcoming sections, we described our transformation approaches for different aspects of mobl.

4.2.1 User Interface

As mentioned in section 3.3.1, two major elements of the mobl user interface are screen and control. On the other hand, according to section 4.1.1, the user interface of an iOS application is built upon a set of root controllers acting as screens which consist of a different kind of inner widgets called views. Obviously, in order to transform mobl code to Objective-C code, their user interface constructs should be mapped together. In this section, first we describe how the screens and controls are implemented in mobl-ios. Afterwards, the transformation of other user interface elements will be explained.

Screen

In order to generate an iOS application from a mobl program, first of all a mapping between the mobl screens and iOS screens should be defined. As mentioned in 3.3.1, the mobl user interface is built using a set of screens act as a root container for other controls. As described in 4.1.1, in an iOS application, different kind of views (with their corresponding controllers) can be used as the root views of different screens and the navigation between them is achieved using UINavigationController. In order to map the mobl screens and the iOS root views, in mobl-ios, a root view called widgetHolder has been defined. This class is nothing more than a regular mobl-ios control will be described next. Corresponding to each mobl screen, a class is inherited from widgetHolder and the body of the screen is implemented in render method of that. Every control in mobl-ios has a render method in order to render the appearance and the inner controls.

The widgetHolder control employs iOS UINavigationController in order to manage the navigation between screens. This controller uses a stack to handle the navigation and the current screen is always placed at top of the stack. When a screen is called to be shown, it will be pushed to that stack. Whenever the user wants to see the previous screen, the current screen will be popped from the stack.

As can be seen, although different kind of controllers can act as the root controller of an iOS screen, since there is no difference in declaration between screens in mobl, we use UINavigationController as a general root controller. Other controllers are added to this root controller. This mechanism will be explained later.

Predefined Controls

In order to achieve the mapping between the mobl and iOS user interface controls, first a set of ios specific external controls have been declared in mobl. As described in section 3.3.1, the predefined mobl controls can be categorized in different libraries based on the target platform. Thus, iPad controls can also be defined in a particular library. The implementation of these controls should be done in the Objective-C side unless either we inject the Objective-C grammar into mobl or deal with the Objective-C code as constant strings.
Clearly, both of these approaches are expensive. Additionally, they are completely in contradiction with the re-targetability principles. If we do so, then for each new target, the capability of using the target language code in mobl should be provided and this makes mobl a complicated and non-declarative language. Consequently, the implementation of the iPad control library is better to be done entirely in the Objective-C side and thus, the predefined controls are declared as external controls.

In order to increase the degree of portability, the controls defined in iPad control library provide the same interfaces as the generic mobl controls offer. However, in order to capture the specialties of an iPad application, a set of iPad specific controls have also been defined there (e.g. splitView).

Now, we continue with the explanation of the implementation of these controls in mobl-ios. Corresponding to each control defined in iPad control library, a class has been defined in the user interface layer of ios-mobl with the same name. Having the same name makes the code generation easier since there is no need to map the mobl controls name to mobl-ios controls name.

In fact, the task that is done by the mobl-ios controls is similar to what the compatibility library does in XMLVM. Basically, a mobl-ios control wraps an iOS view but offer it as a mobl control. For example, "label" is one of the generic mobl controls. iOS also has a control (or view) called "UILabel". A mobl-ios control wraps UILabel by providing a label class which has the same behavior as mobl label. On the other word, mobl-ios label class acts as a controller for UILabel to handle its behavior such as data binding, event responding, style managing etc. Thus, we use MVC pattern in the user interface layer of mobl-ios. As mentioned in 4.1.1, iOS itself also uses MVC pattern and provides some sort of controller for the views. Sometimes, especially for the complicated controls, a mobl-ios controller (wrapper class) employs the iOS controller to delegate some of tasks to it. For example, "group" is a mobl control to show items in a list. Its corresponding iOS control is UITableView which is controlled using UITableViewController. The group class implemented in mobl-ios to wrap UITableView uses UITableViewController as a property to benefit its capabilities.

As can be seen in Figure 4.5, all of the controls defined in mobl-ios user interface layer inherit from a base class called "baseControl". All of the common features of the controls such as handling the child controls, managing the arguments etc. are done by this class. Figure 4.6 shows a sample code for the control transformation between mobl and iOS. As can be seen, first, the arguments of the control are collected into an array. Then, the corresponding mobl-ios user interface view is initialized. Afterwards, this view is rendered using that array of arguments. Finally, the rendered view is added to its container.

Mapping the mobl controls to iOS controls has its own challenges and difficulties. The following bullets summarize these difficulties and describe our solution for them.

- **Choosing the most appropriate iOS control for a mobl control**: As described before, one of the goals of the DSLs like mobl is to capture the features of the different platform through a set of common interfaces. As a result, it is expected when a control is invoked in mobl, the most appropriate matching of that control will be chosen in the target platform. For example, mobl provides button control to enable the user to do
Figure 4.6: mobl to Objective-C control transformation.
some actions. On the other hand, iOS presents different kinds of buttons. One of the basic one is UIButton. Figure 4.7 shows a sample of this control.

![Figure 4.7: iOS default button.](image)

If you have any experience of working with iPod touch, iPhone or iPad, you may realize that this kind of button is not common in the standards applications of these devices. Instead, the iOS applications mostly use "UISegment" loaded with "UISegmentedControlStyleBar". Figure 4.8 shows a sample of this control which might be more familiar.

![Figure 4.8: iOS standard button.](image)

Consequently, the users will satisfy more if the mobl button is mapped to this type of iOS button. It can be concluded that to provide a DSL which generates standard applications of the target platform, studying the target platform concepts and APIs might not be sufficient and all the conventions and standards of that platform should be investigated.
Moreover, a DSL should be also flexible and be able to cover the specialties of the target platform. Imagine that in a particular situation, the developer actually needs to use the normal UIButton of iOS. As a result, it is worthy to provide another iOS special control which is mapped to UIButton. However, capturing these platform dependent features brings some difficulties when a mobl program is ported to another platform.

- **Differences between implementations**: Sometimes, the user interface concepts might be implemented in different ways in mobl and iOS. For example, in a group control, every item of the list is presented using an item control located in the elements of the “group” control (see 3.3.1). As a result, normally, two different controls corresponding to group and item should be developed in the user interface layer of mobl-ios. Figure 4.9 shows a sample code to create a group in mobl and figure 4.10 shows its web output.

On the other hand, iOS uses UITableView to present a list of items. Unlike the group control in mobl which accepts items as its inner control, in iOS, representing the items is done using UITableViewController using a function. When this function is overridden, based on the index of the current item, a customized UITableViewCell will be returned to represent an item.

To fill this implementation gap between mobl and iOS, using the mobl-ios wrapper classes, the mobl model has been built on top of iOS. Technically it means that, similar to other controls, a wrapper class for the item control has been implemented in mobl-ios. But unlike the common controls, no actual iOS view is related to item class. Instead, when this control is added to a “group” control, it will be added in an auxiliary array defined as a property of the “group” control. As mentioned before, the group control delegates some of it tasks to UITableViewController. In this case, UITableViewController renders the items stored in that auxiliary array. Figure 4.11 shows the iOS output of the code shown in figure 4.9.
Limitations in the user interface hierarchical structure: iOS has imposed a set of restrictions on the composition of different controls. These limitations cause some difficulties in order to transform a mobl user interface to an iOS user interface. One of these difficulties occurred in the screen transformation. As described earlier, in an iOS application, different kind of controllers can be used as the root controllers. UINavigationController, UITabBarController, and UISplitViewController are some of these controllers. On the other hand, in mobl, the screens are not typed. Consequently, one way to find the most appropriate iOS root controller for the mobl screen is to parse the inner controls. For example, if the mobl screen has a root tab control, UITabBarController can be selected for the root controller. The analysis of the inner controls to find the type of the root control usually is not that easy. For example, it is possible that the tab control is wrapped by a custom control and that control is called. It is also possible that the tab control is shown conditionally (using “when” clause). Then the condition in the iOS side should be applied at the root controller which is not possible since the root controller cannot be changed after loading the screen. As a result, the most promising solution is to provide a general root controller and transform the inner controls of mobl screen to regular mobl-ios controls. This solution also brings some challenges because of the limitation of combining views in iOS. For example, UISplitViewController (see 4.1.1) can be used only as the root controller and cannot be placed inside other controls. This controller has been used.
4. iOS Native Applications as a New Target for mobl

by the "masterDetail" mobl-ios class. In mobl, "masterDetail" is a control to show the master-detail items. In iPad, the master-detail information is represented using UISplitViewController. Then, it is expected to map masterDetail control to UISplitViewController. But, when we have a masterDetail control inside a screen, in the generated code, since a general root controller (UINavigationController has been chosen as the general root controller) is created for the screen, the view of the corresponding UISplitViewController should be located inside the root view which is not possible. In order to solve this solution, we have defined a custom controller to present master-detail data in mobl-ios. However, in the implementation of this controller, we use all the facilities of UISplitViewController. As a result, the resulting view has almost the same view and the same behavior as the original view. Figure 4.12 shows two views, one created by iOS split view predefined controller and one created with the mobl-ios custom split view controller. The differences are highlighted although they can be removed in the future. As can be seen, the border of the sections in the default split view is curved while in mobl-ios one is not.

Custom Control

In addition to the external controls that are transformed to the predefined mobl-ios controls, a mobl developer can develop her own controls by combining the predefined controls and
Figure 4.12: The iOS default split view (bottom image) vs. mobl-ios split view (top image).
other user interface elements. In order to implement this feature in the iOS back-end, a "customControl" wrapper class inheriting from "baseControl" has been developed in mobl-ios. Neither an iOS view nor an iOS controller has been connected to this class and this class only acts as a virtual container for its inner controls. In the code generation phase, Corresponding to each custom control, a class is generated which inherits from "customControl". Similar to the screen, the implementation of the body of the custom control is generated as the "render" method of this class. The only difference between screen and custom control is that in screen, the inner controls are added to its associated controller but, in the custom controls, no control is associated and the inner controls are added to the parent of the custom control. Similarly, if the parent is also a custom control, the inner controls are automatically passed to the next ancestor. This mechanism makes the generated user control more efficient by decreasing the depth of the inner controls. Although, the structure created in mobl-ios actually is not the same with the mobl program, to simplify the control manipulation such as styling, showing/hiding etc. the original hierarchical structure is kept in a special data structure. Figure 4.13 shows a sample code of creating custom controls in mobl. Figure 4.14 shows the resulting iOS output.

As can be seen in figure 3.7, a set of additional user interface elements can be also passed to a control as elements control. The elements control can be invoked anywhere in the body of a control. In order to implement this feature in Objective-C, we treat the elements control as a regular control. Therefore, a custom control corresponding to that is generated in the code generation phase. An instance of this custom control is passed to the control in which the elements should be rendered. The invocation of the elements control will be translated as calling the render method of that custom control.

Figure 4.13: A custom control sample.
Modal View

Modal view controllers are the iOS facilities to provide a temporary interruption to obtain key data from the user. Different types of controllers such as UITableViewController,UITabBarController etc. can be shown modally. In iPad applications, the size of modal view can be also customized by setting the modal view style. Modal views are not supported in the web mobl. But in mobl iPad library, we have provided an external function to show modal contents. This function accepts a custom control as its argument. Using this function, the custom control argument will be shown modally. Figure 4.15 shows a sample code using modal views. The resulting output can be seen in figure 4.16.

Closure

As explained in chapter three, one of the mobl features is that the functions are considered as a first-level data structure. Since the controls have the function structure, they are also treated as a first-level data structure. Thus, functions and controls can be defined seamlessly in the flow of the code. Figure 3.9 shows a sample of a nested control. As mentioned in section 4.1.3, introduced in iOS 4.0, Objective-C also supports closures by providing basic blocks. One option to generate the Objective-C code for the mobl nested controls is to use basic blocks. In this case, the creation of a control should be wrapped into a method. If a
4. iOS Native Applications as a New Target for mobl

control modal(){
    header("New Person"){
        button("Done", onclick={
            //some logic
            UIUtil.dismiss(); //hide modal view
            //if it is not in
            //modal view, has
            //no effect.
        })
    }
}
group{
    section("General Info"){
        itemEdit(text="", placeholder="John", label="First Name:")
        itemEdit(text="", placeholder="Smith", label="Last Name:")
    }
    section("Contact"){
        itemEdit(text="", placeholder="john@smith.com", label="Email:")
    }
}
}
screen root(){
    header("User"){
        button("Add", onclick={
            UIUtil.dialog(modal); //To show the modal view.
        })
    }
}

Figure 4.15: A modal view sample code.

control is defined as a nested control, this function should be defined as a basic block. The
Objective-C implementation of closure is promising. It means that reading and modifying
the variables of the outer scope of a closure is possible because those variables are preserved
for the life of all copies of the block.

Although using Objective-C blocks is a good strategy to implement mobl closures, we
use our own implementation of the closure. The reason of this decision is that we actually
deal with closures in the compiler front-end component instead of the back-end component.
In fact, by traversing the AST, we find all the nested controls in any level and lift them up to
the first level. Then, using the scope manager component provided in mobl-ios (see figure
4.5), accessing to the context variables of those controls is handled. The advantage of this
approach is that since the nested structures are removed in the front-end part, the resulting
model has a higher degree of portability. Imagine that we want to port mobl to another
target platform which does not support closure or its closure support has some limitations (such as Java). In this case, this normalized model can be used as a source AST. By re-implementing the scope manager (which is a very small component), the new target will also support closures completely.

The mechanism is that after desugaring phase, the whole AST is traversed and all the nested controls are found. Afterwards, to prevent repeated names, a unique name is assigned to all of them and the references to their name are updated to their new name. Then, every nested control is lifted up to the first level of the context module and the corresponding Objective-C code is generated like a regular custom control. The only thing that remains is accessing to the context variables. For this purpose, a data structure called "scope" has been defined in the mobl-ios core services layer. This data structure consists of a dictionary to preserve the context variables. Corresponding to each variable, an entry is added to the dictionary. The name of the variable is the key of that entry and a reference to its memory location is the value of that entry. Additionally, each scope instance preserves a pointer to the scope of its outer context. Figure 4.17 shows a class diagram of this data structure.

Every control in mobl-ios keeps its context scope variables using a scope property.
When a variable is defined in the body of the control, the corresponding dictionary entry will be added to the control scope. Moreover, whenever another control is called via the body of the current control, by calling the "createInnerScope" function of scope property, a child scope corresponding to the inner context is created and its parent is set to the scope of the current control. This new scope is passed to the control that being invoked and assigned to its scope property. The scope class provides some functions to access to the scope variables. If the control body needs to access to a variable, the "get" method of the scope property is invoked and the name of the variable is passed to it. To find the variable, the scope instance first looks at its dictionary. If the variable is found, it will be returned. Otherwise, the get method of the parent scope will be called. This look up process continues in ancestors until the variable is found.

As can be seen, by lifting up the inner controls and using the scope data structure, an inner control can access to both the local variable and the upvalues (the variable defined in its outer contexts).

Callback is another construct in which scope variables should be managed. A mobl callback is an inline function which does not accept any argument and it is mostly used as a handler for a control event. Instead of using the arguments, a callback can access to its outer scope variables. For example, a "button" control receives a callback as its handler of its click event. A sample of invocation a button with a callback can be shown in figure 4.18.

The callbacks can be also defined seamlessly in the code flow and they can access to their outer scope variables. As a result, a callback should preserve its context variables. For this purpose, a "Callback" base class has been defined in mobl-ios core services layer which has a property of type scope to keep the context variables. Corresponding to each callback, a subclass of Callback class is generated in the code generation phase. The logic of the callback is implemented in an "action" method. Whenever a callback is defined and passed to a control in a mobl program, in the Objective-C code, it is considered as an inner
screen root()
{
    var items = [1, 2, 3]

    group{
        list (n in items){
            item(onclick={
            UIUtil.message(n.toString(), "Content");
            }){
                label("Item" + n)
            }
        }
    }
}

Figure 4.19: A sample of using callback in a loop.

scope and similar to the control invocation, an inner scope is created in the current context and passed to the instance of the corresponding callback class. As a result, by keeping the context variables as a property of the callback, it can access to its context variables even after it is destroyed. A similar approach is used for other kind of closures such as functions.

Figure 4.19 shows a sample of callback creation inside a loop. Using this code, according to each number in a list, an item is created in a group control. By clicking that item, the associated number should be shown as a message box. In this case, the iteration causes the change on the context variables. It means that in each iteration, the variable "n" holds a new value. Therefore, in the loop, an inner scope should be created and the current iterator should be added into it as a variable. Then, this inner scope is passed to the callback instance as its context variables snapshot. Now, in the callback "action" function, looking up for the variable "n" via scope property will result in finding the iterator of that iteration in which the callback has been created. Obviously, these variables are accessible even after the loop is finished. The iOS output of the code shown in figure 4.19 can be seen in figure 4.20.

Derived Variables and Data Binding

As mentioned in 3.3.1, mobl provides the reactive programming using derived variables and the controls data binding. In order to implement derived variables in Objective-C, a wrapper class called "BindableObject" has been provided in the mobl-ios core services layer. Every variable declared in the user interface context or returned from the logic context to the user interface context is wrapped by an instance of BindableObject. BindableObject has a property called "value" of type "NSObject" which is the base class of all objects in iOS. This property is used to keep the value of the variable that is wrapped. In addition, BindableObject has an array for its listeners. Whenever the value of a BindableObject...
Figure 4.20: The iOS output of the code shown in figure 4.19.
is changed, all of its listeners will be notified. The listeners of a BindableObject can be either other BindableObjects or the controls. The former is used to implement the derived variables and the latter is used to implement the control data binding. For this purpose, we have used the protocol concept of Objective-C. Similar to programming interfaces in high-level general purpose languages such as Java and C#, Objective-C provides the protocol construct. Basically, a protocol is a set of function signatures. Every class which is willing to follow a protocol should implement its functions. For mob-ios notification process, we have also defined a protocol called "Notifiable" in the mobl-ios core services layer which has a changeNotification function. baseControl and BindableObject are two classes that follow this protocol by implementing changeNotification function and thus, they can be used as the listeners of a BindableObject. When the value of a BindableObject is changed, it will iterate on its listeners and call their changeNotification function.

**Derived Variable.** A derived variable can be initialized using an initial expression consisting of some other variables. For example, Figure 4.21 shows a sample in which a function call as an initial expression has been bound to a date derived variable. This function uses three variables as its arguments. Consequently, whenever any of these variables are changed, the date variable should be notified to update itself. By updating the derived variable, the date control receiving that as a parameter also updates itself. Figure 4.22 shows this behavior. In this sample, as can be seen in the code, if the "Next Day" button is clicked, the "day" variable will be incremented and since it is bound to "date" variable and "date" variable itself is bound to a view, the view will be updated. In fact, the date variable is a listener of those three variables.

Since these variables are defined in the user interface context, in the code generation phase they are wrapped by BindableObjects and the date variable which is itself a BindableObject is added to their listeners. Whenever any of those variables are changed, they will notify the date variable. Now, the date variable should re-evaluate its initial expression to update its value. For this purpose, a class called "Evaluator" has been defined in mobl-ios core services layer. In the code generation phase, whole AST is traversed and corresponding to each initial value of a derived variable, a subclass of "Evaluator" is generated. The implementation of this initial expression is placed in the "evaluate" method of that class. Obviously, in order to access to the context variables, a reference of the current scope is passed to Evaluator subclass instance. Since the scope data structure keeps the references of the context variables, automatically it will access to the most updated value of them. The Evaluator subclass instance is kept as a property in the BindableObject of the derived variable which in our example is date variable. Now, whenever the derived variable is notified about the changes of the variables of its initial expression, it will re-evaluate the expression by calling evaluate function and update its value.

**Control Data Binding.** According to the previous chapter, the control-value two-way data binding is another feature of mobl to support the reactive programming. In mobl, when a control is defined as an external control, its data binding should be handled in the target platform. The implementation of the value-to-control data binding using "Notifiable" protocol is straight forward. Whenever a BindableObject is passed to a control, since controls
control setDate(){
    var year = 2011
    var month = 8
    var day = 1

    var date <- DateTime.create(year, month, day) //date is a derived variable.

    button("Next Day", onclick=
        day = day + 1;
    ), style = customButtonStyle) //style is defined somewhere else.

    datepicker(date)
}

screen root(){
    header("Screen"){
        button(text="Button", contextMenu=setDate)
        //context menu is realized as popover control
        //in iOS.
    }
}

Figure 4.21: A derived variable sample.

follow Notifiable protocol, they can be added to the listeners of that BindableObject. As a result, the listening control is notified about the variable changes and thus, the layout will be updated.

BindableObjects are also used for the other way around data binding. We explain this mechanism using an example. In mobl, "textField" (in iOS mobl, "textBox" is also supported) is a predefined text input control. It receives an argument of type String as its text. In the Objective-C side, the BindableObject instance which is passed to the control as its text is preserved in a property called "textBO". In iOS, the action methods can be added as the targets of the control events. Whenever an event is raised, these methods are responsible to handle that event. In the implementation of textField in mobl-ios, these handlers are used to propagate the user interface changes to the corresponding BindableObject property which in our case is textBO.

With this mechanism a two-way data binding is achieved. Figure 4.23 shows a concrete example of the control data binding. As can be seen, there are two input fields which are bound to one single variable. As a result, editing each of those fields updates the "content" variable value. When this variable is updated, another input field will be notified and then, its value will be synchronized. Figure 4.24 shows this behavior.
Figure 4.22: The iOS output of the code shown in figure 4.21.

screen root(){
    var content = ""
    //two text fields are bound to
    //one variable.
    textField(content)
    textField(content)
}

Figure 4.23: A control data binding sample.
when-else Clause

As described in the previous chapter, the when-else clause is used in mobl to show a part of the user interface conditionally. Since the when-else clause is reactive and it should be re-rendered if the variables of its condition are updated, in mobl-ios, we have implemented it as a custom control. For this purpose, in the code generation process, the "when" and "else" clause are transformed to two regular custom controls with their own body. The instances of these two custom controls are passed to another custom control called "whenManager". This custom control is a special control defined in the mobl-ios user interface layer and has two properties to keep both the "when" and "else" controls. The condition of the when-else clause is passed as an argument to this control. As can be seen, now, the whenManager control is completely like a regular control. Whenever any of the variables of the condition expression is changed, the whenManager control is notified and based on the result of the condition, it renders either "when" control or "else" one and it removes and releases another one. Figure 4.25 shows the iOS output of the code shown in figure 3.13.

list Clause

Basically, a list clause is nothing more than a for loop. But, as explained in chapter three, when a list iterates on a collection, it becomes reactive. It means that if a new item is
added to the collection or an existing item is removed from that, list clause is notified and its content is re-rendered. Our approach to support this feature is exactly the same with what has been used for when-else clause. For this purpose, a customControl subclass called "list" has been defined in the mobl-ios user interface layer. This class has a property of type customControl called "listContent" which represents the body of the list. Therefore, in the code generation phase, first the whole AST is traversed and corresponding to each list clause, a customControl subclass to represent its body is generated. The instance of this custom control is passed to the list instance and assigned to its "listContent". Additionally, the list control receives the collection on which the iteration is done as an argument. Using the BindableObject mechanism, the list control will be notified about any changes occurred in the collection and thus, its layout will be re-rendered. It should be mentioned that re-rendering in mobl-ios means that the old layout is removed from the screen and the new one is created. This mechanism needs precise consideration about the controls positioning which will be described later. Figure 4.26 shows the iOS output of the code shown in figure 3.15.

**Control Variables**

In mobl, the controls can be assigned to the variables. In order to be rendered on the screen, a control variable can be invoked wherever through a mobl program. Since the controls
are transformed to classes in the Objective-C side, clearly a control variable is nothing more than an instance of that class. The invocation of the control variable is also easily transformed to the invocation of its "render" method. As described in section 3.3.2, based on the signature of the control, they have different types. Similar to the web mobl, we also use type erasure approach to transform these types in the iOS target code generation phase. It means that, the generic type information of the controls is not available in the generated code.

The control variables are reactive. It means that if a new control is assigned to a control variable while the old value of that control variable has already been rendered, the old control component should be replaced with the new one. Figure 3.17 shows this concept. Similar to other variables in the user interface context, a control variable is also wrapped by a BindableObject. Whenever the value of this BindableObject has been changed, it calls styling manager component to apply the replacements and re-order all the controls. This mechanism is described next. As can be seen, using the reactive control variables, a mobl developer can easily restructure the user interface layout regardless to the target platform. The iOS output of the code shown in figure 3.17 can be seen in figure 4.27.
Styling

Since the styling language of mobl is actually built on top of CSS, porting it to another platform is challenging. An idealistic solution in order to have a high degree of portability is to build a CSS engine for the iOS native applications. Using this engine, the layout of an iOS application is adjusted based on the CSS specifications. This approach has been applied to some extent in DIMAG. However, because of the differences in the user interface concepts between the web and native applications, this approach cannot be promising. For example, using CSS, every part of a web user interface can be customized easily. On the other hand, in a native application, in order to force the developers to follow the platform standards, the vendors try to impose a set of restrictions on the widgets customization. Consequently, the styling specifications declared for the web target cannot be ported completely into the iOS target.

On the other hand, although in mobl-ios, it is very expensive to simulate all of the CSS tags, a small set of them can be used to specify the size, the position and some other simple properties such as background color, border style etc. It means that a user can use the styling syntax of mobl with a limited set of tags to customize the layout. Additionally, we provide two more tags called "place" and "anchor" for the iOS target. Using these two tags, the user interface components can be placed on the screen without any need to their absolute positions. The "place" tag accepts two values: "currentline" and "nextline". Using these
two values, the designer can locate a component either in front of the previous component or below that. The "anchor" tag accepts three values: "left", "right", "leftright". If the anchor is set to the left, then the control position should be started from the left side of the screen. For the right anchor, the control position is started from the right side. For the leftright value, the control is stretched between the controls with the left and right anchor in the current line. Although by using these values, the user has flexibility to design her user interface, for more complex cases, she can use absolute positioning. Figure 4.28 shows the usage of our positioning system in some samples.

In order to manage the user interface styling in the Objective-C side, in the mobl-ios user interface layer, a base class called "UIStyle" has been implemented. This class has a set of predefined fields corresponding to those styling tags that are supported in mobl-ios. In the code generation phase, for each style definition, a UIStyle subclass is generated. In the "initialize" method of this class, the values of the styling tags specified in the mobl declaration are assigned to the corresponding styling fields. In this version of mobl to iOS compiler, any new tags are ignored to translate. Since in desugaring phase, the content of the mixin styles are copied to the style declaration from which the mixin style is invoked, there is no need to translate the mixin styles directly.

In mobl, the styling specifications are passed as an argument to the controls. Similarly, in the generated code, an instance of the UIStyle subclass is passed to the control. As can be seen in figure 4.5, there is a styling manager component to adjust the controls.
based on their styling specifications. When all the controls are located on the screen (before making the screen visible), the styling manager component is invoked to order the controls. Generally, the styling manager is responsible for the positioning of the controls. Since our positioning system is relative, we prefer to order the controls all at once to simplify the process. Other control-specific styles such as border style (for textField) etc. are done by the controls themselves.

One another advantage of doing positioning all at once is that it simplifies the re-ordering of the controls when their locations are changed. For example, using when-else clause, a part of the user interface may be hidden and thus, all the controls located below that should be lifted up. Using our relative positioning mechanism, these adjustments are done automatically. The only thing needed is to call the styling manager to re-order whole screen. This mechanism is also useful for the control variables. As described before, although mobl-ios does not consider any actual view for the custom controls, the hierarchical structure of the controls is kept in mobl-ios as it has been defined in mobl. This structure helps the controls replacement. The control replacement is needed when a new control is assigned to a control variable. In this case, the styling manager component finds the old control (or its inner controls if it is a custom control) and removes it from the screen. Then, the new control is located exactly at the place of the old control. Finally, using the styling manager, whole screen is re-ordered. It should be mentioned that this replacement procedure does not make the control hierarchical structure inconsistent.

4.2.2 Application Logic

The application logic context of mobl has an imperative nature and thus, the transformation of that to a general purpose language such as Objective-C is not difficult. In our mobl-to-Objective-C compiler, first, corresponding to each module, a class is created. All the functions defined in that module are transformed to the static methods of this class. Since Objective-C does not support the module context (e.g. package in Java, namespace in .NET), using classes instead makes the function resolving easy. All the mobl APIs have been also implemented in the appropriate classes. For example, "mobl" module is the core module of mobl which provides different APIs. In the Objective-C side also, we have implemented a "mobl" class and all those APIs have been implemented there.

In the current version of mobl-ios, the asynchronous function call is not supported; however using GCD described in section 4.1.3, the realization of this feature is not difficult.

Typing System

Similar to mobl, Objective-C is a statically typed language but without type inference. In turn, Objective-C provides a special pointer called "id" which is similar to void* in C. Consequently, whenever a variable is declared in mobl without the explicit type declaration, its type will be transformed to "id" in the Objective-C side. On the other hand, if the type is explicitly mentioned, by using type transformation mechanism that provided in our code generator, its type will be transformed to an appropriate destination type. Now, imagine that we have declared a variable of type id which is implicitly of type "X" and we want to
pass it to a function that accepts an argument of type "X". In the languages such as Java which do type checking at the compile time, this task can be done using casting. But, since Objective-C provides a message passing mechanism with dynamic binding, any message can be sent to any object. If the object can respond to that message, everything will be fine; otherwise an exception will be thrown at runtime. By this mechanism, there is no need to cast our id variable to type "X" since it is not checked at the compile time.

Since the data binding is only available in the user interface context, in the application logic, the variables are not wrapped by BindableObjects. If a variable is passed from the user interface context to the application logic, only the value of the corresponding BindableObject will be passed. Similarly, the variables returned from the application logic to the user interface context are wrapped by BindableObjects after their return. This separation between the application logic context and the user interface context decreases the amount of memory usage.

As described before, mobl provides a set of predefined data types such as Num, String, Bool etc. Logically, these basic data types should be transformed to the primitive data types of the target platform to keep the efficiency. Unlike some languages such as JavaScript and C#, the primitive types in Objective-C are not objects. As a result, they cannot be added to the arrays, they cannot be declared using id, they do not inherit from NSObject etc. The latter issue causes many problems in the user interface context. Since the "value" field of the BindableObject class is of type NSObject, none of the primitive types can be assigned to that. In order to solve this problem, Objective-C provides a wrapper class called "NSNumber" to offer the primitive type as the objects. Although using NSNumber is a good alternative solution for our problem, since after initializing an NSNumber, it is not possible to recognize its initial type, using NSNumber causes a set of limitations. Therefore, we have implemented our type wrappers in the mobl-ios core services layer. Num, Bool etc. are the wrapper types that have been created there. Choosing the same name between these types and the mobl types makes the transformation process easier. In addition to these types, all other mobl predefined types such as Tuple (a row of data consisting of different fields), Object, Array, Collection, DateTime etc. have been also wrapped in the mobl-ios core services layer.

Similar to the web mobl, using the type erasure method, the generic type information is removed when the code is generated. Moreover, the APIs that the predefined types provide have also been implemented in those wrapper classes. The only exceptions are String and Object. In the mobl-ios core services layer, instead of wrapping NSString which is the iOS string type, we have extended the existing NSString. In Objective-C, using the category concept, it is possible to modify an existing class without inheriting from that. The advantage of using category is that by modifying the existing class (e.g. by adding new functions), the name of the class will be kept. But if the sub-classing is used, it is needed to refactor whole the legacy code and all the variables of type base class should be changed to the variables of type sub-class to be compatible with the new functionalities. This refactoring process may be expensive especially when the IDE does not support refactoring well. We have also used the category mechanism to extend NSString and add the mobl APIs to that. Since firstly, we used NSString without implementing mobl APIs, to achieve backward compatibility with our legacy code, we have decided to extend the existing NSString.
Furthermore, since according to the type hierarchical structure defined in mobl, every type inherits from Object and NSString inherits from NSObject and cannot inherit from a wrapper class, we have also extended existing NSObject for Object type of mobl.

Using the wrapper types has their own disadvantages. For example, if you want to add three numbers and store the result into a variable, you have to instantiate four Num objects. This makes the memory usage inefficient; however, most of the languages that offer primitive types as objects have the same mechanism. In the code generation phase, a set of optimizations can be done for this problem. For example, it is possible to find the intermediate calculations and evaluate them using the primitive types. These types of optimizations have not been applied in this version of our mobl to Objective-C compiler.

Similar to the mobl predefined types, corresponding to each user defined type, a class is generated in the code generation phase which we call "type class". The predefined base class for whole types is the extended version of NSObject which is a wrapper class for the mobl Object type. If the mobl user defined type has another base class, it will be considered when the code is generated. As a result, the type hierarchical structure of mobl is preserved in the generated code. Clearly, the type functions are also implemented as methods in the type class.

In order to simplify the data binding mechanism between type class fields and the user interface widgets, all of the fields of a type class are defined as BindableObject. With this mechanism, there is no difference between a type field and a regular variable and both of them can be bound to the user interface views in the same way. Using this strategy, access to a type class field should be done differently based on the current context. In the user interface context, the BindableObject is needed and thus, invoking a type class field should return its corresponding BindableObject. On the other hand, in the application logic context, instead of the BindableObject, the actual value of the type class field is needed. Therefore, in the application logic context, invoking a type class field should return the actual value and not the wrapping BindableObject. These decisions are made in the code generation phase and according to the current context, appropriate code to determine how the type class field is accessed will be generated.

Reference Resolving

As can be seen, for different mobl constructs such as the types, callbacks, modules etc. different classes are created in the Objective-C side. Since Objective-C does not have the code classifying mechanism such as Java packages or .Net namespaces, different classes cannot be accessed directly or through their qualified names. Therefore, in order to access to other classes, their header files should be imported in the requesting class. For this purpose, before starting the code generation, every constructs that are mapped to the classes are traversed. For each of them, all the qualified names to which they refer are found. Using these qualified names, the import statements of each class are generated.
4. iOS Native Applications as a New Target for mobl

4.2.3 Persistence

As described in section 3.3.3, entities are the persistable data types in mobl language. mobl aims to handle the persistence of these entities automatically. As a result, in the code generation phase, all the entities are traversed and according to their structure, appropriate data storage is created. Mapping the information stored in that storage and the data structures representing entities should be done automatically. Based on the target platform, different persistence management strategy can be used. For example, in web mobl, the HTML5 storage facilities are used to store entities on the device. As explained in section 4.1.3, iOS provides Core Data framework as a technology to manage the data model of the iOS applications. In iOS back-end of mobl, we have used this technology for data management.

In the Core Data framework, every object to be persistable should inherit from "NSManagedObject". The only difference between a managed object and a normal object is that an entity description (NSEntityDescription object) should be associated to a managed object in order to describe it properties. In fact, this entity description specifies the name of that entity, the class (subclass of NSManagedObject) used to represent that entity and the properties of that entity. Core Data framework uses this description to create appropriate databases and tables. As mentioned before, the Core Data framework uses SQLite as its underlying database and thus, the entities are stored as relational data. Using the entity description, the mapping between these relational data and the corresponding managed objects are done automatically.

In mobl to Objective-C code transformation, first, all the entities are traversed and the related classes are generated. For simplicity, we call a corresponding class of an entity "entity class". Every entity class should inherit from a mobl-ios core class called "Entity" (see figure 4.5). Similar to types (see 4.2.2), in order to facilitate data bindings, all the properties of an entity class are generated as BindableObjects.

As mentioned earlier, in the Core Data framework, a class can be persistable if it inherits from NSManagedObject. In addition, an entity description should be associated to that. Since we have used BindableObjects as the properties of the entity classes, this makes the object-relation mapping complicated. Consequently, beside an entity class, a small class inheriting from NSManagedObject is also generated. This class has only the properties of the original entity class but with their actual data types. Whenever, an entity should be persisted, its data will be copied to that auxiliary class and the persistence done using the Core Data framework. This mechanism is the same for data retrieval. Additionally, an entity description should be associated to the auxiliary class. For this purpose, a function called "createPersistence" is created in the entity class. Moreover, a class called "AppInit" is also generated which calls the "createPersistence" method of all entity classes to initialize the data storage. When the application is launched, the "init" static method of "AppInit" will be invoked to build the storage. If the data storage has already been created, Core Data framework skips this phase automatically.

Another alternative to generate the entity description is to load the auxiliary class at runtime and read its properties. Since this affects on the performance of the application, we have preferred to generate a static code at the compile time using the information obtained from AST. Obviously using auxiliary class for an entity class makes the data persistence and
retrieval expensive. However, we use all the lazy loading facilities of Core Data framework to defer loading unnecessary data. Although these strategies have been used to make the data management more efficient, there is still much room for improvement which can be considered in the future work. Figure 4.29 shows a schematic view of the Objective-C code generated for the entity declared in figure 3.24. As can be seen, ”sync” and ”reSync” methods are responsible for converting the entity class to the auxiliary class and the other way around. Moreover, the Core Data framework cannot work with primitive data types and this is the reason why NSNumber has been used as the type of ”age” field.

The Core Data framework provides a special class called ”NSFetchRequest” in order to retrieve the stored data. For this purpose, an entity description should be assigned to an ”NSFetchRequest” object. The desired criteria can be defined using an ”NSPredicate” object and passed to that ”NSFetchRequest” instance. By calling the ”executeFetchRequest” method of this instance, the filtered list of the entities will be loaded. As can be seen, conceptually this mechanism is really close to the ”Collection” APIs described in section 3.3.3. Consequently, mapping the Collection APIs to ”NSFetchRequest” is not complicated. Using this mapping, similar to the web mobl, the iOS mobl also provides the automatic persistence and retrieval facilities in a robust way.

![Diagram of Objective-C code generated for Person entity declared in figure 3.24.](image)

Unlike web mobl which pushes the object changes into storage in a certain time intervals, in the iOS version, any changes occurred in objects are pushed to the storage immediately. Although the mechanism can be heavy, it guarantees that the in-memory data are synchronized with the stored data. Since the properties of the entity classes are defined as BindableObject, they can notify persistence components about their changes and using persistence APIs, these changes are applied to the data storage.

In this chapter we introduced the iOS back-end of mobl. Using this back-end, the data
4. **iOS Native Applications as a New Target for mobl**

Centric iOS applications (specifically for iPad) can be developed using mobl language. In the next chapter, we will evaluate our work.
Chapter 5

Evaluation

In this chapter, in order to evaluate our new back-end for mobl, we will try to port an existing mobl application developed for the web target to the iOS target. Using this approach, we can find out which concepts of mobl are abstract enough to be ported easily to the new target and which ones have been designed specifically for the web target. Additionally, the code adjustments needed to capture the specialties of the new target can be analyzed. Obviously, those mobl features such as consuming web services, asynchronous code execution etc. which have not been implemented yet for the iOS target have been excluded from our evaluation.

For this evaluation, we have chosen "Shopping List" application presented in [5]. In order to cover more features of mobl and increase the complexity of the application, we have extended the scope of original Shopping list application. The complete mobl source code of this application can be found in appendix B.

5.1 Shopping List

Shopping list application is a handy application in order to make categorized list of items that should be ordered. Defining different shopping items, categorizing them into different custom groups, and searching through the shopping items are some of the major capabilities that the user will achieve by using this application.

The user interface of this application is constructed using different screens. The root screen has a master-detail structure. The categories are loaded into the master section. By selecting each category, the shopping items of that category will be presented in the detail part. There is also a predefined category called "All". By selecting this category, all the shopping items regardless to their category will be loaded in the detail part. It is also possible to search through the shopping items loaded in the detail section.

As described in section 4.2.1, mobl provides a "masterDetail" predefined control to visualize master-detail data. Similarly, for the iPad device, iOS provides a UISplitViewController which is appropriate to present master-detail data. Thus, we have implemented mobl-ios "masterDetail" wrapper class using the custom version of UISplitViewController. Figure 5.1 shows the web and iOS output of the root screen of the Shopping list applica-
5. Evaluation

Figure 5.1: The web (top image) and iOS (bottom image) layouts of the root screen of the Shopping list application.

As can be seen in figure 5.2, by selecting another category, the shopping items of that category are loaded into the detail section.

In order to generate the iOS layout for this screen, no adjustment has been done in the mobl code. The only minor difference which can be seen is about the location of the "Add Item" and "Clean" button. For the iOS version, these buttons have been located in a "footer" control. "footer" control is a simple control defined in the iPad control library of mobl, but it is not accessible as a generic built-in control. The only reason to apply this change is to
Figure 5.2: The web (top image) and iOS (bottom image) layouts when the selected category is changed.
make the iOS layout more similar to common iPad applications. Obviously, the “footer” control can be also implemented easily for the web target. It should be mentioned that since this application has been originally built for the mobile devices, the resulting layout has empty spaces in the iPad device.

Figure 5.3 shows on demand search capability of the application. The shopping items presented in the detail section are filtered based on the search term that the user enters in the search box. In addition to the search term, the shopping items are filtered according to the selected category. Clearly, if the user is willing to search through whole shopping items, the ”All” category has to be chosen. Figure 5.4 shows this option. All the search processes are done using ”Collection” APIs of mobl described in section 3.3.3. As described in section 4.2.3, these APIs have been implemented in mobl-ios back-end using iOS Core Data framework.

A new category can be created using ”Add Category” button on the header of the screen. If the user is willing to clear all the shopping items, he can use ”Clean” item. Using ”Add Item” button, the user is navigated to another screen in order to create a new shopping item. Basically, the new shopping item is created in the category that has been already selected in the root screen. If the new shopping item does not belong to any category, the user can select ”All” category. Figure 5.5 shows the ”Add Item” screen. Moreover, by tapping on each shopping item, the user is navigated to ”Edit Item” screen where he can edit different attributes of a shopping item. In this screen, an additional item called ”Is Ordered” can also be found. If this item is checked, it means that the item has already been ordered and thus, it no longer appears in the shopping list items.

As can be seen, this application has the set of features that the majority of mobile data centric applications provide. Showing master-detail data, searching through items, creating entities, modifying entities, navigating through screens etc. are some of those features that can be seen in these types of applications.

Now, we continue with the negative sides of porting an application from the web to iOS. As described before, mobl has been originally designed to cover the mobile web applications. As a result, several features which are compatible with the web environment have been embedded into that. For example, mobl allows the developers to use HTML tags and also JavaScript code directly in their mobl code. Figure 5.6 shows a possible way to increase the importance of a label. If we want to handle the HTML tags for the iOS back-end, an HTML parser engine should be embedded in mobl-ios to map the HTML tags to the equivalent concepts in iOS. Clearly, this is not an easy task and in some cases it is not possible at all.

It can be concluded that mobl has a set of dependencies to its original target environment. Although these dependencies may help the developer to have more flexibility, it causes re-targetability side effects. In fact, using the target environment facilities in the DSL decreases its degree of portability. The degree of portability is directly related to the abstraction level of the language. By increasing the abstraction level, irrelevant details are replaced by high level concepts. Since these high level concepts do not have any dependencies to a specific technology or platform, they can be realized in different targets and as a result, they can be ported into various platforms. But usually determining the abstraction level that fulfills the requirements of a language is not straight forward. For example, defin-
Figure 5.3: The web (top image) and iOS (bottom image) layouts for searching through the shopping items of the selected category.
Figure 5.4: The web (top image) and iOS (bottom image) layout for searching through all shopping items.
Figure 5.5: The web (top image) and iOS (bottom image) layouts to create a new shopping item.
5. Evaluation

```java
screen aScreen()
{
    <h2>label(daysOfWeek.get(day.getDay()))</h2>
}
```

Figure 5.6: Using `<h2>` HTML tag in mobl source code.

ing a high level, cross-platform user interface declaration language has been one of the most challenging problems in the model based user interface design studies. Most of approaches in this area define a set of abstract user interface elements such as input, check, select etc. Based on the target platform, these abstract elements are mapped into the concrete ones. But this approach is also not promising since different complex user interface concepts should be shown differently according to the target platform. For example, the master-detail concept can be shown in one screen or multiple screens according to the screen size of the target device. If in the user interface declaration, a master-detail control is described using basic components such as a composition of a list and a grid located in one screen, it cannot be realized in the devices with small screen size such as smart phones. Nilsson in [32] has introduced the pattern oriented user interface declaration. For example, the master-detail relation is a pattern and it is abstract enough to be realized in different targets. However, the realization can be varied according to the target device characteristics. It means that instead of building master-detail using other basic elements; it is worthy to have an abstract master-detail control as a single AIO.

Presenting a list of editable items is another sample where determining a correct level of abstraction can ease the portability. As described before, mobl provides "group" control to show a list of items. Basically, in order to edit the items in the list, "Edit" and "Delete" buttons can be located as the elements (see 3.3.1) of each item. As can be seen, using the basic user interface controls, a composite user interface element can be built in mobl. On the other hand, in iOS, "UITableView" is provided to present a list of editable items. In this case, the items of the table can be modified or removed when the style of the table is set for editing. Thus, instead of building a composite control using the basic user interface elements, iOS provides a high level control supporting various features. In fact, iOS looks at listing items as a general pattern and realizes this pattern in a very robust way. Every different aspects of this list view such as inline modification or item removal are considered as a part of this pattern and not as the separate concepts.

Moreover, since this robust user interface element fulfills most of the user’s requirements, the developer is automatically prevented to create his own table and instead, he uses the default one. Therefore, the resulting application meets the iOS standards.

In mobl, on the one hand, the existing flexibility enables the developers to define these types of high level controls even as the external ones. On the other hand, again by this flexibility, the developers have a higher degree of freedom to create their own composite components which causes portability side effects.

Another issue which should be taken into consideration is the adaptation of the resulting application to its target platform. Although a DSL, to be re-targetable, should be abstract
enough to capture commonalities between various platforms, in order to generate an application which follows the standards of the target platform, the DSL should also provide the capabilities to capture the specialties of the destination platform. As mentioned earlier, in the user interface context, mobl has solved this problem by providing different user interface libraries for different platforms. However, using the platform specific user interface elements complicates the re-targeting process. In case of using these types of widgets, to port the application to another platform, the mobl code should be modified. This modification can be from changing a simple user interface element to changing the user interface design of whole screen or even whole application. For example, in the original version of the "Shopping list" application presented in [5], to edit a shopping item, the user should choose "edit" item from a special user interface element called "contextMenu". Figure 5.7 shows this control in the web mobl. The design of this control has been inspired by Android context menu widget which can be seen in figure 5.8. iOS does not support this type of context menu. Instead, it supports "UIPopoverController" which can be seen in figure 4.27. This control can be popped up on other controls especially buttons. Thus, in mobl button API for iPad, we have provided an optional argument of type custom control called "contextMenu". If a custom control is passed for this argument, it will be shown inside a UIPopoverController. However, the usage of this control is not the same with the usage of the context menu in Android. If each of these two concepts is used in a user interface declaration, in order to port to another platform, that user interface should be re-designed. Then we will have different mobl code versions for different platforms which make the maintenance process difficult.
We have also applied some code adjustments in our Shopping list application for the iOS target. For this purpose, we have used "section" for the master part. Additionally, because of the wider screen of iPad, in the "Add Item" screen, there are a lot of empty spaces which make the user interface ugly. As a result, instead of a complete screen, we have used a modal view in order to define a new shopping item. These adjustments can be seen in figure 5.9.

Apart from the user interface context, the target based code adjustments might be needed in other APIs. However, in the logic context, it is easier to wrap different APIs of different platforms with a general interface. In addition, it is possible to implement the missing APIs of a special platform since it is not something related to the standards of an application. These approaches have been described in section 2.6 and 2.9.

In conclusion, it is possible to say that a large portion of the mobl application developed for the web target can be ported to iOS target. However, in most cases, a set of modifications needed to be done to make the resulting application familiar for the users.

5.2 A Comparison between Different Approaches

In order to have cross-platform mobile application, as our approach, we have employed a DSL to hide the differences between targets. Afterwards, for this DSL, different specific compilers for different destination platforms have been built (currently, web and iOS).

In this process, we use desugared AST of the source language. The desugaring process is done only once but the resulting model might be used for several back-ends. However, some platform specific pruning can be also done at the front-end component of the compiler. In our approach, the target code is generated using the destination software platforms.
A Comparison between Different Approaches

Figure 5.9: Master part with section and "Add Item" screen as dialog.
Consequently, it can use the high level features of the destination platform which makes the code generation process easier.

The main advantage of our approach in order to develop cross-platform mobile applications is addressing high level programming by employing a DSL. This aspect is missing in the approaches like XMLVM. Using this DSL, a clean separation between the abstract model and the concrete one is achieved. Although, in web mobl, some platform specific elements such as HTML tags can be embedded but generally it can be avoided for other platforms. This separation is missing in the approaches such as UIML. Additionally, UIML is only for the user interface declaration and the logic of the application should be implemented using another language.

By providing the external components, new transformation strategies can be added to mobl. For example, a new control can be defined as an external control. The implementation of this control can be implemented in the target platform. Using the naming convention, the generated code can use this new control properly. However, after developing a mobl compiler for a specific target platform, other transformation strategies cannot be altered. From this aspect, XIML, which provides semantics stating on the transformation rules using relations (see 2.2) is more flexible. However, XIML is only a specification and there is no engine available to perform the transformation strategies specified by XIML.

MyMobileWeb and DIMAG are not flexible enough in order to be adapted to different target platforms because they can only use CSS to customize their layout. As mentioned before, sometimes, the target specific adaptation should be done at the conceptual level according to the destination device characteristics. Using categorized libraries in mobl, this problem has been solved. However, as described before, using platform specific controls and APIs may result in re-targetability issues.

Applause is a very limited language and it is only suitable for content rendering and not for content modification. However, supporting different destination platforms makes it appropriate for cross-platform mobile applications development.

MobDSL suffers from complexity in its syntax and semantics. However, it provides robust computational model. One of the advantages of mobl is its simplicity which can speed up the mobile application development process.

PIL as an intermediate language facilitates the development process of DSL compilers. Using the intermediate language such as PIL between mobl and the target platform simplifies the compiler development and thus, porting mobl to more targets becomes cheaper.

After this evaluation, in the next chapter, we will summarize the thesis and draw some conclusions.
Chapter 6

Conclusions and Future Work

In this chapter, an overview of the project’s contributions is presented. Afterwards, the research questions of this thesis introduced in section 1.3 are discussed and some conclusions are drawn. Finally, we present some ideas for the future works.

6.1 Contributions

The contributions of this thesis are summarized as follows:

- **Building a compiler to transform mobl to Objective-C**: Basically, Objective-C is not a high-level language. A developer who wants to build an application in this programming language should take care of many irrelevant details such as pointers, allocating and de-allocating memories etc. which slow down the development process. In addition, since the smart phones popularity is vastly growing, there is a tough competition between different vendors to build mobile applications. Consequently, even without considering the re-targetability, using a high level programming language such as mobl with iOS back-end to speed up the application development process can be a key to success.

- **Achieving portability between mobile web applications and iOS applications**: As described in this thesis, an application developed using mobl for the web target can be ported to iOS platform with minimal changes. Although in most cases, a set of adjustments are needed, the advantage is that both versions of the application (web and iOS) are developed in one language. Consequently, the maintenance team members only need to know a single programming language. It should be mentioned that basically, the adjustments needed to make the application compatible with the new target are not really considerable and can be applied easily.

6.2 Conclusions

As introduced in section 1.3, this thesis is based on three major research questions. In this section, we discuss these questions.
6.2.1 What are the main challenges and difficulties in DSL re-targetability?

Basically, DSLs are built based on a set of abstract concepts. Finding appropriate mapping between these concepts and the target platform facilities is one of the most important aspects that should be considered carefully. For this purpose, the target platform standards and conventions have to be investigated. Using these studies, the constructs of the DSL are mapped to the most appropriate equivalent in the destination environment. We have enumerated several samples such as user interface elements mapping in this thesis.

Another major challenge is filling the structural gap between the DSL and its new target platform. For example, different frameworks use different models in order to declare the user interface. For example, the mobl user interface model is based on the composition of the basic elements. On the other hand, iOS provides more restrictive user interface model in which the developer can customize an existing user interface element to achieve different capabilities. For instance, in order to delete items from a "group", in mobl, it is typical to put a context menu consisting of a "Delete" button on each list item. But as described in section 5.1, "UITableView" in iOS has its own delete style. Figure 6.1 shows a UITableView in its delete mode.

In this case, although in the mobl code, a context menu is added to each table item elements, but this structure should not be built similarly in the iOS side since it does not meet its standards.

One solution to fill this gap is to design a new customized group control in mobl which accepts an argument about its state. By changing this variable (since it is bound), the control style will be updated. However, since this control is specifically designed for iOS, it has re-targetability side effects.

The next important problem which makes re-targeting a DSL difficult is that in most cases, the DSL has been inspired from its original platform. Thus, it provides some facilities of the target platform for users to increase the flexibility of the language. For example, MyMobileWeb (see section 2.4), similar to mobl, uses CSS to customize its user interface elements. Consequently, it is difficult to use this language for developing native applications.

Implementing the missing APIs in the new platform to increase the degree of portability is another problem. Because of differences in the APIs provided by different platforms, defining a set of abstract APIs is not an easy task and in some cases it is necessary to provide some APIs which are missing in a specific platform.

Emulating the programming language concepts can also be challenging. Some set of programming language concepts such as closures can be really helpful for the developers. But not every software platform supports these concepts properly. As a result, this concepts should be emulated efficiently in the target platform.

Covering the target platform deficiencies is also a problem that should be considered when a new compiler is being developed for a DSL. For example, there is no garbage collector provided in iOS and thus, the developer should take care about releasing allocated memories. Consequently, the generated code should do the memory management. However this aspect has not been focused in this thesis.
Conclusions

6.2.2 Which characteristics of DSL do have influences over its degree of portability?

As mentioned before, the characteristics of a DSL which effects on its portability degree can be summarized as follows:

- **Abstraction level of the DSL constructors**: The language constructors of the DSL should be abstract enough to be ported to various targets. On the other hand, by using very abstract concepts, the degree of flexibility of the developer in order to customize her application will be decreased. As a result, it is always a trade-off between being general and capturing specialties of the target platform.

- **Platform dependency**: Any dependency of the DSL to a specific target platform makes its retargeting process difficult. As described before, some facilities such as the ability of using the target programming language inside the DSL decrease the degree of portability of that DSL.
6. CONCLUSIONS AND FUTURE WORK

6.2.3 What are the techniques to reduce the efforts needed to map the DSL constructors to the target language constructors?

Various strategies can be applied in order to simplify the DSL to the target platform transformation process. Some of these strategies are as follows:

- **Applying early simplifications in the DSL structure:** During desugaring phase, most of the high level constructors of the DSL should be broken down into the core DSL. The resulting simplified model can be transformed to new target platforms easier. For example, in our compiler, we have removed the nested structures such as nested controls and nested functions at the model level. Consequently, in the code generation phase, there is no nested structure to deal with and this makes the code generation process simpler (As mentioned in section 4.2.1, we did not want to use Objective-C basic blocks). These simplifications can be done based on the capabilities of the target language. On the other hand, sometimes a DSL uses a high level concept which the target software platform also supports that concept. In this case, the simplification of the DSL model may complicate the code generation process. Consequently, according to the target software platform structure, the AST of DSL should be purified in order to make the code generation process easier. Moreover, applying early simplifications makes the evolution of the language cheaper. It means that new abstraction levels can be built on top of the language without any need to change the code generator. These new abstraction levels are mapped to core DSL at desugaring phase and thus, they are not appeared in the desugared AST.

- **Increasing the abstraction level of the target software platform by providing wrapper frameworks:** Generally, there is a large semantic gap between the DSL and the target language. Many of the high level concepts of the DSL can be simulated in the target platform by wrapping the existing APIs. In our case study, we have developed mobl-ios framework on top of iOS to provide mobl style APIs. For example, the user interface elements of mobl support two-way data binding. In mobl-ios framework, we have wrapped the iOS user interface widgets in order to embed data binding capability to them. Consequently, the mobl-ios user interface components support two-way data binding and by this feature the large portion of the gap between the mobl user interface components and the iOS user interface widgets has been filled. This is the same with other kind of APIs such as data persistence and retrieval, scope management etc.

6.3 Future Work

The mobl to Objective-C compiler built in this thesis can be improved in several ways. First of all, some of the mobl features have not been supported yet. These features are as follows:

- Consuming web services
- Asynchronous code execution
Future Work

- Supporting more user interface controls such as map

mobl has the potential to be a popular language because of its simple and flexible structure. If it supports more target platforms, the user’s liking for that can be grown. Consequently, supporting other platforms such as Android and Windows Phone can be done in the future.

Currently, although the user interface components offer almost the same interfaces for the web and iOS platform, they are distributed in various libraries. Thus, to port an application to a new target, the import statements should be changed. It is possible to make this process automatic in order to simplify the maintenance process. One another solution is to provide a higher level of abstraction on top of different libraries. Designing this level of abstraction to capture the specialties of different platforms in addition to their commonalities can be one of the research topics of the future investigations.

Moreover, the application adaptation based on the target device characteristics is another topic for the future work. For example, if an application is ported to a device with a larger screen size, how it can adapt itself with this new situation? mobl supports manual adaptation using some annotation such as ”@when” which makes the resulting application adaptive based on some environmental conditions such as the screen size. After desugaring, this annotation is desugared to ”when-else” clause. This feature is not completely supported in iOS back-end which can be completed in the future.

On the other hand, iOS supports device re-orientation. In current version of mobl-ios we only support the default re-orientation behavior of the controls but using ”@when”, it is possible to implement customized re-orientation behavior. This can be also considered as a task for the future.


Appendix A

Glossary

In this appendix, we give an overview of frequently used terms and abbreviations.

**AIO:** Abstract Interface Object

**AJAX:** Asynchronous JavaScript and XML

**API:** Application Programming Interface

**AST:** Abstract Syntax Tree

**CIO:** Concrete Interface Object

**Closure:** A block of code that can be represented as a first-class data structure and placed seamlessly into the flow of code by allowing it to reference its lexical scope [23].

**CLR:** .NET Common Language Runtime

**Cocoa Touch:** The top layer of iOS architecture which provides user interface APIs.

**CPS:** Continuation-Passing Style

**CSS:** Cascading Style Sheets

**Derived Variable:** A variable which keeps itself synchronized with the other variables constructing its initial expression.

**DIMAG:** Device Independent Mobile Application Generation (an approach)

**Domain Model:** A specialized model for a specific context.

**DSL:** Domain Specific Language - a computer programming language of limited expressiveness focused on a particular domain [23].

**GCD:** Grand Central Dispatch

**GPS:** Global Positioning System
A. Glossary

HQL: Hibernate Query Language

HTML: HyperText Markup

IB: Interface Builder - Apple tool in order to design the user interface of an application.

IDE: Integrated Development Environment

IDEAL: Interface DEscription Authoring Language

iOS: The operating system of Apple portable devices.

JSP: JavaServer Pages

JVM: Java Virtual Machine

mobl: A DSL of mobile web

Model: An abstraction of a (real or language based) system allowing predictions or inferences to be made [29].

MDSD: Model Driven Software Development - a software engineering approach that aims to improve software development process by employing high-level, domain-specific models in every cycle of a software system development such as implementation, integration, maintenance, and testing [21].

MVC: Model-View-Controller

ORM: Object-Relational Mapper

OS: Operating System

PIL: Platform Intermediate Language

Platform: A set of subsystems and technologies that provide the capabilities needed to support the execution of software [18].

Reactive Programming: A programming paradigm based on which the data flows and change propagation should be expressed easily within a software system.

Re-targetability: An attribute of compilers to generate code for more than one platform.

SCXML: State Chart eXtensible Markup Language

SDK: Software Development Kit

SQL: Structured Query Language

SQLite: An embedded relational database management system

Tablet: In this thesis stands on any kind of personal digital assistants (PDAs) such as smartphones, pocket PCs etc.
UI: User Interface - The presentation layer of a software system.

UIML: User Interface Markup Language

XAML: eXtensible Application Markup Language (an approach)

Xcode: Apple IDE in order to develop MacOS and iOS applications.

XIML: eXtensible Interface Markup Language (an approach)

XML: eXtensible Markup Language
Appendix B

Shopping List Source Code

In this appendix, we give the Shopping List source code used in chapter five.

```cpp
application MyShoppingList
import mobl::ui::generic //import mobl::ipadui //for iOS back-end

entity Category{
    name : String
}

entity ShoppingItem{
    name : String (searchable)
    price : Num
    order : Num
    isOrdered : Bool
    category : Category
}

function loadCategories() : Collection<Category>{
    if (Category.all().count() == 0){
        add(Category(name = "All"));
    }
    return Category.all();
}

function loadShoppingItems(category : Category, searchPhrase : String) : Collection<ShoppingItem>{
    if (category.name == "All"){
        return ShoppingItem.searchPrefix(searchPhrase).filter("isOrdered", ",false);
    }
}
```
else{
    return
    ShoppingItem.searchPrefix(searchPhrase).filter(
        "category", ",=", category).filter(
        "isOrdered", ",=", false);
}
}

control categoryItem(category: Category){
    label(category.name)
}

color shoppingItemsByCategory(category : Category){
    var searchPhrase = ""
    searchBox(searchPhrase)

    var shoppingItems = loadShoppingItems(category, searchPhrase)

group{
    list (i in shoppingItems){
        item(onclick={
            editItem(i);
        }){
            label(i.name)
        }
    }
}

button("Clean", onclick={
    shoppingItems.destroyAll();
})
button("Add Item", onclick={
    addItem(category, shoppingItems);
})
}

color addCategory(){
    var c = Category(name = "")
    header("Add Category"){
        button("Done", onclick={
            add(c);
            screen return;
        })
    }
}
group{
```swift
item{
    textField(c.name, placeholder = "Category Name")
}
}

screen editItem(s : ShoppingItem){
    header("Edit Item"){
        backButton()
    }
    group{
        item{
            textField(s.name, placeholder = "Item Name", label="Item Name")
        }
        item{
            numField(s.price, placeholder = "Price", label="Price")
        }
        item{
            numField(s.order, placeholder = "Order", label="Order")
        }
        item{
            checkBox(s.isOrdered, label="Is Ordered:")
        }
    }
}

screen addItem(category : Category, shoppingItems : Collection<ShoppingItem>){
    var s = ShoppingItem(
        name = ",",
        price = 0,
        order = 0,
        category = category
    )

    header("Add Item"){
        button("Done", onclick={
            //add(s); This fixes web mobi Bug.
            shoppingItems.add(s);
            screen return;
        })
    }
    group{
        item{
```
B. Shopping List Source Code

textField(s.name, placeholder = "Item #1", label = "Item Name")
}
item{
    numField(s.price, placeholder = "120", label = "Price")
}
item{
    numField(s.order, placeholder = "14", label = "Order")
}

screen root() {
    header("Shopping List"){
        button("Add Category", onclick={
            addCategory();
        })
    }
    masterDetail(loadCategories(), categoryItem, shoppingItemsByCategory)
}