

Power of Clouds In Your Pocket: An Efficient Approach for Cloud Mobile Hybrid Application Development

Ashwin Manjunatha, Ajith Ranabahu, Amit Sheth and Krishnaprasad Thirunarayan
Ohio Center of Excellence in Knowledge-Enabled Computing (Kno.e.sis)
Wright State University, Dayton, Ohio 45435
Email: { ashwin, ajith, amit, tkprasad }@knoesis.org

Abstract

The advancements in computing have resulted in a boom of cheap, ubiquitous, connected mobile devices as well as seemingly unlimited, utility style, pay as you go computing resources, commonly referred to as Cloud computing. However, taking full advantage of this mobile and cloud computing landscape, especially for the data intensive domains has been hampered by the many heterogeneities that exist in the mobile space as well as the Cloud space.

Our research focuses on exploiting the capabilities of the mobile and cloud landscape by defining a new class of applications called cloud mobile hybrid (CMH) applications and a Domain Specific Language (DSL) based methodology to develop these applications. We define Cloud-mobile hybrid as a collective application that has a Cloud based back-end and a mobile device front-end.

Using a single DSL script, our toolkit is capable of generating a variety of CMH applications. These applications are composed of multiple combinations of native Cloud and mobile applications. Our approach not only reduces the learning curve but also shields developers from the complexities of the target platforms. We provide a detailed description of our language and present the results obtained using our prototype generator implementation. We also present a list of extensions that will enhance the various aspects of this platform.

1 Introduction

Lately there have been interesting changes at both ends of the *spectrum of computing power*. On one end there has been a boom in mobile computing devices, supported by fast growing communication networks. On the other end, there has been substantial growth in high-end data centers that offer cheap, on-demand, and virtually unlimited computing resources, popularly named *computing Clouds*.

In the backdrop of these advances in computing and the growth of data intensive domains such as social networks,

a new class of applications has emerged taking advantage of not only on-demand scalability of computing clouds but also the sophistication of current mobile computing devices. This class of applications that we name as *cloud-mobile hybrids* (CMH), is characterized by the need for heavy computations on the back-end and mobile device based front-end. The front-end and back-end, that may appear to be two independent applications, are collectively considered to be a single application in terms of the overall functionality.

An illustrative example of a CMH is an implementation of the *Privacy Score* [13] algorithm. Privacy score is a numerical indicator of the level of private details exposed by an individual in a social network. This score is a relative measure and requires substantial computations in the back-end. The incentive to house the front-end of such an application in a mobile device comes from the fact that an increasing number of social network interactions are performed via mobile devices¹.

The present state of the art in mobile front-ends has changed from mobile-enabled Web sites to platform native applications. These native applications offer a better user experience by tightly integrating with the host platform and taking full advantage of the capabilities of the device, but greatly increase the complexity in development. The three major challenges discussed below highlight why developing a CMH application is significantly more difficult and complicated than developing a regular application.

(1) The multitude of existing Clouds offer different paradigms, programming environments and persistence storage. The heterogeneity present in the core Cloud services effectively locks the developers to a particular vendor, making the porting of applications across Clouds problematic.

(2) A number of mobile development platforms exist today, each with different development environments, Application Programming Interfaces (API), and programming languages. Fragmentation of APIs even within a single plat-

¹<http://www.facebook.com/press/info.php?statistics>

form forces mobile application developers to focus on only specific platforms and versions [18, 12]. The current practice in the industry is to concentrate the development efforts on selected mobile platforms, leaving out a significant portion of devices and platforms.

(3) Developing the back-end and front-end as separate components require managing the communication interfaces. The presence of Remote Procedure Calls (RPC) makes the whole development process tedious, even with an arsenal of sophisticated tools at a developer's disposal. The separation of the front-end and the back-end is also a source of version conflicts with Clients and Services where the service API has to be maintained at the level of the least capable client. Introducing changes to the service API could create incompatibility for the existing clients requiring frequent updates and patches. This is a common problem faced by many of the mobile application vendors.

The objective of this research, therefore, is to provide a disciplined approach to overcome the above challenges. Our solution is centered around a DSL based platform agnostic application development paradigm for CMH applications. We demonstrate that treating a CMH application as a single entity that use a single DSL script to describe it, can significantly reduce complexity and also facilitate portability. By taking this approach, the developers are shielded from the heterogeneities of each of the platforms as well as lengthy debug cycles of RPCs. The DSL is also capable of providing abstractions over certain special mobile and Cloud functions such as location and power awareness, enabling developers more flexibility. Some of the limitations in using this generative approach has been discussed in detail in Section 7.

Our current prototype toolkit is capable of generating code for four target platforms. Evaluations performed with this prototype language and associated tools indicate significant reduction of effort in creating Cloud-mobile hybrid applications. These results are discussed in detail in Section 5.

The rest of this paper is organized as follows. We present our motivation in Section 2 and the reasons to use a DSL in Section 3. Then we present our prototype DSL in Section 4. The evaluation and comparisons with existing frameworks are presented in Section 5.

2 Motivation

The primary motivation for our research comes from the promising nature of the Cloud and Mobile combination. Current trends indicate a boom in CMH applications in the future and our contributions stand to provide a well-defined methodology to exploit them. However, there is ample evidence that both the mobile space and the cloud space are facing difficulties due to vendor lock-in.

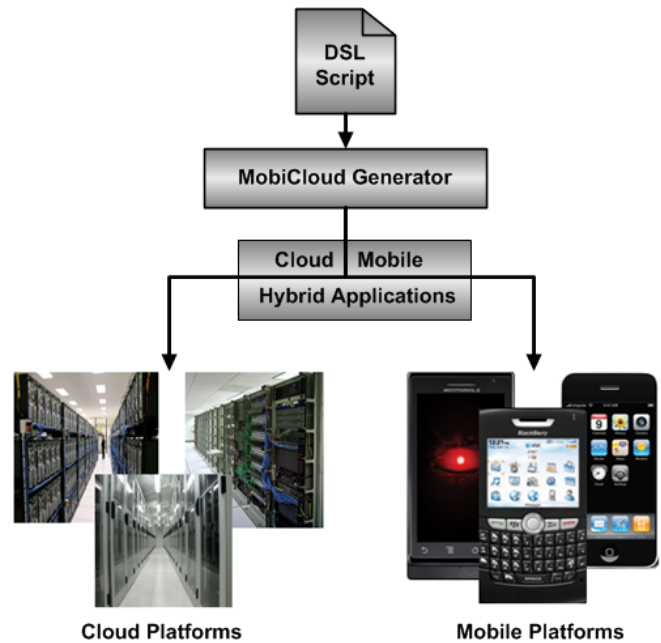


Figure 1: An overview of Cloud mobile hybrid application generation process

The Consumer Electronics Show (CES)² is the premier showcase of the consumer electronics devices and is indicative of trends in the current and future mobile device markets. During the last CES event, developers openly expressed frustration over a lack of consolidation of mobile platforms [11]. Rajapakse [20] discusses in detail the fragmentation in mobile platforms.

Similar fragmentation has occurred in the Cloud space in which vendors tend to develop their own paradigm [6]. Hence, the Cloud remains a largely non-standard space despite the efforts of the National Institute of Standards and Technology (NIST). Many recent industry surveys indicate that the practitioners still consider vendor lock-in a serious hindrance to Cloud computing adoption[21]. Some experts have also suggested that vendors may purposely promote the Cloud to be a heterogeneous patchwork of frameworks for business reasons [5].

Fragmentation on both ends of the spectrum presents a serious challenge in developing Cloud-mobile hybrid applications. Addressing the heterogeneity at both ends increases the effort required in all stages of the software development life cycle, driving up the cost [20]. For example, although the high-level design and the intended functionality are the same, two different engineering efforts are required to address two mobile platforms. Such efforts increase drastically with multiple mobile and Cloud platforms.

The total number of combinations that exist for CMH appli-

²<http://www.cesweb.org/>

cations (T_c) is

$$T_c = \sum_{i=0}^m \{MV_i\} \times \sum_{j=0}^c \{CV_j\} \quad (1)$$

Where m is the number of mobile platforms, c is the number of cloud platforms, MV_i is the number of versions of the i th mobile platform, and CV_j is the number of versions of the j th cloud platform.

However, the number of generators that need to be maintained (T_g) is

$$T_g = \sum_{i=0}^m \{MV_i\} + \sum_{j=0}^c \{CV_j\} \quad (2)$$

Approximating the real world numbers, assuming there are 4 mobile platforms with 2 versions each and 3 cloud platforms with 2 versions each, the total combinations that exist is 48 according to Equation 1. The total number of generators required is 14 according to Equation 2. In practice, the number of required generators is lesser since some platforms are backward compatible. Without a clear development methodology, Cloud-mobile hybrid applications will remain an expensive and exotic option for businesses.

3 A Case for DSLs

A DSL is a programming language or an executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [23]. DSL centric approaches have been used in many domains, particularly due to the expressiveness in the domain of interest, runtime efficiency and reliability due to the narrow focus [22]. For example, mathematicians are quite familiar with specialized languages such as Matlab [10] that provide a convenient way to write matrix oriented programs.

The emergence of interpreted languages such as Ruby have been a key enabler for many modern DSLs. A Ruby based DSL to provide programming abstractions for light weight service compositions (a.k.a. mashups) has been successfully used in the IBM Sharable Code project [15], where one of the authors of this paper is a key developer.

DSLs however, are not the silver bullet that provide a universal solution. A DSL by definition, caters only to a specific domain and not applicable outside the targeted domain. The rule of thumb is more narrowly focused a DSL is, the better suited it can be for the domain of interest compared to competing generic solutions. Hence, the design of a DSL involves careful trade-offs between the breadth of the target domain, features of the target platforms, performance, and many other issues. Given a class of applications, a DSL greatly reduces the effort required to create programs and lowers the barriers to entry.

Greenfield et al.[9] have advocated that DSL centric development processes may be the best for the future. They argue that Object Oriented programming (OOP) based methods regularly fail to adhere to time and budgetary constraints. According to Greenfield, some of the OOP methods do not provide enough levels of abstractions. DSLs excel in providing high levels of abstractions given a constrained domain. Just as domains can be defined with various degrees of granularity, DSLs that cater for these domains are also at different levels of granularity. For example, the base Matlab DSL provides abstractions for general mathematics. Matlab toolkits provide operators for specialized sub-domains of mathematics such as neural networks or statistics. Greenfield also provides an excellent categorization of different types of DSLs that highlight the difference between the levels of granularity. In this case we focused on providing a *sufficiently* high level of abstraction with predefined transformations, a *logical* DSL according to Greenfield's categorization.

Following a similar line of thinking, we advocate a model-driven development process for CMH applications. However, our model is pre-set and expressed in a developer friendly DSL script that can be directly compiled into executable artifacts. Although the generated executable code may not be the optimum in all cases, the human effort required to optimize it can hardly be justified in comparison to the expense on additional computing power. This is highlighted in the so called *Carbon vs Silicon debate* which argues that in many cases it is cheaper to add extra computing power (silicon) rather than optimizing the software with human effort (carbon) [1].

4 A DSL for Hybrid Applications

In this research, we focused on interactive Web applications driven by Create, Retrieve, Update, and Delete (CRUD) operations. These applications typically use multiple data structures in a data centric back-end and use a mobile or Web based front-end to manipulate these data structures. The use of Cloud in these applications is primarily for scalability, i.e., the application itself would not require a massive processing capability but is likely to receive a large number of simultaneous requests and hence, needs to scale accordingly.

An example of such an application is a *to-do list manager* similar to the very popular task manager application offered by *Remember the Milk*³. This application allows users to create *to-do items* using their mobile devices and stores them in a Cloud data store. These reminders can later be retrieved as a list, either on a mobile device or on the Web.

Developing an application of this nature from scratch requires developing the following components:

³<http://www.rememberthemilk.com/>

- (1) A data storage mechanism tied to the storage technology of choice. It is customary to employ an Object-Relational layer to supplement the data access when the considered programming language is object oriented.
- (2) A service layer capable of exposing the operations on the data store. Lately the choice of developers has been RESTful services, but standard Web service technologies may be used to fulfill enterprise customer requirements.
- (3) A service access layer in the targeted front-end capable of accessing the services defined on the server side.
- (4) Relevant user front-end components.

The most appropriate design pattern for this type of application has been identified as the Model-View-Controller (MVC) pattern. Figure 2 illustrates the major components present in a MVC based design.

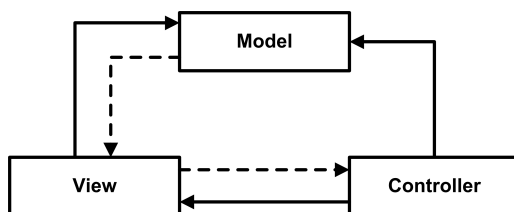


Figure 2: Model-View-Controller design pattern

Model represents a data-structure that holds neutral representation of the data items pertinent to the application. A *view*, representing the data in a format suitable to the user, observes the model and updates its presentation. Any interactions with the view are processed via a *controller* which adjusts the model that are reflected in the view. The controller typically restricts the operations on the model and may directly update the view to notify the status of an operation. This pattern has been the basis for many of the current Web application frameworks such as the Oracle Application Development Framework, Apache Struts⁴, and Ruby on Rails⁵.

The DSL we have experimented with is designed according to the MVC principles and directly reflects the definitions of the relevant components. Listing 1 illustrates the major components of the language in BNF notation. Note that some definitions such as ARGLIST and IDARG are omitted for brevity. The complete BNF specification of the grammar is available from the on line resources⁶. This language has been developed by restricting the Ruby base language. Extending or restricting a base language is a known DSL design technique [22] and provides many conveniences later in the development life-cycle, primarily due to the presence of language machinery for parsing.

⁴<http://struts.apache.org/>

⁵<http://rubyonrails.org/>

⁶<http://knoesis.wright.edu/mobicloud>

Ruby has been especially noted for its suitability as a base language for DSLs [4]. The IBM Sharable Code DSL was designed by restricting the Ruby base language and has been quite successful in providing a significant level of abstraction in defining a light weight service composition.

Listing 1: Partial BNF grammar for the DSL

```

RECIPE           : 'recipe' IDARG 'do'
                  METADATA
                  MODEL*
                  CONTROLLER*
                  VIEWS* 'end'
METADATA         : 'metadata' HASH
CONTROLLER       : 'controller'
                  IDARG 'do' ACTION*
                  'end'
ACTION           : 'action' SYMBOL_LIST
VIEW             : 'view' ARGLIST
MODEL            : 'model' ARGLIST

```

We now present a *hello world* application written using this DSL to exemplify the features of the language. Listing 2 depicts the DSL script for this application. The intention of this application is to *illustrate* the components.

- (1) A minimal *model* with only one attribute.
- (2) A minimal *controller* with only one action.
- (3) A minimal *view* demonstrating a minimal user interface.

This application displays a greeting message on the mobile device by fetching it from remote, cloud based data storage via a RESTful service interface.

Listing 2: The DSL script for the *hello world* application

```

recipe : helloworld do
  metadata : id => 'helloworld-app'

  model : greeting ,
         { :message => :string }

  controller : sayhello do
    action : retrieve , :greeting
  end

  view : show_greeting ,
        { :models => [:greeting] ,
          :controller => :sayhello ,
          :action => :retrieve }
end

```

We now describe each of the major constructs of the language in detail.

Metadata

A collection of key-value pairs indicating metadata associated with this application. There are no enforced metadata values, but depending on the choice of the targets, certain metadata values may be deemed essential. For example, when targeting the Google Appengine⁷, the `:id` value assumes the Google Application Id value and is deemed mandatory.

Models

The models section defines each model with a name and a list of key-value pair attributes. The key-value pairs indicate the attribute name and the data type of the attribute. In this example `greeting` is the name of the model and it has one string attribute called `message`. A single DSL can include any number of models. The name of the model acts as a unique identifier for a model and is used to refer to models in others sections of the DSL script. Models may translate to data objects on both the client and the server to represent the same data structure.

Controllers

Controllers define actions on models. The standard actions include Create, Retrieve, Update, and Delete and their operations are implied. For example, `:create` implies creating a relevant model object, assuming the required and optional parameters are provided. `:retrieve` implies retrieving the attribute values of a selected model object.

Views

Views define GUI components, translated to the necessary code, that generate a suitable rendering on the targeted platform. The visual components of the views are implied from the action and the model the view is associated with. For example, a `:retrieve` operation implies that attributes of a model object needs to be displayed. Hence, the view contains labels (or other suitable components) to display the attribute values.

Recipe

Recipe encapsulates all other components and acts as the housing for the components mentioned before. Figure 3 illustrates the mapping of the generated artifacts to the original MVC pattern.

5 System Implementation and Evaluation

The system was implemented in Ruby in order to take advantage of the existing Ruby parser and interpreter. Figure 4 illustrates major components of the system. The parser is a top down parser that takes the DSL scripts (a.k.a.

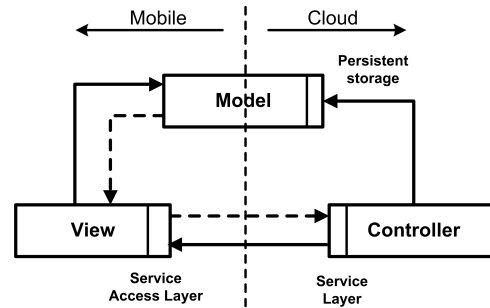


Figure 3: Mapping of Artifacts to MVC components

recipes) and converts them into in-memory object representations. These object models are then converted into platform specific code using the corresponding generator and the associated templates. To support a new platform, the system requires only an additional generator targeted towards the new platform. We present an evaluation based on code metrics of the generated artifacts for two programs in Table 1.

These metrics were obtained using the Eclipse Metrics plugin⁸ and excludes non-java code (such as Android view XML files and build files). For both cases of Android and Google Appengine combination, developers have to write approximately 3% of the code they would have written otherwise. This is even lesser for the Blackberry and Google Appengine combination (2.5%). The number of classes and methods also indicate the complexity of the generated code. These metrics do not reflect the relieving of the debugging effort for RPCs. Auto generating the remote communication components removes many sources of errors and inconsistencies.

The generator tool, complete set of programs and XML version of all results, is available on the Kno.e.sis Website⁹.

6 Related Work

Many frameworks that support remote communications (RPC) contain tools to generate concrete code by compiling an interface definition. For example, Common Object Request Broker Architecture (CORBA) uses a special language called Interface Definition Language (IDL) to define interfaces. The IDL scripts are then used with an IDL compiler to generate executable code for the targeted platform. A similar role is played by the Web Services Description Language (WSDL) for Web services. However, all of these languages focus only on providing a portable interface. Generating a complete program is harder than catering only for the interface.

The closest framework in concept to this research is Google Web Toolkit (GWT) [3]. GWT is an AJAX[8] de-

⁷<http://appengine.google.com>

⁸<http://metrics.sourceforge.net/>

⁹<http://knoesis.org/mobicloud>

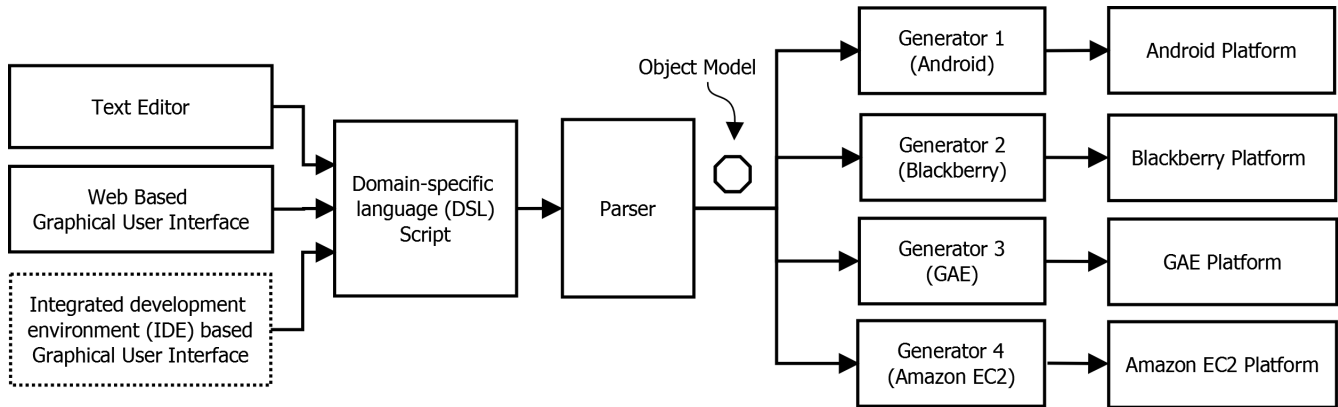


Figure 4: System Implementation Components and Flow

Application	DSL LoC <i>a</i>	Target platform	LoC <i>b</i>	N_C <i>c</i>	N_M <i>d</i>
HelloWorld	8	Android	170	9	4
		Blackberry	168	6	8
		Amazon EC2	110	4	10
		Google Appengine	80	4	8
Todolist	12	Android	225	10	6
		Blackberry	324	8	19
		Amazon EC2	215	5	27
		Google Appengine	158	5	22

^aLines of Code in DSL

^bLines of Code Generated

^cNumber of Classes

^dNumber of Methods

Table 1: Comparison of Code Metrics for the Generated Applications

velopment tool from Google, targeted for Java Developers. Web applications (both GUIs and RPCs) are written in Java using the GWT API. The Java files are then compiled into compact, optionally obfuscated, JavaScript files. GWT offers a scalable solution that manages complexity of cross browser compatibility issues by generating functionally equivalent but browser specific Javascript and corresponding back-end code for the server side. GWT has been successfully used to build many high profile Web Applications.

ISC is another example of a similar tool but uses a custom DSL rather than a generic programming language. ISC reduces the amount of code significantly although the scope of it is only mashups. Features of GWT, ISC, and MobiCloud are compared in Table 1.

7 Discussion

7.1 Deployment complexity

Although the generated applications can be tested on the provided mobile device emulators, deployment to the actual device may require a signing step (using an authenticated key) and optionally an upload to a vendor controlled *app store*. Some of these workflows have been deliberately kept as human centric operations by the vendors. Even if there are Web APIs present, managing keys, certificates and other deployment operations require the presence of a different layer of automation. Although such facilities are out of scope of this work, adding a middleware layer capable of managing deployments and subsequent management tasks, such as Altocumulus [14] (where the second author is a major contributor), would improve the reach and the usability of the DSL.

7.2 Application UI Features

Another potential limitation is the generic nature of the applications that are being generated. For example, the generated UI's use minimal decorations and are focused on functionality, rather than visual appeal. Even if the generic UI features can be improved, developers may want to customize their application's visual components. There are two possible solutions:

(1) Use a secondary DSL to define custom UI components and attach them to the views. This is discussed in detail in Section 8.2.1.

(2) Use the generated projects to bootstrap custom development. This is similar to the model driven development process followed by many major software companies where a high level model, such as UML diagram, is used to bootstrap the development process. Special attention has been given in generating mobile artifacts to support this style of development.

Feature	MobiCloud	ISC	GWT
Language Support	Custom DSL (Ruby based)	Custom DSL (Ruby based)	Java
Generation capability	Multiple Cloud and mobile applications	Ruby on Rails Web application	Java Servlet based Web application
Available Tools	Web based text editor and compiler	Advanced Web based text editor and compiler	Rich IDE based editors, IDE and command line based compilers
Supported Clouds	Google Appengine and Amazon EC2	Amazon EC2 (via Heroku)	Google Appengine
Mobile platforms	Android 1.5, Blackberry	None	None

Table 2: Feature comparison of MobiCloud, ISC and GWT

7.3 Custom Actions

Currently the capabilities of the language are limited in terms of actions. Although the standard CRUD operators are sufficient for simple applications, custom actions become an absolute necessity when the applications grow in complexity. Similar to the customization of the UI, we outline an enhancement to the language that enables plugin actions using user defined functions. These actions may also be written in other DSLs such as PIGLatin [19] scripts. A possible way to incorporate custom actions is outlined in Section 8.2.2.

8 Future Work

We discuss four key extensions and enhancements we are currently working on, some of them to alleviate the shortcomings discussed in Section 7.

8.1 Generic Data Definitions

One avenue of future work we plan to pursue is enabling generic data definitions. Rather than defining the data inline, well known data items may be referenced. These data items may be defined in the now widely adopted RDF (Resource Description Framework, W3C's Sematic Web data representation language) and referred via URL references. For example, a data type referring to a *Person* can use the Friend Of A Friend (FOAF) Person definition as exemplified in Listing 3. The advantage of using a generic model such as a RDF definition is the ability to convert it to different storage representations. This becomes extremely valuable when data needs to be moved from one representation to another using the famous *lifting-lowering* mechanism.

Listing 3: Using a Reference to Define Data Types

```
model : person , { : ref => "foaf:Person" }
```

8.2 Language Extensions

8.2.1 UI customization

The mobile UIs may be customized by adding UI specific *templates*. These templates may be written in a platform agnostic UI oriented DSL such as XAML [17]. The generators, however, need to be aware of specific UI compilations

of this DSL for the target platform. A sample XAML template for the Hello World application is illustrated in Listing 4. Some details such as namespaces are omitted in this listing for brevity. Listing 5 illustrates the reference being added to the Hello World application. Note the use of embedded code fragments to retrieve data from model objects.

Listing 4: An Example XAML template for the Greetings UI

```
<Canvas>
  <Rectangle Fill="PowderBlue" />
  <TextBlock
    Foreground="Teal"
    FontFamily="Verdana"
    FontSize="18"
    FontWeight="Bold"
    Text="<%@model.message%" />
  />
</Canvas>
```

Listing 5: Using a Reference to XAML based UI template

```
view : show_greeting ,
{ : models => [: greeting ] ,
  : controller => : sayhello ,
  : action => : retrieve ,
  : uieref => "hello.xml" }
```

8.2.2 Action customization

Similar to the UI customizations, the language can be extended to include custom actions. The operations may be specified by other DSLs and either embedded in the code or referred to external files in a similar fashion to UI customization. These custom actions may take advantage of certain cloud features such as the capability to do map-reduce style processing.

Listing 6 illustrates a possible way to add a custom action written in PIGLatin script that sorts a (fictitious) set of items having multiple attributes. In order to use this type of custom actions, the necessary persistence storage (such as HDFS [2]) should be available in the targeted Cloud platform.

Listing 6: Embedding a PIGLatin script in a custom action

```

action : sort_items ,
:item , { : lang => 'PIG' } do
%{
  A=load 'items' using PigStorage ()
  as ( a , b , c );
  B=sort A by a;
}
end

```

8.2.3 Graphical Abstractions

The simplicity of the DSL enables it to be generated from a graphical representation similar to Yahoo! pipes [7]. Such graphical abstractions are capable of enabling non-programmer use this DSL to generate custom applications. Due to faster development cycles, it is possible to have customized applications for personal use that can later be discarded. These graphical abstractions may be used to create mobile mashups as envisioned by Maximilien[16].

9 Conclusion

Our research clearly indicates that using a DSL significantly reduces the development effort for Cloud-Mobile hybrid applications. A DSL shields developers from minor details of the target platform and reduces the number of defects by auto-generating communication interfaces. Although the DSL presented in this research has room for many improvements, it has clearly demonstrated the applicability of DSLs in both Cloud and Mobile spaces.

References

- [1] J. Atwood. Hardware is Cheap, Programmers are Expensive, 2008. Available online at <http://bit.ly/avyNiN> - Last accessed Spt 3rd 2010.
- [2] D. Borthakur. The Hadoop Distributed File System: Architecture and Design.
- [3] E. Burnette. Google Web Toolkit—Taking the pain out of Ajax. *USA: The Pragmatic Programmers LLC*, 2006.
- [4] H. C. Cunningham. A little language for surveys: constructing an internal DSL in Ruby. In *ACM-SE 46: Proceedings of the 46th Annual Southeast Regional Conference on XX*, pages 282–287, New York, NY, USA, 2008. ACM.
- [5] D. Durkee. Why cloud computing will never be free. *Communications of the ACM*, 53(5):62–69, 2010.
- [6] Economist Opinion Section. Clash of the Clouds. *The Economist*, 2009. published online at <http://bit.ly/cBRAfB> : Last accessed August 27th 2010.
- [7] J. Fagan. Mashing up Multiple Web Feeds Using Yahoo! Pipes. *Computers in Libraries*, 27(10):8, 2007.
- [8] J. Garrett et al. Ajax: A new approach to web applications. 2005.
- [9] J. Greenfield and K. Short. Software factories: assembling applications with patterns, models, frameworks and tools. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 16–27, New York, NY, USA, 2003. ACM.

- [10] D. Hanselman and B. Littlefield. *Mastering MATLAB 5: A comprehensive tutorial and reference*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1997.
- [11] A. Johnson. Apps call, but will your phone answer? published online at <http://bit.ly/7OfKeO> : Last accessed August 27th 2010.
- [12] O. Kharif. Android's spread could become a problem. <http://bit.ly/d0IHG8>.
- [13] K. Liu and E. Terzi. A Framework for Computing the Privacy Scores of Users in Online Social Networks. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, pages 288–297. IEEE Computer Society, 2009.
- [14] E. Maximilien, A. Ranabahu, R. Engehausen, and L. Anderson. Toward cloud-agnostic middlewares. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 619–626. ACM, 2009.
- [15] E. Maximilien, A. Ranabahu, and K. Gomadam. An Online Platform for Web APIs and Service Mashups. *IEEE Internet Computing*, 12(5):32–43, 2008.
- [16] E. M. Maximilien. Mobile mashups: Thoughts, directions, and challenges. In *ICSC '08: Proceedings of the 2008 IEEE International Conference on Semantic Computing*, pages 597–600, Washington, DC, USA, 2008. IEEE Computer Society.
- [17] Microsoft. Extensible Application Markup Language. *Microsoft Developer Network (MSDN)*, 2008.
- [18] R. Moll. Knowing is half the battle. <http://bit.ly/cvvWaR>.
- [19] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.
- [20] D. C. Rajapakse. Techniques for de-fragmenting mobile applications: A taxonomy. In *SEKE*, pages 923–928. Knowledge Systems Institute Graduate School, 2008.
- [21] Rightscale.com. The Skinny on Cloud Lock-in. <http://blog.rightscale.com/2009/02/19/the-skinny-on-cloud-lock-in/>.
- [22] D. Spinellis. Notable design patterns for domain-specific languages. *The Journal of Systems & Software*, 56(1):91–99, 2001.
- [23] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.