

A Domain Specific Language Based Method to Develop Cloud-Mobile Hybrid Applications

Ashwin Manjunatha, Ajith Ranabahu, Amit Sheth and Krishnaprasad Thirunarayan

Kno.e.sis Center

Wright State University, Dayton, Ohio 45435

Email: { ashwin, ajith, amit, tkprasad }@knoesis.org

Abstract—The advancements in computing have resulted in a boom in cheap, ubiquitous, connected mobile devices as well as seemingly unlimited, utility style, pay as you go computing resources, commonly referred to as Cloud computing. Taking advantage of this powerful computing landscape however has been hampered by the many heterogeneities that exist in the mobile space as well as the Cloud space.

This research attempts to alleviate many of the issues faced in developing these Cloud-mobile hybrid applications by introducing a Domain Specific Language (DSL) centric approach to generate applications. The generators are capable of creating native applications for a variety of Cloud and mobile platforms using a single DSL script. This not only reduces the learning curve but also shields the developers from the native complexities of the target platforms. We provide a detailed description of our language and present the results obtained using our prototype generator. We also present a list of extensions we are working on related to this research.

I. 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, fueled 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 *Cloud computing*.

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, named *cloud-mobile hybrids*, are characterized by the need for heavy computations on the back-end but only simple user interfaces on the front-end. An example of this class of applications is an implementation of the *Privacy Score* [1] 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 that can be done in parallel. Presenting the score to the user however requires only a simple UI that may be implemented on a mobile device. With the popularity of social networks and microblogging services, a vast amount of data is being collected everyday that provides a plethora of opportunities to implement similar types of data and compute intensive applications.

Harnessing the power of this new computing landscape however, is still a challenge. This challenge needs to be overcome on three fronts:

- (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.
- (2) The heterogeneity in the mobile application platforms stem from different development environments, Application Programming Interfaces (API) and programming languages. Fragmentation of APIs even within a single platform forces mobile application developers to focus on only specific platforms and versions [2], [3]. The current practice in the industry is to concentrate the development efforts on selected mobile platforms, leaving out a significant portion of various devices and platforms.
- (3) There is difficulty in managing the communication interfaces. The front-end and back-end separation that requires 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 would break the existing clients requiring frequent updates, a common problem faced by many of the mobile application vendors.

The objective of this research is to provide one approach to address the first two of the above mentioned challenges and partially address the third challenge. Our solution is centered around a Domain Specific Language (DSL) based platform agnostic application development paradigm for cloud-mobile hybrid applications. We demonstrate how applications described using a single DSL script can significantly reduce complexity. At the time of deployment, a generator compiles the DSL script into the native format of the targeted platform. The client server interfaces are automatically generated, relieving the developers from the minute details of the communication interfaces. By taking this approach, the developers are shielded from the heterogeneities of each of the platforms as well as lengthy debug cycles of client-server communication. Our prototype compiler/generator is capable of generating code for four target platforms and our evaluations indicate significant

reduction of effort in creating a simple but functional Cloud-mobile hybrid application. These results are discussed in detail in Section V.

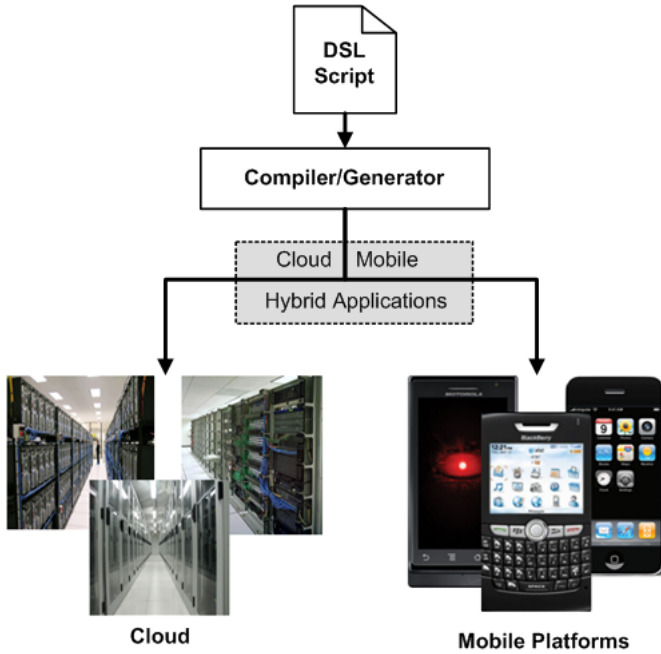


Fig. 1. An overview of Cloud mobile hybrid application generation process

The rest of this paper is organized as follows. We present our motivation and background in Section II. The DSL for hybrid applications is discussed in Section III. Translations are described using a simple “hello world” application in Section IV. Section V briefly describes system implementation and presents a code metric based evaluation. A detailed discussion of the limitations of this approach is presented in Section VI, and our on going research is outlined in Section VIII. Section IX concludes the paper.

II. MOTIVATION AND BACKGROUND

The Consumer Electronics Show (CES)¹ 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 [4]. Rajapakse [5] discusses in detail the fragmentation in mobile platforms and provides a number of defragmentation approaches. Similar fragmentation has happened in the Cloud space with each vendor developing their own paradigm [6]. The National Institute of Standards and Technology (NIST) has been instrumental in Cloud standardization efforts and has published the most accepted definition for *Cloud computing* [7]. However, the Cloud still remains a largely non-standard space with many vendor-specific services present. For example, many recent industry surveys indicate that the practitioners still consider vendor lock-in a serious hindrance to Cloud computing adoption[8].

¹<http://www.cesweb.org/>

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 [5]. Without a clear development methodology, Cloud-mobile hybrid applications will remain such an expensive endeavor for businesses that reaching a wider customer base via mobile devices remains unattainable and thereby lucrative business opportunities are missed.

A DSL is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain [9]. 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 [10]. For example, mathematicians are quite familiar with specialized languages such as Matlab [11] that provide a convenient way to write matrix oriented programs. Similar DSL based abstractions have been created for many domains. The emergence of many 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 [12], for which the second author of this paper is a key developer.

DSLs are not a silver bullet that provide a generic solution. DSLs by definition, cater to only a specific domain and become inapplicable outside the targeted domain. For example, the IBM Sharable Code DSL is only useful to prepare service compositions. Similarly, a single DSL is not capable of catering to every type of Cloud-mobile hybrid application. However, given a class of applications, a DSL greatly reduces the effort required to create programs and lowers the barriers to entry.

III. 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*². 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:

²<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.

Long running software engineering research on design patterns has identified the most appropriate design pattern for this type of applications is the Model-View-Controller (MVC) pattern. Figure 2 illustrates the major components present in a MVC based design.

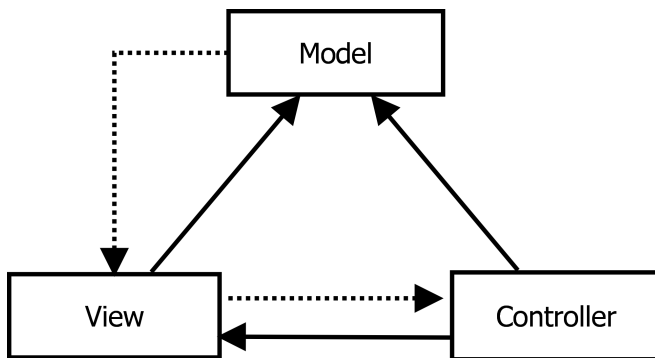


Fig. 2. Model-View-Controller design pattern

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 Struts³ and Ruby on Rails⁴.

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. The complete BNF specification of the grammar is available at [13] under the *Resources* section. This language has been developed by restricting the Ruby base language, a well known DSL design technique [10]. Extending or restricting a base language provides many conveniences later in the development life-cycle, primarily due to the presence of language machinery for parsing. Ruby has been especially noted for its suitability

as a base language for DSLs [14]. 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' ARG_LIST
MODEL       : 'model' ARG_LIST
  
```

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'
  # models
  model :greeting ,
        { :message => :string }

  # controllers
  controller :sayhello do
    action :retrieve , :greeting
  end

  # views
  view :show_greeting ,
      { :models => [:customer] ,
        :controller => :sayhello ,
        :action => :retrieve }
end
  
```

Any Ruby aficionado would easily identify the close resemblance of this DSL to the Ruby language. However although the syntax has been based on Ruby, only a restricted set of constructs are valid. We now describe each of the major constructs of the language in detail.

³<http://struts.apache.org/>

⁴<http://rubyonrails.org/>

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 Appengine⁵, the **:id** value assumes the Googles 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. 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.

Views: Views are 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 **:create** operation implies that a model object needs to be created, hence the view contains inputs for the attributes of the relevant model.

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. These translations are described, in detail, in Section IV.

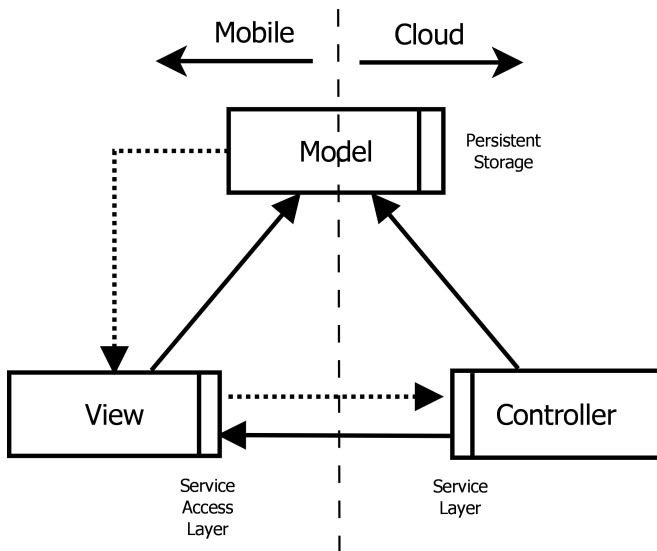


Fig. 3. Mapping of Artifacts to MVC components

IV. TRANSLATIONS

We now describe the translations in detail. The running example used in this section is the *hello world* application in Listing 2. Although we use this hello world application to describe the salient features of this language, it is not an exhaustive demonstration of all the possible options of this DSL. Some extended sample applications that demonstrate advanced capabilities of the DSL are available at [13].

The current system implementation (described in detail in Section V) consists of four target platform capabilities.

Back-end

- 1) A Google Appengine (GAE) compatible Java Web application with data storage capabilities.
- 2) A *pure* Java Servlet / JSP based Web application with data storage capabilities compatible with Amazon EC2. Being *pure* signifies that only standard Java libraries are used in the generated code.

Front-end

- 3) A native Android application containing auto generated user interfaces.
- 4) A native Blackberry application containing auto generated user interfaces.

For the back-end, i.e., the Cloud portions of the application, the following artifacts are generated.

- 1) A set of *bean* classes for the models that optionally include the Java Data Object (JDO) standardized annotations. Each of the beans represent an implementation of the relevant model description.
- 2) A set of JSP based HTML views that mimic the mobile UI. The intention of this UI is to act as a testing tool.
- 3) The implementation of a relevant persistence manager class.
- 4) RESTful Web Service that manages inputs via HTTP POST requests and responds with XML.
- 5) Platform specific configuration files (e.g. the `appengine-web.xml` file required by GAE).

The front-end, i.e., the mobile portion of the application consists of the following artifacts.

- 1) An optional set of *bean* classes for the models depending on the platform requirements. For some platforms the beans are omitted.
- 2) A set of UI classes, event handlers and related artifacts depending on the platform for each of the views.
- 3) An *Index* view that acts as the default entry point to the application.
- 4) Web service access classes that use platform specific communication libraries.
- 5) XML parsing methods that use platform specific XML processing libraries and depend on the specific XML format followed by the back-end serializer.
- 6) Platform specific configuration and descriptor files (e.g. the `AndroidManifest.xml` file required by the Android platform).

⁵<http://appengine.google.com>

The generators also output an Ant⁶ build file capable of generating deployable artifacts. To execute the Ant build, the relevant Standard Development Kit (SDK) needs to be properly configured.

In this specific example, the hello world GAE application consists of the following important artifacts.

(1) Greeting.java: The bean corresponding to the model definition of Greeting. Greeting consists of a single string attribute that represents the message.

(2) Sayhello.java: An HTTP Servlet [15] corresponding to the controller definition. This servlet handles the *:retrieve* action by generating an XML output of the list of Greeting items in the data store. The XML output is generated by a JSP.

Similarly, in the mobile side, the hello world Android application consists of the following major artifacts.

(1) Index.java: An Android activity that acts as the entry point to the application. Provides access to all the other functions, in this case, to view the greeting message.

(2) Corresponding XML files that form the UI.

(3) AndroidManifest.xml: Android specific configuration file, considered mandatory to deploy an Android application.

V. 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. *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 I.

These metrics were obtained using the Eclipse Metrics plugin⁷ and excludes non-java code (such as Android view XML files and build files). The metrics clearly indicate a significant reduction in code that needs to be hand-crafted which removes many sources of errors and inconsistencies.

The generator tool, complete set of programs and XML version of all results, is available on the Knoesis Website⁸ [13].

VI. DISCUSSION

Limitations

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

⁶<http://ant.apache.org>

⁷<http://metrics.sourceforge.net/>

⁸<http://knoesis.org/mobicloud>

Application	Target platform	LOC	Classes	Methods
HelloWorld	Android	170	9	4
	Blackberry	168	6	8
	Amazon EC2	110	4	10
	Google Appengine	80	4	8
Todolist	Android	225	10	6
	Blackberry	324	8	19
	Amazon EC2	215	5	27
	Google Appengine	158	5	22

TABLE I
COMPARISON OF CODE METRICS FOR THE GENERATED APPLICATIONS

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 [16], would improve the reach and the usability of the DSL.

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 VIII-B1.

(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.

3) *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-in actions using user defined functions. These actions may also be written in other DSLs such as PIGLatin [17] scripts. A possible way to incorporate custom actions is outlined in VIII-B2.

VII. RELATED WORK

The closest framework in concept to this research is Google Web Toolkit (GWT) [18]. GWT is an AJAX[19] development 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.

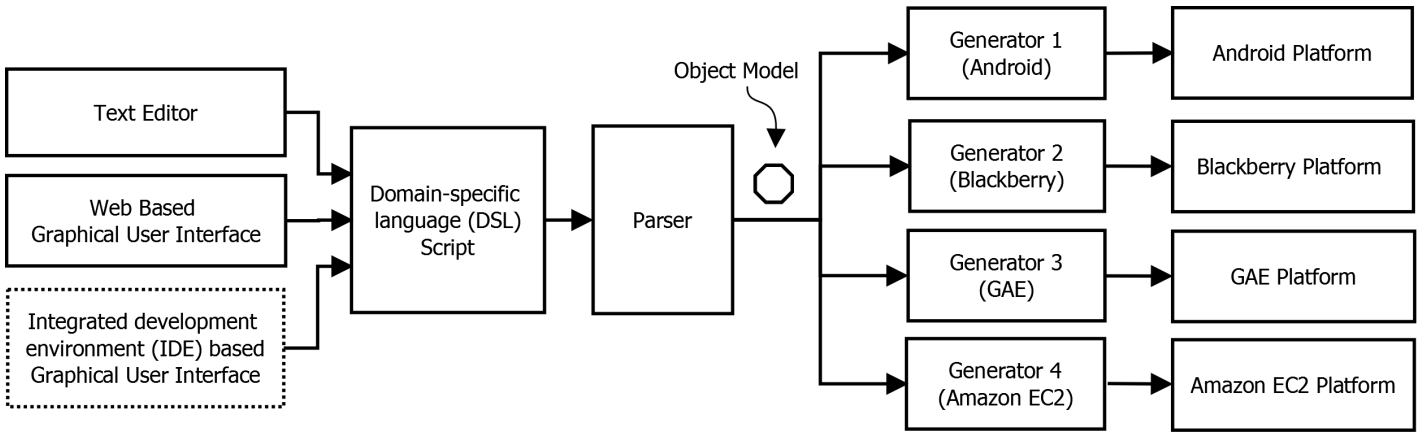


Fig. 4. System Implementation Components and Flow

GWT has been successfully used to build many high profile Web Applications and has been accepted by the industry as a viable solution for generating cross browser capable Asynchronous Javascript and XML (AJAX) applications.

Many frameworks that support remote communications (or 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. Similarly, all leading Web Service frameworks include tools capable of generating code (referred to as *stubs* and *skeletons*) by using WSDL files. The WSDL acts as a high-level interface definition of the service.

VIII. FUTURE WORK

We discuss three key extensions and enhancements we are currently working on.

A. 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 Semantic 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" }
```

B. Language Extensions

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 [20]. 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 =>[:customer] ,
  :controller => :sayhello ,
  :action => :retrieve ,
  :uiref => "hello.xaml" }
```

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 persistence storage should be a distributed file system such as HDFS [21].

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

```

3) *Graphical Abstractions*: The simplicity of the DSL enables it to be generated from a graphical representation similar to Yahoo! pipes [22]. Such graphical abstractions are capable of enabling non-programmer use of 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 in [23].

IX. 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] 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*. IEEE Computer Society, 2009, pp. 288–297.
- [2] R. Moll, "Knowing is half the battle," <http://bit.ly/cvvWaR>.
- [3] O. Kharif, "Android's spread could become a problem," <http://bit.ly/d0IHG8>.
- [4] A. Johnson, "Apps call, but will your phone answer?" <http://bit.ly/7OfKeO>.
- [5] D. C. Rajapakse, "Techniques for de-fragmenting mobile applications: A taxonomy," in *SEKE*. Knowledge Systems Institute Graduate School, 2008, pp. 923–928.
- [6] "Clash of the Clouds," *The Economist*, 2009.
- [7] National Institute of Science and Technology (NIST), "NIST Definition of Cloud Computing - Version 15," <http://csrc.nist.gov/groups/SNS/cloud-computing/cloud-def-v15.doc>.
- [8] Rightscale.com, "The Skinny on Cloud Lock-in," <http://blog.rightscale.com/2009/02/19/the-skinny-on-cloud-lock-in/>.
- [9] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography," *SIGPLAN Not.*, vol. 35, no. 6, pp. 26–36, 2000.
- [10] D. Spinellis, "Notable design patterns for domain-specific languages," *The Journal of Systems & Software*, vol. 56, no. 1, pp. 91–99, 2001.
- [11] D. Hanselman and B. Littlefield, *Mastering MATLAB 5: A comprehensive tutorial and reference*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1997.
- [12] E. Maximilien, A. Ranabahu, and K. Gomadam, "An Online Platform for Web APIs and Service Mashups," *IEEE Internet Computing*, vol. 12, no. 5, pp. 32–43, 2008.
- [13] A. Manjunatha, A. Ranabahu, A. Sheth, and K. Thirunarayan, "Mobi-Cloud," <http://knoesis.wright.edu/mobicloud>.
- [14] 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*. New York, NY, USA: ACM, 2008, pp. 282–287.
- [15] D. Coward, "Java servlet specification version 2.3," *Sun Microsystems*, 2001.
- [16] 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*. ACM, 2009, pp. 619–626.
- [17] 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*. ACM, 2008, pp. 1099–1110.
- [18] E. Burnette, "Google Web Toolkit—Taking the pain out of Ajax," *USA: The Pragmatic Programmers LLC*, 2006.
- [19] J. Garrett *et al.*, "Ajax: A new approach to web applications," 2005.
- [20] Microsoft, "Extensible Application Markup Language," *Microsoft Developer Network (MSDN)*, 2008.
- [21] D. Borthakur, "The Hadoop Distributed File System: Architecture and Design."
- [22] J. Fagan, "Mashing up Multiple Web Feeds Using Yahoo! Pipes." *Computers in Libraries*, vol. 27, no. 10, p. 8, 2007.
- [23] E. M. Maximilien, "Mobile mashups: Thoughts, directions, and challenges," in *ICSC '08: Proceedings of the 2008 IEEE International Conference on Semantic Computing*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 597–600.