2011

A Domain Specific Language Based Approach for Developing Complex Cloud Computing Applications

Ashwin Kumar Manjunatha
Wright State University

A Domain Specific Language Based Approach for Developing Complex Cloud Computing Applications

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering

By

ASHWIN KUMAR MANJUNATHA
B.E., Visvesvaraya Technological University

2011
Wright State University

WRIGHT STATE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

March 14, 2011

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Ashwin Kumar Manjunatha ENTITLED A Domain Specific Language Based Approach for Developing Complex Cloud Computing Applications BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Computer Engineering.

Amit P. Sheth, Ph.D.
Thesis Director

Mateen Rizki, Ph.D.
Chair, Computer Science and Engineering

Committee on
Final Examination

Amit P. Sheth, Ph.D.

Krishnaprasad Thirunarayan, Ph.D.

Paul E. Anderson, Ph.D.

Ramakanth Kavuluru, Ph.D.

Andrew Hsu, Ph.D.
Dean, School of Graduate Studies

ABSTRACT

Manjunatha, Ashwin. M.S.C.E, Department of Computer Science & Engineering, Wright State University, 2011. *A Domain Specific Language Based Approach for Developing Complex Cloud Computing Applications.*

Computing has changed. Lately, a slew of cheap, ubiquitous, connected mobile devices as well as seemingly unlimited, utility style, pay as you go computing resources has become available at the disposal of the common man. The latter commonly called Cloud Computing (or just *Cloud*) is democratizing computing by making large computing power accessible to people and corporations around the world easily and economically.

However, taking full advantage of this computing landscape, especially for the data intensive domains, has been hampered by many factors, the primary one being the complexity in developing applications for the variety of available platforms.

This thesis attempts to alleviate many of the issues faced in developing complex Cloud centric applications by using a Domain Specific Language (DSL) based methods. The research is focused in two main areas. One area is hybrid applications with mobile device based front-ends and Cloud based back-ends. The other is data and compute intensive biological experiments, exemplified by applying a DSL to metabolomics data analysis. This research investigates the viability of using a DSL in each domain and provides evidence of successful application.

Contents

List of Figures

List of Tables

Acknowledgement

There are a number of people without whom this thesis might not have been written, and to whom I am greatly indebted.

Foremost, I would like to thank my advisor, Dr. Amit Sheth for his guidance and the opportunity to work and learn from outstanding students at Ohio Center of Excellence in Knowledge-Enabled Computing (Kno.e.sis).

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Krishnaprasad Thirunarayan, Dr. Paul Anderson, and Dr. Ramakanth Kavuluru for their valuable advice and comments.

I owe my most sincere gratitude to Ajith Ranabahu, who gave me untiring guidance during my research and for those sleepless nights we worked together before deadlines at Kno.e.sis. My sincere thanks also goes to Sanjay Sharma and his research team at LexisNexis for the opportunity to work on exciting projects at LexisNexis during my internship.

Finally, I would like to thank my friends and fellow lab-mates at Kno.e.sis: Ashutosh Jadav, Delroy Cameron, Harshal Patni, Hemant Purohit, Kalpa Gunaratna, Karthik Gomadam, Meena Nagarajan, Pablo Mendes, Pavan Kapanipati, Pramod Anantraman, Raghava Mutharaju, Sarasi Sarangi, Sujan Perera, Vinh Nguyen, and Wenbo Wang for the stimulating discussions and for the good times we have had working together in the last two years.

Dedicated to my
father Manjunatha who offered me unconditional support,
mother Meena for instilling the importance of hard work,
and friend Ajith Ranabahu.

Introduction

> *The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do.*
>
> -Ted Nelson (American sociologist, philosopher, and pioneer of information technology)

Lately there has been huge interests and development at both ends of the *spectrum of computing power* shown in Figure 1.1. On one end there has been a boom in mobile computing devices, supported by sophisticated hardware and operating systems. 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*.

Unfortunately, the boom in high availability of cheap computing devices does not necessarily translate into more problems being solved. As Ted Nelson indicates, computers need to be programmed to be usable. On one hand, the influx of multiple computing devices has aggravated the programming problem since these devices run different operating systems and there is no single development platform that can be applied universally. On the other hand, taking advantage of the available parallel processing power of the Clouds requires rewriting or redesigning many existing programs and algorithms. In short the current advancements in hardware ubiquity has not translated fully into the expected level of growth,

Figure 1.1: Spectrum of Computing Power

primarily due to the difficulty in programming.

Thus the intention of the this thesis is to address the programming problem. We present two research activities targeted towards different domains, in two parts. The primary theme running across these two research activities is the use of Domain Specific Language (DSL) as the key enabler.

1.1 MobiCloud : Programming hybrid applications

MobiCloud is a DSL and the associated tools that enable developing hybrid applications that have a mobile device based front-end and a Cloud based back-end. The motivation for such applications is discussed in Chapter 4. The prototype DSL is presented in Chapter 5. The evaluation and comparisons with existing frameworks are presented in Chapter 6.

1.2 SCALE : Generating applications for the scientists

Scalable Cloud based AppLication GenErator (SCALE) is a DSL based approach to data analysis in the life sciences domain. Chapter 9 presents the ongoing work in applying a DSL to Nuclear magnetic resonance (NMR) based metabolomics data analysis. The science of metabolomics is a relatively young field that requires intensive signal processing and multivariate data analysis for the interpretation of experimental results. A set of fundamental operators for NMR based metabolomics is proposed and an implementation of these operators using a DSL is presented. These operators are implementation independent, and can be used to easily and precisely describe the processing and analysis steps that led to research conclusions. The DSL is convenient to use for a domain scientist, and can be easily transformed into multiple target platforms.

Background

2.1 Cloud Computing

Cloud Computing has been defined and described in many different ways. The two well accepted definitions are from the National Institute of Standards and Technology (NIST) and University of California, Berkley.

Peter Mell and Tim Grance of NIST define Cloud Computing as a model for enabling convenient, on-demand network access to a shared pool of configurable and reliable computing resources (e.g., networks, servers, storage, applications, services) that can be rapidly provisioned and released with minimal consumer management effort or service provider interaction [1].

According to Michael Armbrust and others, Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services [2].

Cloud Computing is comprised of the following characteristics

1) On-demand, Pay-as-you-go and Measured Service

On-demand, pay-as-you-go and the self-service paradigm of Cloud Computing enables Cloud users to use the Cloud resources (such as computation, storage, and network) as needed without direct interaction with the Cloud service provider. The customer has no long term contract with the Cloud provider and usually is billed based on the his usage of only the Cloud resources that were allotted to him in that particular session.

2) Resource Pooling and Elasticity

The Cloud resources are pooled and dynamically allocated by the Cloud provider to multiple Cloud users based on their need (Resource pooling). The Cloud user has the ability to increase or decrease the allocated resources at any time and the Cloud quickly responds to meet these requirements (Elasticity). This provides the Cloud user the illusion of unlimited resources.

3) Network Access and Location-Independent Resources

The Cloud resources are accessed by the Cloud user through the network. The customer has no knowledge or control over the exact location of the provided Cloud resources.

2.1.1 Service models of Cloud Computing

1) Infrastructure as a Service (IaaS)

IaaS refers to delivery of computing resources as a service. IaaS includes virtualized computers with specified configuration such as processing power, storage, and network bandwidth. E.g., Amazon EC2.

2) Platform-as-a-Service (PaaS)

PaaS refers to delivery of a computing platform and solution stack as a service. PaaS includes programming languages, tools, and an application delivery platform hosted by the service provider to help development and delivery of end user applications. E.g., Google App Engine.

3) Software as a Service (SaaS)

SaaS refers to delivery of hosted software applications as a service. In SaaS service provider develops web-based software applications, and then hosts and manages those applications over the Internet for use by end-users. E.g., Google Docs.

Figure 2.1 illustrates various service models of Cloud Computing.

Figure 2.1: Service models of Cloud Computing

2.1.2 Deployment models of Cloud Computing

1) Public Cloud

In the Public Cloud, the Cloud infrastructure is made available to the general public and is owned by an organization selling Cloud services.

2) Private Cloud

In the Private Cloud, the Cloud infrastructure is operated solely for an organization. It is usually managed by the same organization and exist on premise (but in few cases it can be off premise and managed by a third party).

3) Hybrid Cloud

In the Hybrid Cloud, the Cloud infrastructure is a composition of public and private Clouds that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (E.g., Cloud bursting for load-balancing between Clouds) [1].

Figure 2.2 [3] summarizes Cloud Computing and illustrates the relationship between the deployment models, service models and application domains.

2.2 Mobile Computing

Mobile Computing is a generic term describing the application of small, portable, and wireless computing and communication devices such as Mobile phones and Personal Digital Assistants (PDAs) [4].

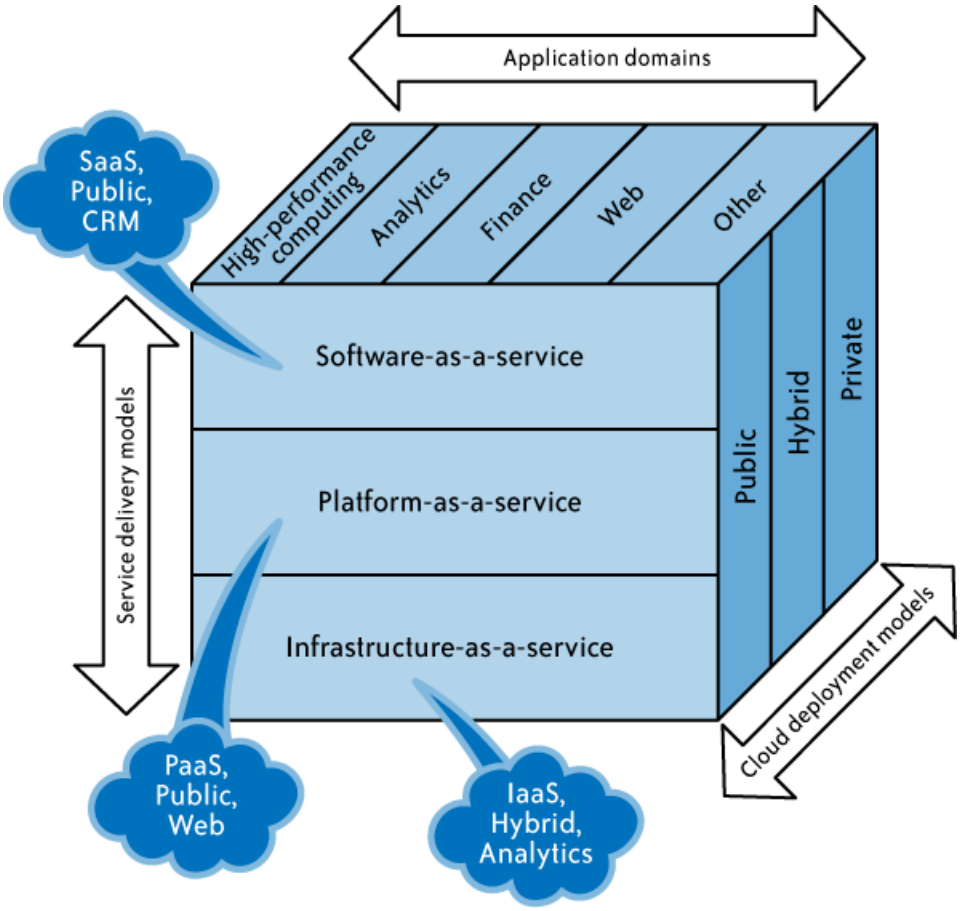


Figure 2.2: Overview of Cloud Computing

Earlier a mobile phone had limited capabilities and just allowed its user to make and receive telephone calls to and from the public telephone network. The modern mobile phones also support many additional services, and accessories, such as Internet access, e-mail, gaming, music player, camera, Multimedia messaging, GPS etc. These mobile phones that offer more advanced computing ability and connectivity than a basic feature phone are called Smartphones. Smartphones run complete operating system software providing a platform for application developers, making them mobile computers.

2.2.1 Types of Mobile Computing Devices

The term mobile computing device or just "mobile device" covers a wide range of electronic devices. The category of mobile devices includes the following devices, as well as others.

Personal Digital Assistant: Personal Digital Assistant (PDA), also known as a palmtop computer or pocket computer, is a mobile device that functions as a personal information manager. A typical PDA is composed of personal organizer and internet browser in a single device. E.g. Palm TX.

Smartphone: Smartphones combine both mobile phone and PDA into a single device. Smartphones can store information, install applications, along with the capabilities of mobile phone. E.g. Apple iPhone4.

Tablet Computer: A tablet computer, or simply a *tablet*, is a complete personal mobile computer, larger than a mobile phone or personal digital assistant, integrated into a flat touch screen and primarily operated by touching the screen. It usually uses an onscreen virtual keyboard or a digital pen rather than a physical keyboard. E.g. Apple iPad.

2.2.2 Mobile Operating Systems

A mobile operating system, also known as a mobile OS or handheld OS, is the operating system that controls a mobile device.

Android

Android is an open source mobile operating system developed and supported by Google and Open Handset Alliance (OHA). The OHA is a business alliance for developing open standards

for mobile devices that include Google and eighty other companies such as HTC, Dell, Intel, Motorola, Qualcomm, etc.

The Android OS kernel is based upon a modified version of the Linux kernel. The Android SDK provides the tools and APIs necessary to develop applications on the Android platform using the Java programming language.

The main strength of the Android operating system is its Open Source license. Application developers prefer android as it provides equal access and performance for all applications running on it. Android is more economical than other mobile operating systems. A disadvantage of Android is that it is prone to many security related risks such as hacking and viruses due to its open source nature.

iPhone OS (iOS)

iOS is a proprietary mobile operating system developed by Apple Inc and used only on Apple's devices such as iPhone, iPod touch, iPad, etc. Apple does not license iOS for installation on third-party hardware.

iOS is derived from Mac OS X and is based on Darwin operating system. iOS applications are written in Objective-C, a custom version of the C language by Apple. The iOS SDK which is used to develop native applications for the iOS can only be installed on Apple-branded computers. Although, the iOS SDK is free to download, in order to load the software on to the device, or release the software, one must enroll in the iPhone Developer Program that requires payment and Apple's approval.

iOS is known for its good user experience and ease of use. This has prompted a large

number of applications available for download. However, the drawback of iOS is that Apple retains tight control on applications that are allowed to run on it.

Blackberry OS

BlackBerry OS is a proprietary mobile operating system, developed by Research In Motion for its BlackBerry smartphones. The Blackberry SDK is used to develop applications on the Blackberry platform using Java Micro Edition (Java ME).

A strong point of Blackberry OS is its security and enterprise integration features. The main limitation of Blackberry OS is that it is considered not user friendly, in comparison to other mobile operating systems.

Palm webOS

webOS is a proprietary mobile operating system running on the Linux kernel, initially developed by Palm, which was later acquired by Hewlett-Packard. Palm webOS is designed to run on a variety of hardware with different screen sizes, resolutions and orientations, with or without keyboards and works best with a touchpanel [5].

Most applications are written in HTML and Javascript, but there is also a PDK (Plugin Development Kit) for developers wanting to go straight at the Linux kernel (for access to the GPU) [6]. webOS runs on *webOS phones* such as Palm Pre, Palm Pixi, etc.

The advantage with webOS is its tight integration with the Web and the enhanced multi-tasking capabilities. The disadvantage with webOS is that it has a relatively fewer number of applications developed for it and is fairly new.

Symbian

Symbian is an open source operating system (OS) and software platform designed for smartphones and maintained by Nokia. The Symbian platform was created by merging and integrating software assets contributed by Nokia, NTT DoCoMo, Sony Ericsson and Symbian Ltd., including Symbian OS assets at its core, the S60 platform, and parts of the UIQ and MOAP(S) user interfaces [7].

The strength of Symbian OS is that it requires minimum resources. The major weakness of Symbian OS is its limited capability in the enterprise space.

Windows Mobile

Windows Mobile (the latest version known as Windows Phone7 to signify a complete overhaul of the platform) is a mobile operating system developed by Microsoft for smartphones and mobile devices.

The positive aspect of Windows Phone7 is its integration with other Microsoft products and the usability improvements of the latest version. Microsoft operating systems however have a reputation of being more resource hungry and that may be seen as a negative aspect of this OS. Some of the previous versions have been criticized for their usability.

2.3 Domain Specific Languages (DSL)

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 [8]. 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 [9]. For example, mathematicians are quite familiar with specialized languages such as Matlab [10] that provide a convenient way to form Mathematics oriented programs.

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. [11] 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.

2.3.1 DSL Example

Unix shell scripts

Unix shell scripts are a good example of a domain-specific language for data organization. They can manipulate data in files or user input in many different ways. Domain abstractions and notations include streams (such as stdin and stdout) and operations on streams (such as redirection and pipe). These abstractions combine to make a robust language to talk about the flow and organization of data.

In practice, shell scripts are used to weave together small Unix tools such as *AWK*, *ls*, *sort* or *wc*. A simple shell script is illustrated in Listing 2.1.

Listing 2.1: Example Shell Script, to be executed with the Bash shell

```
#! /bin/bash
directory="./BashScripting"


# bash check if directory exists
if [ -d \$directory ]; then
  echo "Directory exists"
else
  echo "Directory does not exists"
fi
```

Other examples

The emergence of interpreted languages such as Ruby has 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].

2.3.2 Issues with DSLs

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. Even though many DSLs are treated *self-documented code*, they require a new learning effort to master and use DSL efficiently. The cost of designing, implementing, and maintaining a DSL as well as the tools required to develop with it (IDE) may be significant in some cases.

Part I

MobiCloud : Cloud Mobile Hybrid Application Toolkit

Introduction to Cloud Mobile Hybrid Applications

In the backdrop of the 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 the 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 ideal example of CMH application is the Google Goggles. Google Goggles is an image recognition application created for smartphones. An image of a place or a thing is captured using a smartphone's built-in camera and compared to millions of other images on the web, relevant results are delivered to the user. For example, a tourist can easily obtain information about the artist and the name of the painting in an art gallery using Google Goggles from his smartphone. The application can also be used to scan barcodes on products and compare its price by different sellers on the web. This is an ideal case for a CMH application as it

requires a portable and readily available mobile phone that can take a picture and use a Cloud that can process and do related analysis on the picture.

Another example of a CMH is an implementation of the *Privacy Score* algorithm [13] [14]. The 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 devices[1].

The present state of the art in mobile front-ends has changed from mobile-enabled websites 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 offers 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 platform forces mobile application developers to

[1]http://www.facebook.com/press/info.php?statistics

focus on only specific platforms and versions [15, 16]. 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 MobiCloud, therefore, is to provide a disciplined approach to overcome the above challenges. This 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 uses a single DSL script to describe it, can significantly reduce the 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.

The current prototype MobiCloud 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 Chapter 6.

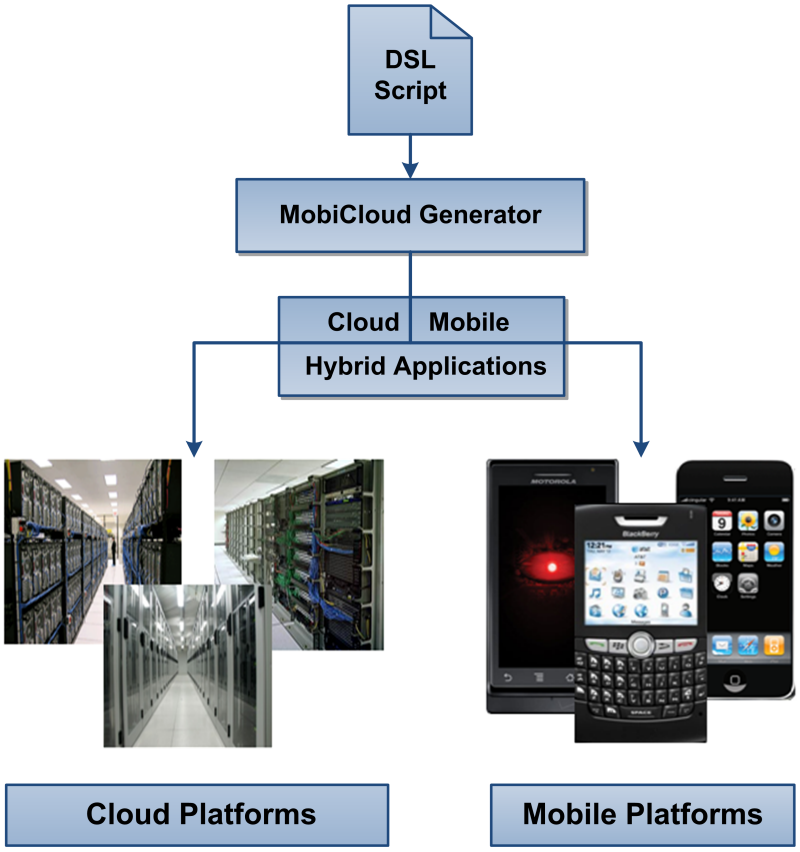


Figure 3.1: An overview of Cloud Mobile Hybrid application generation process

Motivation

4.1 Cloud and Mobile Convergence

The rapid expansion of mobile devices and Cloud computing has set the environment for applications that make the best use of both environments. The example of Google Goggles, explained in the Introduction (Chapter 3) is an ideal case of this convergence where a mobile device acts as the front-end but makes use of the extensive computing power of the Clouds to search through a large image index.

The next part of this section provides evidence of the rapid growth of the two segments, mobile devices and Cloud Computing.

4.1.1 Rapid growth of Smartphones

The global Smartphone device forecast from Frost & Sullivan[1] shows the number of smartphones nearly doubling in the next three years. Figures 4.1 and 4.2 are graphs plotted using the data from Frost & Sullivan illustrate this fact clearly.

[1] http://www.frost.com/

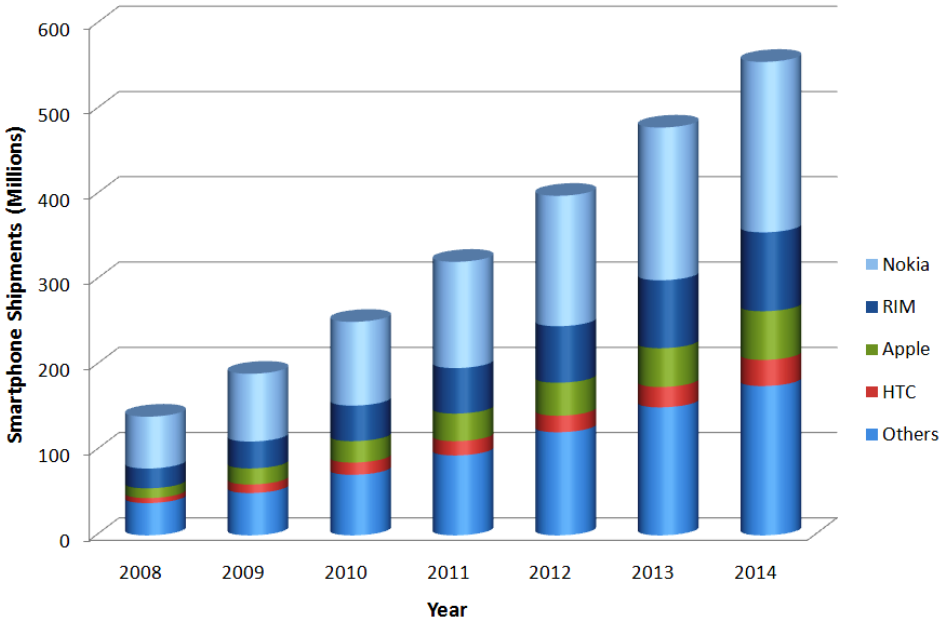


Figure 4.1: Global Smartphone Device Forecast

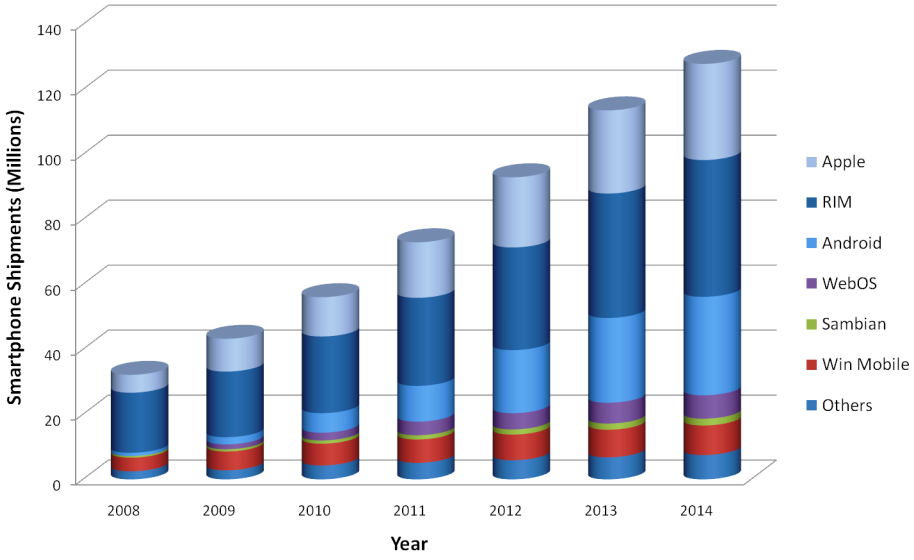


Figure 4.2: North America Smartphone Device Forecast

4.1.2 Adoption of Cloud Computing

Figure 4.3 shows Cloud Computing technologies market forecast by the Market Intel Group LLC (MiG) a popular market research company.

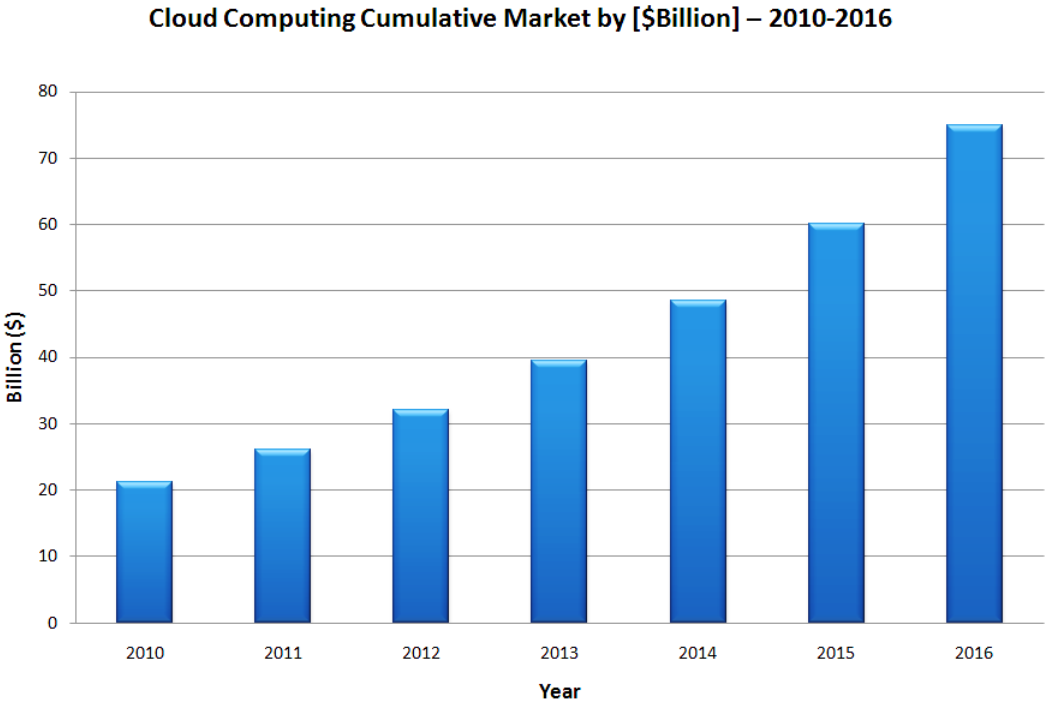


Figure 4.3: Cloud Computing Technologies Market Forecast

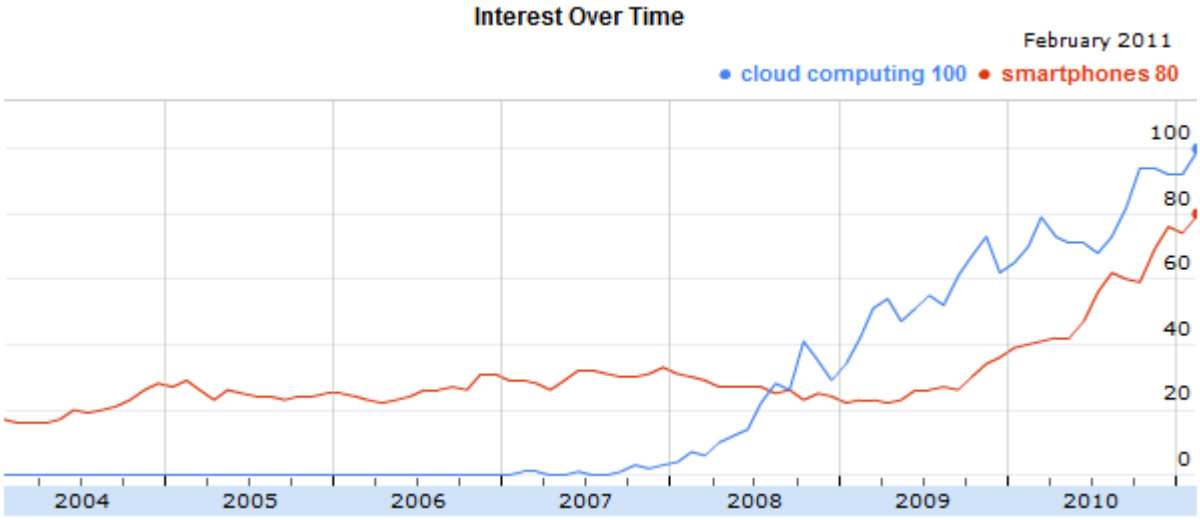


Figure 4.4: Google Trends - Cloud Computing and Smartphones

Figure 4.4 shows Google trends of Cloud Computing and Smartphones. The type of applications that take advantage of this rapid growth and convergence process are named *Cloud mobile hybrid applications*. The goal of such applications is to overcome the multiple short

comings of Smartphones and Cloud Computing by complementing each others capabilities.

The primary strengths of Smartphones are that they are portable and connected. They are equipped with many sensors, can connect to the Internet and can communicate with many devices, making them a compelling platform for many to create custom applications. Smartphones, however, are battery powered and have limited processing and storage capability.

Contrary to Smartphones, Clouds posses almost infinite processing and storage capability. They can be purchased in pay as you go manner making them economical. The weaknesses of smartphones are the strengths of Cloud and hence their convergence opens the door to the creation of a variety of new applications and services that are more personalized, and can be delivered to users at any time and any place.

4.2 Motivation for using DSLs in Hybrid Applications

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.

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 [17]. Rajapakse [18] 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 [19]. 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 as a serious hindrance to Cloud Computing adoption [20]. Some experts have also suggested that vendors may purposely promote the Cloud to be a heterogeneous patchwork of frameworks for business reasons [21].

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 [18]

[2] http://www.cesweb.org/

[22]. 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 applications (T_c) is

$$T_c = \sum_{i=0}^{m} \{MV_i\} \times \sum_{j=0}^{c} \{CV_j\} \tag{4.1}$$

Where m is the number of mobile platforms, c is the number of cloud platforms, MV_i is the number of versions of the ith mobile platform, and CV_j is the number of versions of the jth cloud platform.

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

$$T_g = \sum_{i=0}^{m} \{MV_i\} + \sum_{j=0}^{c} \{CV_j\} \tag{4.2}$$

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 4.1. The total number of generators required is 14 according to Equation 4.2. In practice, the number of required generators is lesser since some platforms are backward compatible.

This calculation highlights that without a clear development methodology, Cloud-mobile hybrid applications will remain an expensive and exotic option for businesses.

System Architecture

In the case of MobiCloud, we focused on providing a *sufficiently* high level of abstraction with predefined transformations, a *logical* DSL according to Greenfields categorization 2.3.

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) [23].

5.1 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*[1]. 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:
(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 5.1 illustrates the major components present in a MVC based design.

[1]http://www.rememberthemilk.com/

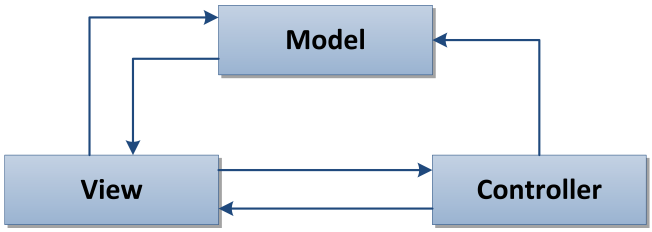


Figure 5.1: Model-View-Controller design pattern

Model represents a data-structure that holds a 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 according to the inputs through 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 Struts[2], and Ruby on Rails[3].

The DSL we have experimented with is designed according to the MVC principles and directly reflects the definitions of the relevant components. Listing 5.1 illustrates the major components of the language in BNF notation. The complete BNF specification of the grammar is available in the appendix A and also from the online resources [4]. This language has been developed by restricting the Ruby base language. Extending or restricting a base language is a known DSL design technique [9] and 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 [24]. The IBM Sharable

[2]http://struts.apache.org/
[3]http://rubyonrails.org/
[4]http://knoesis.wright.edu/mobicloud

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 5.1: 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 5.2 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 5.2: 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 Appengine[5], the

[5]http://appengine.google.com

: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 5.2 illustrates the mapping of the generated artifacts to the original MVC pattern. Figure 5.3 illustrates the actual artifacts generated in this case.

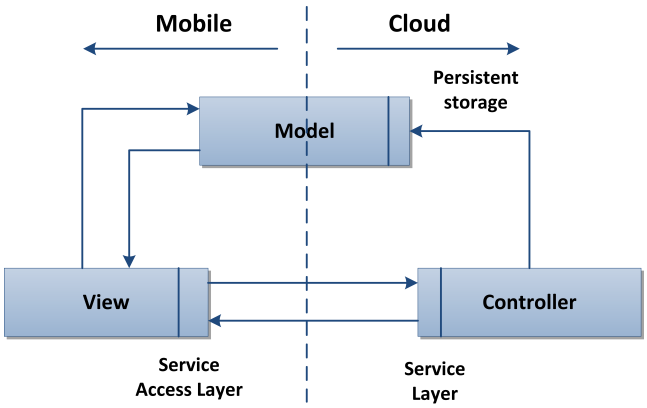


Figure 5.2: Mapping of Artifacts to MVC components

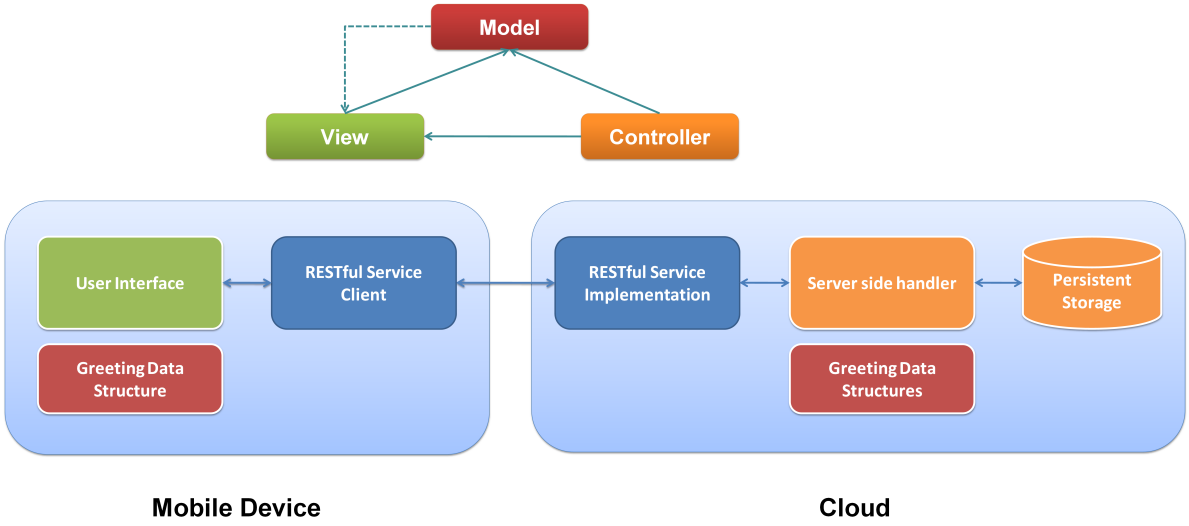


Figure 5.3: System Implementation Details

Listing 5.3 shows the use of language to create a todolist application, a simple but non-trivial application that is also useful. The evaluations (Chapter 6) show that although this application has only few more lines than the hello world, the resulting code is much larger.

Listing 5.3: The DSL script for the *todo list* application

```
recipe(:todolist) do
  #specific metadata for this app
  metadata({:id => 'random-id'})

  #models
  model(:todoitem, {:name => :string, :description => :string,
                    :time => :string, :location => :string})
  model(:user, {:name => :string, :bday => :string})

  #controllers
  controller(:todohandler) do
```

```
    action :create,:todoitem
    action :retrieve,:todoitem
    action :update,:todoitem
    action :delete,:todoitem
  end

  #views
  view :todo_add, {:models => [:todoitem],
                   :controller => :todohandler,:action => :create}
  view :todo_show,{:models => [:todoitem],
                   :controller => :todohandler,:action => :retrieve}
end
```

5.2 System Implementation

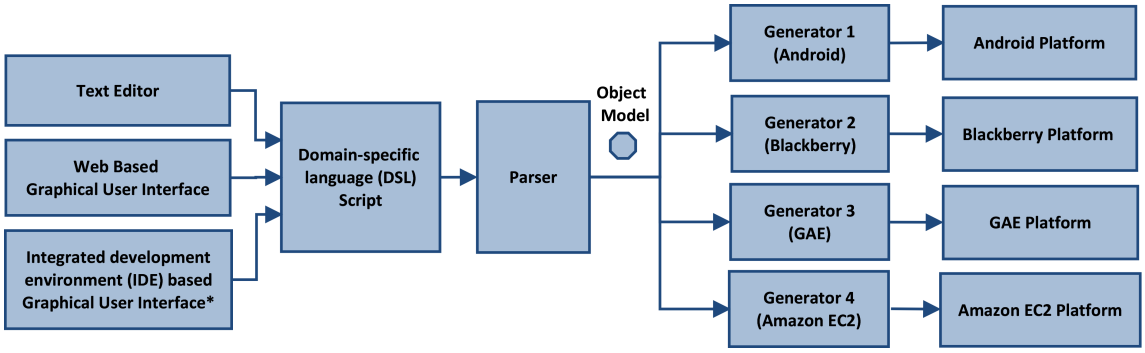


Figure 5.4: System Implementation Components and Flow

The system was implemented in Ruby in order to take advantage of the existing Ruby parser and interpreter. Figure 5.4 illustrates the 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.

An application in each platform is made up of multiple files. Each file has specific functionality associated with it. For example, UI related details in Android platform are contained in XML files, placed in specific directories. These files have to be generated and *stitched* together in a specific order for the operation of the application. The generator driver knows what files to generate in what order. The generator driver looks at the object model and the template and generates platform specific files in the appropriate format.

Listing 5.4 shows a code fragment from the Google App Engine generator driver where bean classes and servlets are generated by observing specific details from the object represen-

tation of the DSL script. The specific details are passed onto templates that generate the code.

Listing 5.4: Generation of bean classes and servlets for Google App Engine

```
# generate the beans
process_and_write_template_loop("bean.tmpl",
  "#{root_folder_name}/#{SOURCE_FOLDER}/#{BEAN_PACKAGE}",
     recipe_model.models.list,
     "", # suffix
     nil, # prefix
     :java, #type
     :model_file) # extension component symbol

# generate the servlets
process_and_write_template_loop("servlet.tmpl",
   "#{root_folder_name}/#{SOURCE_FOLDER}/#{SERVLET_PACKAGE}",
     recipe_model.controllers.list,
     "", # suffix
     nil, # prefix
     :java,
     :controller_file)

# generate the XML servelts
process_and_write_template_loop("xml.servlet.tmpl",
   "#{root_folder_name}/#{SOURCE_FOLDER}/#{SERVLET_PACKAGE}",
     recipe_model.views.list,
```

```
      "Xml",
      nil,# prefix
      :java, #type
      :view_xml_file)
```

Listing 5.5 illustrates the template for a bean class (representation of a model) for Google App Engine. Note the embedding of ruby code fragments that dynamically generate variable names, class names and method names based on the model details.

Listing 5.5: Bean Template file for Google App Engine

```
<%=java_header_comment%>
package <%=Template::Gae::BEAN_PACKAGE%>;

import com.google.appengine.api.datastore.Key;
import javax.jdo.annotations.IdGeneratorStrategy;
import javax.jdo.annotations.IdentityType;
import javax.jdo.annotations.PersistenceCapable;
import javax.jdo.annotations.Persistent;
import javax.jdo.annotations.PrimaryKey;

@PersistenceCapable(identityType = javax.jdo.annotations.IdentityType.
   APPLICATION)
public class <%=@name.to_s.capitalize%> {

    @PrimaryKey
    @Persistent(valueStrategy = IdGeneratorStrategy.IDENTITY)
```

```
     private Key key;
      public Key getKey() {
           return key;
       }

    <% @attrib_hash.keys.each do |key| %>
        <%
       javatype = Template::Gae::JAVA_MAPPINGS[@attrib_hash[key]]
       javaname = key.to_s
       javename_camel = javaname.capitalize
     %>
        @Persistent
        private <%=javatype %> <%=javaname%>;

       public <%=javatype %> get<%=javename_camel%>() {
           return <%=javaname%>;
       }

       public void set<%=javename_camel%>(<%=javatype %> local_<%=javaname
           %>) {
           this.<%=javaname%> = local_<%=javaname%>;
       }

    <% end %>
}
```

Evaluation

We present an evaluation based on code metrics of the generated artifacts for two programs in Table 6.1.

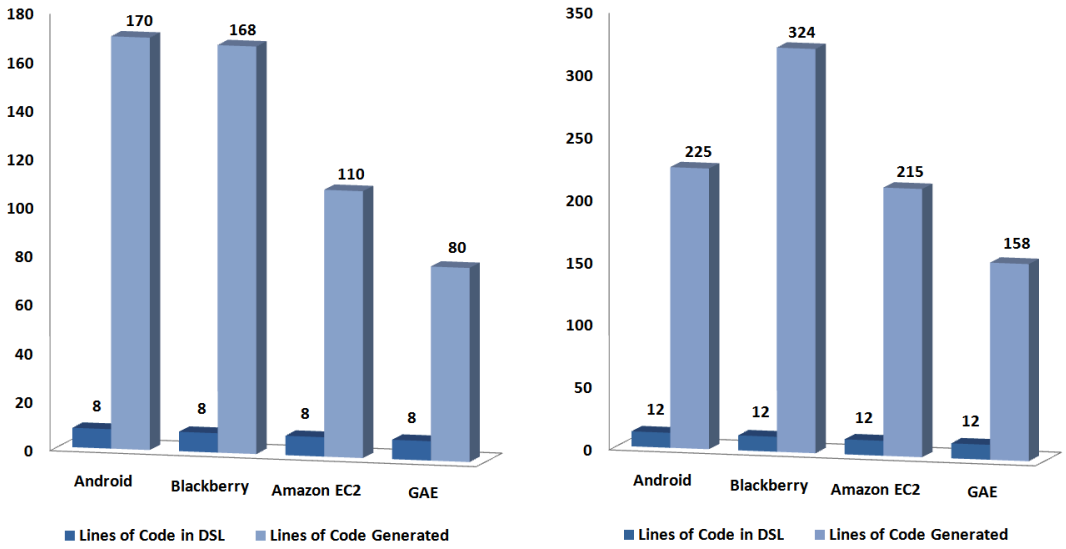
These metrics were obtained using the Eclipse Metrics plugin[1] and excludes non-java code (such as Android view XML files and build files). For both cases of Android and Google App Engine combination, developers have to write approximately 3% of the code they would have written otherwise. This is even lesser for the Blackberry and Google App Engine combination (2.5%). The number of classes and methods also indicates 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 Website[2].

[1] http://metrics.sourceforge.net/
[2] http://knoesis.org/mobicloud

Application	Lines of Code in DSL	Target platform	Lines of Code Generated	Number of Classes	Number of Methods
HelloWorld	8	Android Blackberry Amazon EC2 Google Appengine	170 168 110 80	9 6 4 4	4 8 10 8
Todolist	12	Android Blackberry Amazon EC2 Google Appengine	225 324 215 158	10 8 5 5	6 19 27 22

Table 6.1: Comparison of Code Metrics for the Generated Applications



(a) Hello World Application

(b) Todolist Application

Figure 6.1: Lines of Code Comparison

Online Toolkit

7.1 MobiCloud Toolkit

MobiCloud Online Toolkit along with documentation, tutorials and sample applications is available at http://mobi-cloud.org. The creation of Cloud Mobile Hybrid applications involves three simple steps.

Step 1: Writing the code in MobiCloud DSL (or editing auto-generated code).

Step 2: Selecting target platforms.

Step 3: Downloading the generated code for the selected platform.

These three steps are illustrated in figure 7.2, 7.3 and 7.4 respectively.

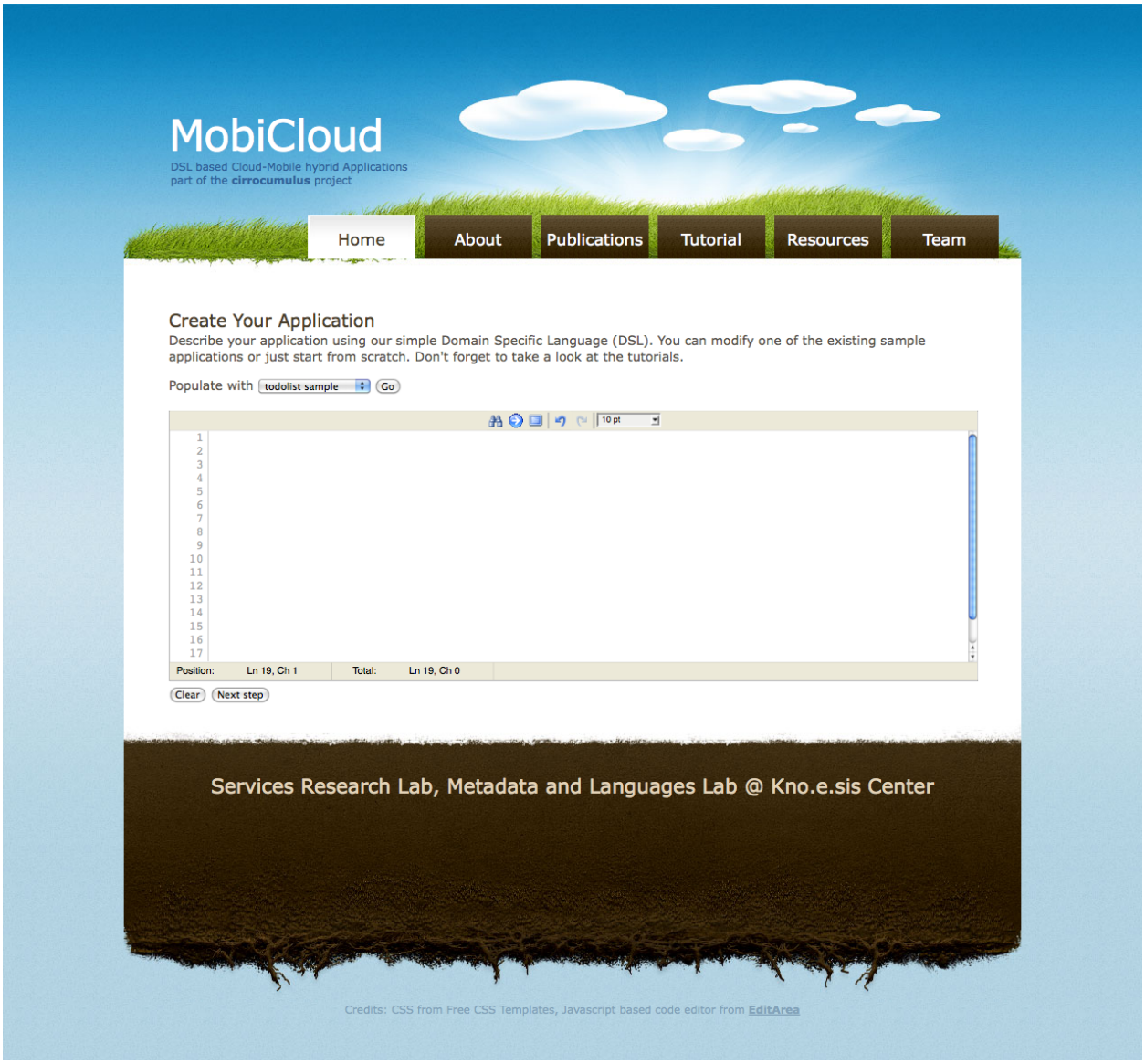


Figure 7.1: MobiCloud Online Toolkit Homepage

Figure 7.2: Writing the code in MobiCloud DSL

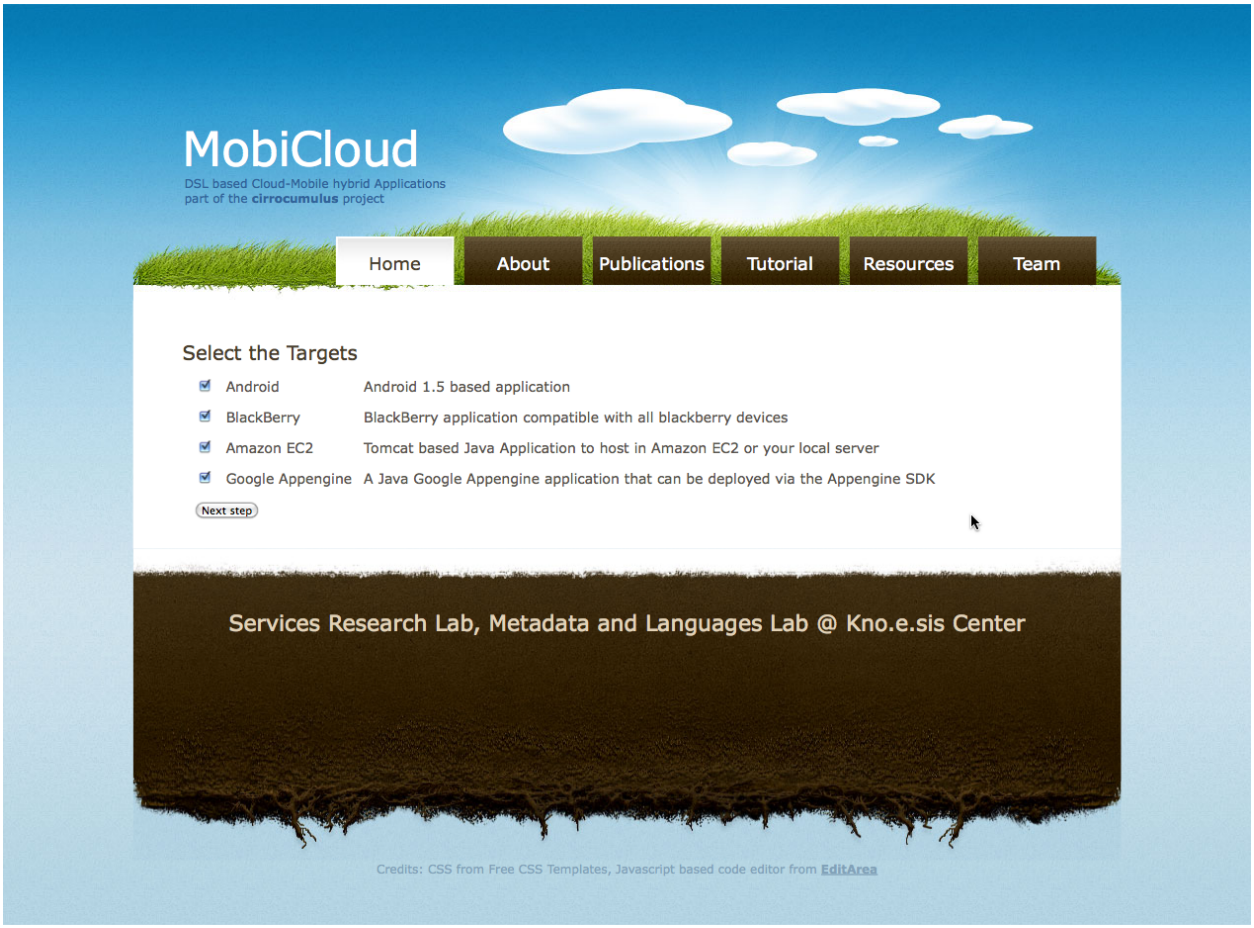


Figure 7.3: Selecting target platforms

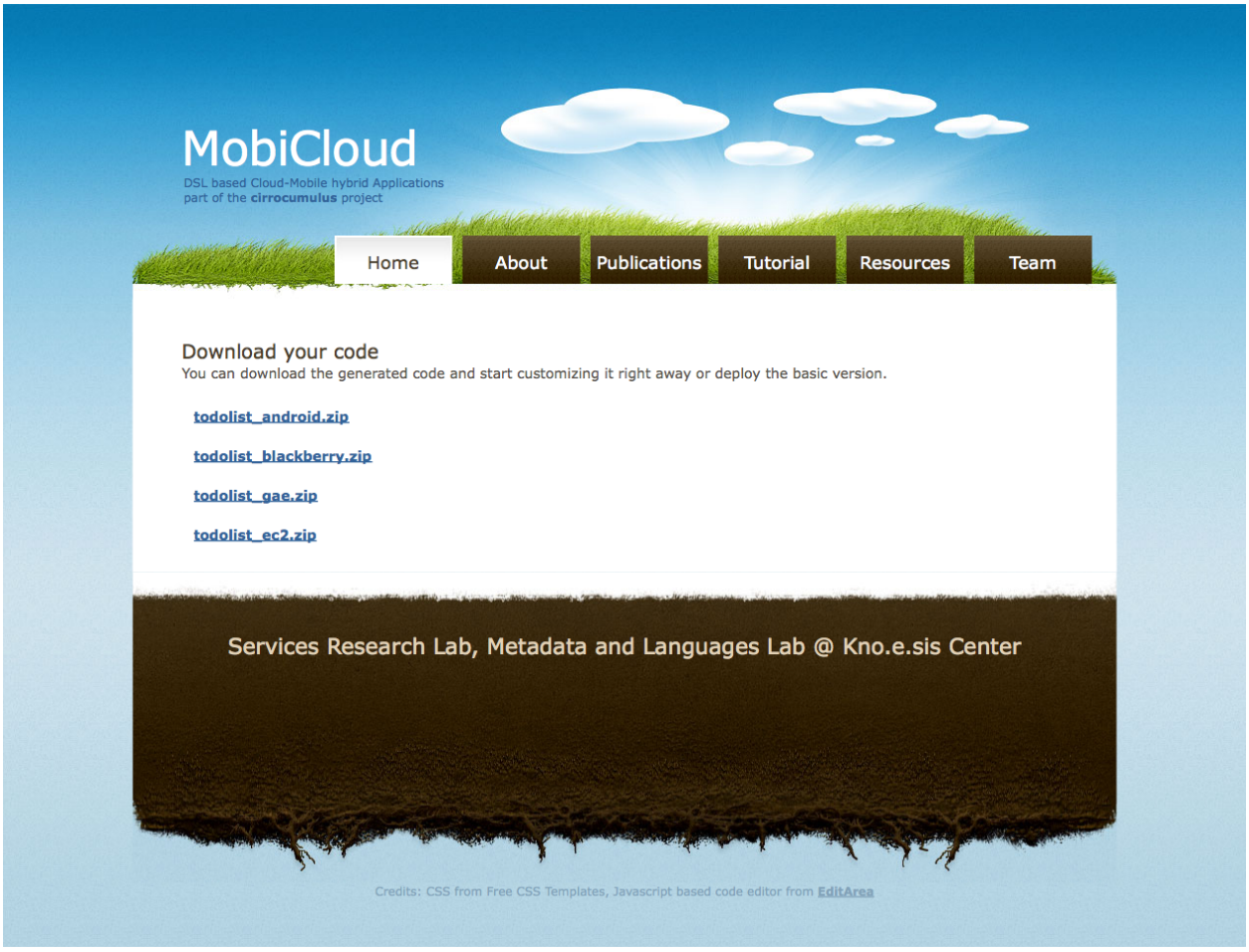


Figure 7.4: Downloading the generated code for the selected platform

Related Work and Discussion

8.1 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) [25]. GWT is an AJAX [26] 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 the 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 6.1.

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 App Engine and Amazon EC2	Amazon EC2 (via Heroku)	Google App Engine
Mobile Platforms	Android 1.5, Blackberry	None	None

Table 8.1: Feature comparison of MobiCloud, ISC and GWT

8.2 Discussion

8.2.1 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 [27] would improve the reach and the usability of the DSL.

8.2.2 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 later.
(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.

8.2.3 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-in actions using user defined functions. These actions may also be written in other DSLs such as PIGLatin [28] scripts. A possible way to incorporate custom actions is outlined in Section 8.2.4.2.

8.2.4 Language Extensions

8.2.4.1 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 [29]. 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 8.1. Some details such as namespaces are omitted in this listing for brevity. Listing 8.2 illustrates the reference being added to the Hello World application. Note the use of embedded code fragments to retrieve data from model objects.

Listing 8.1: 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 8.2: Using a Reference to XAML based UI template

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

8.2.4.2 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 8.3 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 [30]) should be available in the targeted Cloud platform.

8.2.4.3 Graphical Abstractions

The simplicity of the DSL enables it to be generated from a graphical representation similar to Yahoo! pipes [31]. 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 in [32].

Listing 8.3: 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
```

Part II

SCALE : Programming for Scientists

Introduction to Metabolomics

Metabolomics is the systematic study of the unique chemical fingerprints that specific cellular processes leave behind [33]. It deals with the measurement of metabolite concentrations and fluxes in various biological systems. Metabolites are the end product of cellular functions (such as sugars and amino acids) and are contained in biofluids such as blood and urine.

In order to analyze metabolites, one of the common method is to use the Nuclear Magnetic Resonance (NMR) spectrometer. Its function is to apply a varying magnetic field and record the resonance from the sample, resulting in a spectrum. In contrast to various other methods of metabolomic analyses, NMR spectroscopy is considered non-invasive, non-destructive, and requires little sample preparation [34].

The analysis of NMR spectrum is very complicated requiring numerous compute and data intensive algorithms ranging from signal processing to pattern recognition. This is because even the simplest (^{1}H) NMR spectrum of pure proteins, biofluids, or tissue may contain overlapping resonances that are in the order of thousands. Figure 9.1 shows a raw NMR Spectrum and figure 9.2 shows annotated NMR Spectrum after analysis.

The analysis of an NMR spectroscopic dataset involves the following five steps:

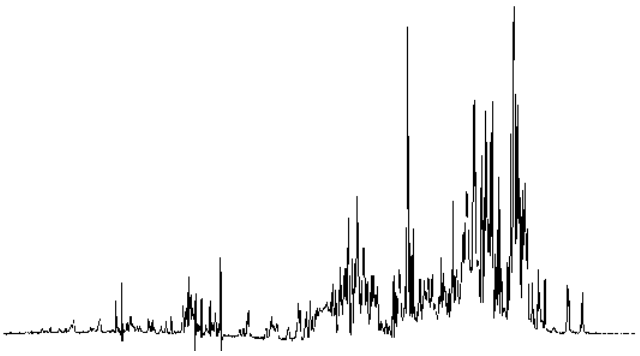


Figure 9.1: A Raw NMR Spectrum

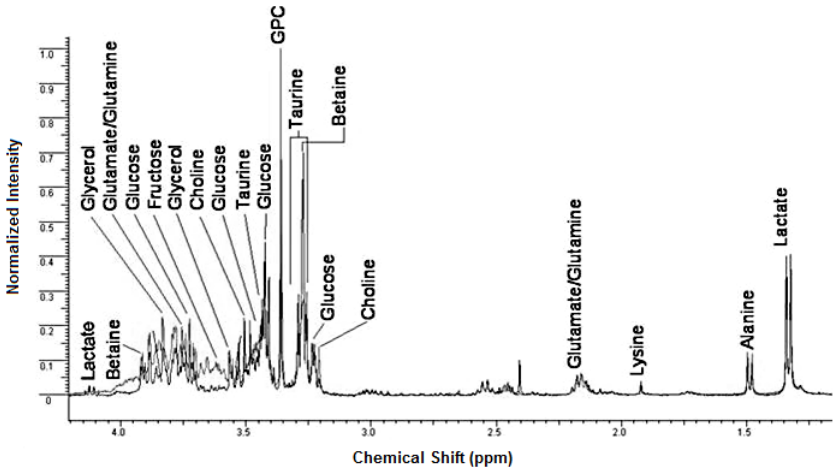


Figure 9.2: Annotated NMR Spectrum

1. **Standard post-instrumental processing:** Post-instrumental processing includes adjusting the results to correct machine based errors, for example, baseline correction [35] where the baseline of the spectrum may have shifted during the experiment.

2. **Normalization:** The purpose of normalization is to make the samples directly comparable to each other and performed on a per-spectrum basis [36].

3. **Quantification of spectral features:** Quantification of spectral features is to partition the spectrum in order to identify spectral patterns that correspond to compounds or known biological substances.

4. **Scaling:** Scaling is designed to control the weighting of features before a multivariate statistical or pattern recognition technique is applied [35].

5. **Multivariate statistical modelling:** Multivariate analysis is complex operations to refine the spectrum further and highlight required features to support pattern identification.

Further details on each of these steps are discussed in detail by Paul Anderson [37].

9.1 Formalization of operators in Metabolomics

Metabolomics is a relatively new field with huge impact and broad applicability in preclinical, clinical, environmental and diagnostic research areas. This expanding field has encompassed and helped in disease research, pharmaceuticals and nutrition. However, in contrast to the other fields in biological study, metabolomics has not benefited from integration and standardization. The lack of integration and standardization for metabolomics is exacerbated by the magnitude and complexity of the experimental data generated, the diversity of experimental instruments, and multitude of analysis techniques presently in use. The selection of any particular tool for analysis of this highly dimensional data can have a profound impact on the research conclusions. Metabolomics researchers employ a variety of proprietary and in house tools, many of which are not universally adopted. This results in fragmentation in the research community.

While there is still no common ground among all scientists on the best technique to process NMR spectroscopic data sets, basic operations that are needed in the data processing

can be identified. For example, different flavors of scaling and normalization are used in almost all NMR data processing tasks, thus they can be named fundamental operations on NMR spectroscopic data. This abstracted level of commonality can be exploited to standardize the field of metabolomics. Such standards will improve inter-lab communication and reduce fragmentation in the research community. Furthermore, a set of domain specific operators can become the basis for a DSL (as discussed in the next section) and enable the definition of the processing tasks, independent of the target platform.

9.2 Applying Cloud computing and DSLs to Metabolomics

9.2.1 Cloud Computing in Metabolomics

> *The newest 'omic science is producing results and more data than researchers know what to do with.*
>
> -Bennett Daviss

Metabolomics is a data driven field. As the quote by Bennett Daviss [38] exemplifies, the enormous loads of data collected by NMR experiments need timely processing to become useful. Sadly, metabolomics data analysis is highly time consuming given the complexity of the algorithms. The computations can often take days in a single computer. More so some analysis tasks require the data to be analyzed multiple ways, often with different algorithms at each step.

This is where the new trend in Cloud Computing becomes useful. Clouds enable the use of a large pool of computing resources on a pay-as-you-go basis. The applicability of

Clouds for NMR data has already being investigated, for example use of Hadoop and Matlab as experimented by Gunaratna et al. [39].

9.2.2 Use of DSLs in Metabolomics

As explained in Chapter 2, DSLs are indeed in use in the many scientific domains. Scientists and biologists are familiar with DSL driven scientific software tools that provide friendly environments for their particular needs. Matlab [10] is one such commercial software that provides specific data structures and modules that biologists need in their routine workflows. Scientists typically run Matlab in a desktop environment and hence they are constrained with respect to computational power. Moving to a distributed environment may require mastering a set of new technologies and many scientists are hesitant to move away from the convenience of domain specific tools such as Matlab.

Two observations are worth noting in this context:
(1) There is an increase in the available computing power and distributed computing tools. These tools however have sharp learning curves, a fact that often discourages scientists from adopting them.
(2) User friendly and domain specific tools are deemed important by scientists. The convenience of such tools is often preferred over their apparent lack of performance. The performance issues can often be alleviated by adding more computing resources rather than code optimization as outlined by the so called *Carbon vs Silicon* argument.

A more appropriate approach in this case would be to introduce a DSL that contain operators and concepts biologists can relate to. When this DSL is defined at a sufficiently

higher level, a DSL based program can be mechanically transformed to many implementations, including distributed implementations that support Cloud Computing.

Formalizing Fundamental Operators for Metabolomics

The definition of the fundamental operators for NMR-based metabolomics will provide a common language that will facilitate inter-lab communication by precisely described the processing and analysis. Some of these operators include:

- **Normalization (N):** This family of operators are performed on a per-spectrum basis to make the samples directly comparable to each other. Two common sub-operators of this family include Sum normalization (N_{sum}) and normalization by weight (N_{weight}).
- **Correction (C):** This family of operators remove errors introduced by measuring equipments such as baseline shift. Sub-operators of this family include baseline correction ($C_{baseline}$) and phase correction (C_{phase}).
- **Quantification (Q):** This family of operators reduce the dimensionality of the data and attempt to extract or approximate metabolite concentrations. Sub-operators of this family include binning ($Q_{binnnig}$) and targeted profiling ($Q_{targetedprofiling}$).

- **Scaling (S):** This family of operators control the weighting of features before a multivariate statistical or pattern recognition technique is applied. Sub-operators of this family include auto-scaling ($S_{autoscaling}$), pareto-scaling ($S_{paretoscaling}$), and mean-centering ($S_{meancentering}$)

- **Mining(M):** This family of operators selects the significantly responding metabolites/features for a given experiment. Sub-operators of this family include t-test (M_{ttest}), and partial least squares with variable selection (M_{pls}).

- **Visualize(V):** This family of operators output a visualized representation of the data and/or results. Sub-operators of this family include principal component analysis (V_{PCA}) and partial least squares scores plot (V_{PLS}).

- **Transformation(T):** This family of operators perform data transformations, such as Fourier transforms ($T_{fourier}$).

These operators operate on Matrices (S) or Vectors (s). For example $N_{sum} : s \rightarrow s$ where $s \epsilon S$.

The primary objective of these operators is to provide an uniform mathematical language to describe a NMR data processing task. As an example Equation 10.1 is a pure function oriented representation of doing a base line correction on Fourier transformed, phase corrected and autoscaled data set S where S' is the processed data set. This representation (and other equivalent symbolic representations) are suitable for scientific exchanges since they formerly indicate the operations and their order.

$$S' = C_{baseline}(Q_{autoscaling}(C_{phase}(T_{fourier}(S)))) \tag{10.1}$$

Since Equation 10.1 may not be intuitive as to the order of the operations, one may use an alternative representation that resembles a workflow. Equation 10.2 uses $\rightarrow$ to denote an input to a operator.

$$S' = S \rightarrow T_{fourier} \rightarrow C_{phase} \rightarrow Q_{autoscaling} \rightarrow C_{baseline} \tag{10.2}$$

Another convenient representation is the pseudocode style, as illustrated in Program 1, which is readily converted to the DSL described in Section 11.

Program 1 A Pseudocode representation of a processing task

$S_1 = T_{fourier}(S)$
$S_2 = C_{phase}(S_1)$
$S_3 = Q_{autoscaling}(S_2)$
$S' = C_{baseline}(S_3)$
$F = M_{pls}(S')$
$V_{pca}(S', F)$

Using a DSL to represent the Fundamental Operators

Now we illustrate the forming of a DSL to be used for metabolomics data processing. The operators discussed in Chapter 10 are used to form the following groups of functions.

- Loading data (csv, excel, text etc)
- Filtering (range filtering, value based filtering)
- Sorting (ascending , descending with respect to a column)
- Simple statistical functions (max,min, average)
- Signal processing algorithms (sum normalization, auto scaling)
- Writing data (multiple formats)
- Data transformations

Listing 11.1 outlines a simple *mini workflow* where a data file is loaded, sum normalized and written back to a new file. The variables raw_data_file and $normalized\_data\_file$ represent

DSL	`load_data_from_csv(raw_data_file)`
PIG	`LOAD '$raw_data_file' USING PigStorage(',') AS (colnum:int, value:double);`

DSL	`sum_normalize(data)`

PIG	`B = GROUP Data BY colnum;`
	`C = FOREACH B GENERATE group, SUM(Data.value);`
	`D = COGROUP Data by colnum inner, C by $0 inner;`
	`F = FOREACH D GENERATE group, FLATTEN (Data),FLATTEN (C);`
	`G = FOREACH F GENERATE $0,($2/$4)*100;`

Table 11.1: Translations of the DSL constructs and equivalent PIG implementation

the input and the output files respectively. Other function references are self explanatory. Equation 11.1 shows the mathematical representation of the script in Listing 11.1.

$$S' = S \rightarrow N_{sum} \tag{11.1}$$

Table 11.1 illustrates mapping of DSL functions and the PIG[1] implementation.

The first attempt in implementing a subset of these operators as a DSL was by restricting the Ruby base language. However, one may implement these operators by many other means, e.g. Matlab functions or C macros. We selected a DSL for its readability and the gentle learning curve.

The current implementation provides abstractions on top of Apache Pig, a platform for analyzing large data sets over the map-reduce framework, Hadoop [40]. Due to its underlying map-reduce architecture and its fault-tolerant file system, Hadoop is ideal for analyzing large spectroscopic data sets. The layered architecture of the implementation is illustrated in Figure 11.1. Note that since the language is based on fundamental operators, the workflow

[1]Apache Pig (or Piglatin) is a declarative MapReduce programming language used with Apache Hadoop.

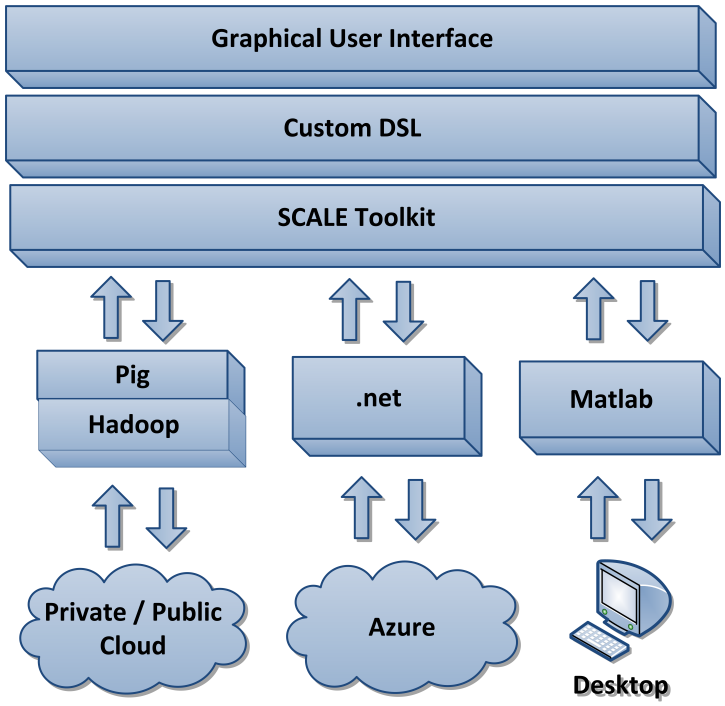


Figure 11.1: Layered Architecture of the Implementation

represented by the DSL can be converted to other forms (e.g. a Matlab based script running on a desktop or .net based program running on the Windows Azure Cloud) in a lossless manner. The SCALE toolkit contains the compiler/generators to convert the DSL script into concrete implementations that run on target platforms.

The drag and drop style graphical user interface, depicted in Figure 11.1, would be a layer of convenience over a textual language. This is important in the context of scientific workflows due to the high complexity of the workflows and the difficulty of visualizing them. The success of tools like Taverna is evidence to the effectiveness of drag and drop style workflow composers in the scientific computing domains.

Listing 11.1: Filtering and Sum normalization implemented using the DSL

```
# Load, do a sum normalization and store the results in a file

# load data

original_data = load_data(:raw_data_file,{:format => "csv"})

# sum normalize

normalized = sum_normalize(original_data)

# write the file

store_data(:normalized_data_file, normalized)
```

In order to contrast the effort in implementing this in PIGLatin, Listing 11.2 shows one of the generated PIG script for the above DSL using the online SCALE generator [2].

Listing 11.2: Sum normalization implemented using the PIG

```
Orignal_Data = LOAD '$raw_data_file' USING PigStorage(',')

AS (colnum:int, value:double);

Bnmlized = GROUP Original_Data BY colnum;

Cnmlized = FOREACH Bnmlized GENERATE group,SUM(Original_Data.value);

Dnmlized = COGROUP Original_Data by colnum inner, Cnmlized by $0 inner;

Fnmlized = FOREACH Dnmlized GENERATE group,

FLATTEN (Original_Data),FLATTEN (Cnmlized);

Normalized = FOREACH Fnmlized GENERATE $0,($2/$4)*100;

STORE Normalized INTO '$normalized_data_file' USING PigStorage (',');
```

There are two observations from these code comparisons.

(1) The PIGLatin script is not intuitive, i.e. its not obvious from the script as to its function.

(2) Creating the PIGLatin script requires a different pattern of thinking and reasoning that

[2]http://metabolink.knoesis.org/SCALE

needs to be obtained through practice.

It is clearly intuitive for the biologist to follow the first script rather than the second.

Related work and Discussion

The NMR based metabolomics, to our knowledge, has not been subjected to a thorough fundamental analysis. Some fundamental analysis and standardization that has been attempted has also stalled.

Introduction of a set of fundamental operators for NMR-based metabolomics is indeed a valuable generalization that provides a means of formal definition of the processing task. Although these operators may not be exhaustive, they can act as a basis to build domain specific languages and tooling that immensely benefits the scientists. These operators can be easily implemented to take advantage of Clouds and other scalable computing environments without exposing complex details of such environments.

Summary

The computing technology is improving every day but the difficulty of programming remains a serious problem. This problem is worsened by the introduction of more complex computing environments such as Clouds. This thesis covered two research activities aimed at solving the complexities in programming by using DSLs. While DSLs do not cover all cases, they are capable of providing solutions for majority of use case. Our research shows that using DSLs in both the consumer space (MobiCloud) and the academic space (SCALE) is viable and fruitful. There are many improvements to be made in the prototypical tools to make them full scale tool platforms but they have proved beyond doubt that DSLs indeed can be used to generate complex Cloud applications in multiple domains.

Bibliography

[1] P. Mell and T. Grance. The nist definition of cloud computing, version 15. *National Institute of Standards and Technology*, 2009.

[2] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50--58, 2010.

[3] T. Mather, S. Kumaraswamy, and S. Latif. *Cloud security and privacy: An enterprise perspective on risks and compliance*. O'Reilly Media, Inc., 2009.

[4] WordIQ.com. Mobile Computing, 2010. Available online at http://bit.ly/fHTZXx - Last accessed Feb 14th 2011.

[5] Overview of HP webOS, 2011. Available online at http://bit.ly/hppalm02-28 - Last accessed Feb 28th 2011.

[6] webOS vs. Android, 2010. Available online at http://bit.ly/webos02-28 - Last accessed Feb 28th 2011.

[7] Symbian, 2011. Available online at http://en.wikipedia.org/wiki/Symbian - Last accessed Feb 28th 2011.

[8] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26--36, 2000.

[9] D. Spinellis. Notable design patterns for domain-specific languages. *The Journal of Systems & Software*, 56(1):91--99, 2001.

[10] D. Hanselman and B.C. Littlefield. *Mastering MATLAB 5: A comprehensive tutorial and reference*. Prentice Hall PTR Upper Saddle River, NJ, USA, 1997.

[11] Jack Greenfield and Keith 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.

[12] E.M. Maximilien, A. Ranabahu, and K. Gomadam. An Online Platform for Web APIs and Service Mashups. *IEEE Internet Computing*, 12(5):32--43, 2008.

[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] A. Manjunatha, A. Ranabahu, A. Sheth, and K. Thirunarayan. A domain specific language based method to develop cloud-mobile hybrid applications. Technical report, Technical report, Kno. e. sis Center, Wright State University, 2010. Available online at http://knoesis.wright.edu/library/publications/MobiCloud.pdf: Last accessed August 27th, 2010.

[15] Raphael Moll. Knowing is half the battle. http://bit.ly/cvvWaR.

[16] Olga Kharif. Android's spread could become a problem. http://bit.ly/d0lHG8.

[17] Alex Johnson. Apps call, but will your phone answer? published online at http://bit.ly/7OfKeO : Last accessed August 27th 2010.

[18] Damith C. Rajapakse. Techniques for de-fragmenting mobile applications: A taxonomy. In *SEKE*, pages 923--928. Knowledge Systems Institute Graduate School, 2008.

[19] Economist Opinion Section. Clash of the Clouds. *The Economist*, 2009. published online at http://bit.ly/cBRAfB : Last accessed August 27th 2010.

[20] Rightscale.com. The Skinny on Cloud Lock-in. http://bit.ly/ rightscale_blog.

[21] D. Durkee. Why cloud computing will never be free. *Communications of the ACM*, 53(5):62--69, 2010.

[22] A. Manjunatha, A. Ranabahu, A. Sheth, and K. Thirunarayan. Power of clouds in your pocket: An efficient approach for cloud mobile hybrid application development. In *2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 496--503. IEEE, 2010.

[23] Jeff Atwood. Hardware is Cheap, Programmers are Expensive, 2008. Available online at http://bit.ly/avyNiN - Last accessed Jan 26th 2011.

[24] H. Conrad 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.

[25] E. Burnette. Google Web Toolkit--Taking the pain out of Ajax. *USA: The Pragmatic Programmers LLC*, 2006.

[26] J.J. Garrett et al. Ajax: A new approach to web applications. 2005.

[27] E.M. Maximilien, A. Ranabahu, R. Engehausen, and L.C. 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.

[28] 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.

[29] Microsoft. Extensible Application Markup Language. *Microsoft Developer Network (MSDN)*, 2008.

[30] D. Borthakur. The Hadoop Distributed File System: Architecture and Design.

[31] J.C. Fagan. Mashing up Multiple Web Feeds Using Yahoo! Pipes. *Computers in Libraries*, 27(10):8, 2007.

[32] E. Michael 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.

[33] Metabolomics, 2011. Available online at http://en.wikipedia.org/wiki/Metabolomics - Last accessed Feb 14th 2011.

[34] N.V. Reo. NMR-based metabolomics. *Drug and chemical toxicology*, 25(4):375--382, 2002.

[35] A. Manjunatha, P. Anderson, A. Ranabahu, and A. Sheth. Identifying and Implementing the Underlying Operators for Nuclear Magnetic Resonance based Metabolomics Data Analysis. In *Third International Conference on Bioinformatics and Computational Biology (BICoB)*. ISCA, International Society for Computers and Their Applications (ISCA), 2011.

[36] A. Craig, O. Cloarec, E. Holmes, J.K. Nicholson, and J.C. Lindon. Scaling and normalization effects in NMR spectroscopic metabonomic data sets. *Anal. Chem*, 78(7):2262--2267, 2006.

[37] Paul Edward Anderson. *Algorithmic Techniques Employed in the Quantification and Characterization of Nuclear Magnetic Resonance Spectroscopic Data*. PhD thesis, Wright State University, 2010.

[38] B. Daviss. Growing pains for metabolomics. *The Scientist*, 19(8):25--28, 2005.

[39] K. Gunaratna, P. Anderson, A. Ranabahu, and A. Sheth. A Study in Hadoop Streaming with Matlab for NMR data processing. In *2nd IEEE International Conference on Cloud Computing Technology and Science*, pages 786--789. IEEE, 2010.

[40] A. Bialecki, M. Cafarella, D. Cutting, and O. OMalley. Hadoop: a framework for running applications on large clusters built of commodity hardware. 2005. Avialable online at http://hadoop.apache.org.

Appendix A: Complete BNF Grammar for the CMH Language

Listing A.1: Complete BNF Grammar for the CMH Language

```
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
ARGLIST   : IDARG
          | '(' SYMBOL ',' HASH ')' | SYMBOL ',' HASH
IDARG   : '(' SYMBOL ')'| SYMBOL
HASH    : HASHA | HASHB
HASHB   :'{' HASHA '}'
HASHA   : HASH_ITEM
```

```
    | HASH_ITEM ',' HASHA

HASH_ITEM : SYMBOL '=>' IDENTIFIER

    | SYMBOL '=>' numeric

    | SYMBOL '=>' SYMBOL_LIST

    | SYMBOL '=>' STRING

SYMBOL_LIST : SYMBOL_LISTA | SYMBOL_LISTB

SYMBOL_LISTB  :'[' SYMBOL_LISTA ']'

SYMBOL_LISTA  : SYMBOL

    | SYMBOL ',' SYMBOL_LISTA

-------------------------------------------------------------

SYMBOL    : ':' IDENTIFIER

STRING    : '"' any_char* '"'

    | ''' any_char* '''

IDENTIFIER  : [a-zA-Z_][a-zA-Z0-9_]*
```