International Journal of Pervasive Computing and Com...

Model-driven framework to support evolution of mobile applications in multi-cloud environments

Nacha Chondamrongkul

Received 20 January 2016
Revised 20 January 2016
Accepted 7 July 2016

Model-driven framework to support evolution of mobile applications in multi-cloud environments

Nacha Chondamrongkul
School of Information Technology, Mae Fah Luang University, Chiang Rai, Thailand

Abstract

Purpose – The development of mobile applications in multiple clouds environment is a complex task because of the lack of platform standards in cloud computing and mobile computing. The source code involves various proprietary programming libraries for different platforms. However, functionalities are inevitably changed over time, as well as the platform. Therefore, a great deal of development effort is required, when changes need to be made at functional and platform level. This paper aims to propose SIMON, a framework that eases complexity of the development to support software evolution.

Design/methodology/approach – SIMON shields the developer from the complexity of mobile and cloud platforms in the development of mobile applications in multiple clouds environment. The framework uses model of application design to automate the development and support execution of mobile applications in system environment that needs integration to the number of data sources located on multiple clouds. The framework is composed of prefabricated components that support function changeability and platform adaptability.

Findings – The framework is examined with the development of a sample application. After it is evaluated with scenarios that involve changing at functional and platform levels, the result shows significant reducing of the development effort by comparing with the other approaches.

Originality/value – The framework facilitates the implementation of mobile applications in the software system that involves integration to multiple clouds, and it supports software evolution with lesser development effort.

Keywords Mobile computing, Cloud computing, Model-driven development, Platform heterogeneity, Software evolution, Software maintenance

Paper type Research paper

1. Introduction

Mobile devices are generally miniature in nature and light weight to support mobility. These general characteristics of mobile devices impose computational constraints such as low computational power and limited memory. This requires changes on both hardware and software level. However, software-level change would be more effective to enable mobile devices to achieve unlimited computational resources (Kemp *et al.*, 2009). Therefore, cloud computing fits to pursue this goal by enabling mobile applications to remotely access necessary resources that are manageable by cloud computing. For data-oriented applications, the mobile application requires intensive unidirectional flow of information to data storage on the cloud and yet maintain various security attributes such as integrity, confidentiality and availability through different security

International Journal of Pervasive Computing and Communications
Vol. 12 No. 3, 2016
pp. 332-351

1742-7371
DOI 10.1108/IJPCC-01-2016-0003

mechanisms (Unhelkar and Murugesan, 2010). In the enterprise infrastructure, both private and public clouds are concurrently used to serves different kind of data. The private cloud hosts sensitive data, while the public cloud hosts non-sensitive data (Erbes *et al.*, 2012). Therefore, mobile applications in multiple clouds environment (MAIMC) need a number of integrations to diverse cloud-based data storage.

There is a lack of standards at the platform level. This brings up issues of platform heterogeneity to the development of MAIMC. For mobile application development, different versions of an application need to be developed for various operating systems that different mobile devices in the market run. The development of a mobile application is obligated to a specific programming language and development platform for an operating system. Likewise, the integration to multiple clouds involves multiple proprietary platforms which have technical variations. Therefore, the development of MAIMC becomes a complex task that needs specialized knowledge of both cloud and mobile platforms (Khan *et al.*, 2014). As a result, the source code of MAIMC becomes tightly coupled to both mobile and cloud platforms; this consequently leads to a major obstacle when any changes need to be made (Sanaei *et al.*, 2014). The data structure and related functionalities are inevitably changed over time during the maintenance phase, as well as the specification of platform. Therefore, software evolution is a challenging topic in a software system of MAIMC that tightly integrates with a number of platforms.

To ease the complexity in the development of MAIMC, the proprietary programming of mobile and cloud platforms have to be abstracted and concealed from the developer (Khan *et al.*, 2014). There are existing approaches proposed from research groups to developer communities. There these approaches aim to make the development independent from either mobile platform or cloud platform. The combination of these approaches can possibly be used to develop MAIMC. However, we found limitations and issues. Therefore, this paper proposes a software framework referred to as SIMON. SIMON automates the development and helps execution of MAIMC. The framework applies two software development techniques: component-based software engineering (CBSE) and model-driven engineering (MDE). Both techniques pursue the same goal: to reduce development and maintenance cost. MDE shields developers from the complexity of the construction of the software system by using high-level models in every phase of the software development (Deursen *et al.*, 2007). CBSE helps developers to construct large system software by assembling independent and reusable software components (Szyperski, 2002). Most of the existing MDE approaches introduce code generation to automate repetitive coding. However, developers are still left with daunting manual tasks such as deployment and runtime performance tuning, which are still tied to the source code and involved configuration artifacts. Therefore, SIMON does not generate code, but it assembles prefabricated components that are pre-deployed and process model at runtime, to give application functionalities to the user. The framework provides an abstraction layer on top of cloud and mobile platforms, so the functionalities can be changed by changing the models of application design. As a result, the developer can focus on the application design without being concerned with complexities in the platform, and the cost of development consequently decreases. To support platform evolution (Deursen *et al.*, 2007), SIMON provides programming libraries which are extensible to support new mobile and cloud platforms for the purposes of reducing development effort, when platform migration is required.

The rest of the paper is organized as follows. Section 2 addresses MAIMC and highlights challenges and issues when existing techniques are used to implement

MAIMC. Section 3 introduces proposed framework, architecture and mechanism. Section 4 presents sample implementation and the evaluation result compared to the conventional approach. The paper concludes in Section 6 with future research direction.

2. Mobile application in multiple clouds

The architecture of MAIMC is shown in Figure 1. The development of MAIMC typically involves three major parts:

(1) Mobile application.

(2) Representational State Transfer (REST)-based backend service on the cloud.

(3) Integration links between mobile application and backend service.

The interoperability becomes a challenge because collaborations have to be done between various mobile devices and number of services located on multiple clouds. Also, the integration to diverse data sources is a key challenge for the mobile application development (Unhelkar and Murugesan, 2010). MAIMC needs to access both local data storage and remote storage service managed by public cloud vendors such as Amazon Web service (AWS)[1] and Google App Engine (GAE)[2]. The cloud vendor provides REST-based interface and application programming interface (API) library that allow the application client to use their services on the cloud. However, different cloud vendors offer different specifications of service interfaces with technical variations such as data structure and query methods. Therefore, the development of MAIMC requires a great amount of effort to apply a number of API libraries provided by cloud platforms in the source code of mobile application for a mobile platform (Flores and Srirama, 2014). Suppose the number of involved cloud platform is P_c, and P_m is the number of mobile platform to support, the number of integration links between mobile application and cloud is $P_m \times P_c$, so the total development effort of a MAIMC can be calculated as follows:

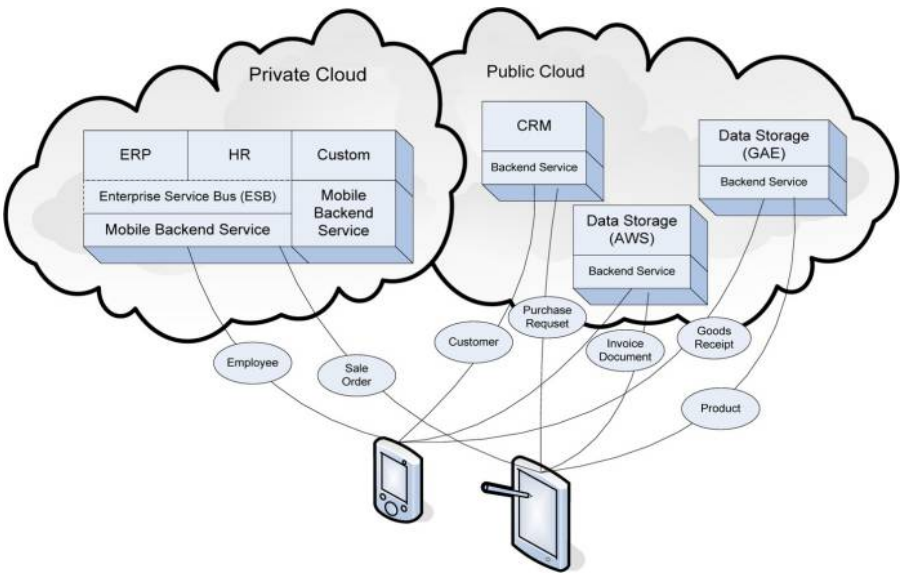

Figure 1.
Mobile application in multi-cloud architecture

$$T_e = \sum_{j=0}^{P_m} T_{m_j} + \sum_{j=0}^{P_c} T_{c_j} + \sum_{j=0}^{P_m \times P_c} T_{i_j}$$

T_m is the amount of effort taken to develop the mobile application for a specific mobile operating system, while T_c is amount of effort to develop backend services on different cloud platforms, and T_i is the amount of development effort for each integration link. The development effort for MAIMC strongly depends on the number of cloud platforms, the number of mobile operating systems it is compatible with and the total number of integration link between mobile application and clouds.

2.1 Vendor lock-in

The major cloud vendors offer storage service such as GAE's *Datastore* service and AWS's *DynamoDB*. A primary goal of these cloud-based storage services is data management which is the same purpose that relational database management system (RDBMS) serves. These storage services are usually NoSQL database and allow managing data remotely through REST-based interface or API library. However, there are technical variations which intensify issues in software evolution (Sanaei *et al.*, 2014). These technical variations can be described as follows:

- *Data management*: For example, *DynamoDB* manages a data entity as *Table*. There are *hash primary key* and *range primary key* which have to be manually defined to *Table,* to identify uniqueness among data records. The read and write operation's throughput need to be specified, when creating a *Table*. While GAE's *Datastore* manages a data entity as *Entity* with *Key* that is automatically generated to identify uniqueness.
- *Pricing model*: The cost of using *DynamoDB* depends on read and write throughput parameters defined to *Table*, while the cost of using *Datastore* depends on data record and transaction size.
- *Data type*: *Datastore* supports integer, floating number, string, date and binary data type, while *DynamoDB* supports only number and string data type.
- *Query method*: *DynamoDB* provides two methods, namely, *Query* and *Scan*. *Query* has better speed of data retrieval because it searches data by specifying value of *hash primary key* and *range primary key*, while *Scan* examines every record of data entity for matching set of conditions. *Datastor* provides query method that takes comparison condition and find a matching record.

These differences between cloud-based services pose semantic and syntactic dependencies from mobile application to the backend service. Therefore, the modification of cloud-based service can cause ripple effect (Bass *et al.*, 2004) to the mobile application. For the mobile application to run correctly, the signature and semantic of cloud-based storage services provided by cloud vendor and invoked by mobile application must be consistent with the assumptions of mobile application. Any changes in cloud services consequently require changing on the mobile application.

To develop a native mobile application, the developer needs to use a specific development platform for a particular operating system. Likewise, the public cloud vendor uses their own proprietary API to access services on the cloud, so the MAIMC has a major problem known as vendor lock-in which is the state when the software implementation is tight to specific

mobile and cloud platforms (Sanaei *et al.*, 2014). This affects two software quality attributes that influence software evolution, namely, changeability and adaptability. The changeability is the ability to make functional change, and adaptability is the ability to change to new specifications of operating environments.

2.2 State of the art

The vendor lock-in problem requires a platform-independent approach which allows different platform of mobile application to integrate with number of backend services on different cloud platform (Sanaei *et al.*, 2014). To achieve platform independence, the platform-specific details have to be separated from the platform-independent details to effectively support software evolution such as platform adaptability (France and Rumpe, 2007) and function changeability.

For mobile application development, different proposals have been proposed to achieve mobile platform independence such as DIMAG (Miravet *et al.*, 2013), PhoneGap[3] and Appcelerator[4]. DIMAG framework allows developers to declare specification of client-server details for a mobile application in XML and capable of generating both backend service and mobile application for several target platforms. The code generation of server side services is only compatible with Java EE platform. Also, it is lack of concrete approach that enables access to storage service on the public cloud platform. PhoneGap is an open source framework that helps developing mobile application with standard Web technologies. The user interfaces are rendered through embedded Web browser component which is available on native mobile operating systems. This enables the application to be portable across different mobile platforms. PhoneGap provides JavaScript API libraries that are able to access different sensors on device and remotely call backend services. However, PhoneGap does not address backend service development. Appcelerator Titanium is a platform for developing mobile applications across different mobile platforms. The developer writes source code of JavaScript containing specification of user interfaces and behaviors. The source code can later be transformed into native mobile application by the platform. The platform facilitates the application testing task with an ability to automate testing on different operating systems and devices. Appcelerator also provides cloud-based data storage service which can be accessible through REST-based API. The Appcelerator's cloud-based storage service is also known as *Backend as a Service* (*BaaS*) in the mobile developer communities. There are similar *BaaS* vendors such as Kinvey[6] Parse[6], Firebase[7] and Dreamfactory[8]. *BaaS* aims to facilitate mobile application developer in the development of backend service, which involves knowledge of infrastructure configuration and the number of proprietary APIs to manage data on the cloud or local data storage. *BaaS* vendors usually provide user-friendly tools that allow developers to define data structure and different service configuration. It provides REST-based API to manage data with some additional features such as user management, push notification and integration to other data sources. Although *BaaS* shields developers from the complexity of backend service development, the application is still locked-in to *BaaS* vendors. This poses a risk because the vendor may change their pricing policy or end their businesses.

Because different vendors provide cloud-based storage services that can be accessible through proprietary APIs, the development of backend services become tightly coupled with APIs which may suffer from any changes made by the cloud vendor. There are several existing open source projects that aim to solve cloud dependency problems such as *Daesin Cloud*[9], Apache Libcloud[10] and Apache

jclouds[11]. These projects provide unified API libraries that abstract differences between proprietary cloud's APIs; however, they can be used by a specific programming language which pose limitations to use on some mobile platforms. For instance, the development of iOS and Windows Phone uses Objective C and C#, respectively, as a programming language, but *Daesin Cloud and Apache jclouds* only support Java, while Apache Libcloud support Python. Therefore, they cannot be used in native mobile applications on iOS and Windows Phone. In Flores *et al.*, the authors propose mobile cloud middleware (MCM) (Flores and Srirama, 2014) which is an intermediary between mobile applications and cloud-based services by abstracting APIs of different cloud vendors. It aims to decrease offloading times to the cloud from the mobile devices by reducing the number of mobile-to-cloud interactions. Although MCM can significantly ease complexity in the development of cloud-based backend services, it does not deal with the development of mobile applications.

The combination of existing approaches could potentially achieve true platform independency in the development of MAIMC. For instance, we used Apache jclouds as a cloud abstraction library to develop backend services that manage data on multiple cloud-based storage services from different vendors such as *AWS S3* and *Microsoft Azure Blobstore*. We used *Appcelerator Titanium* to develop a cross-platform mobile application. The integration to backend services is developed using *Titanium.Network.HTTPClient* library. The development effort of MAIMC decreases because less effort is required for the development of mobile application and back-end service integration. The mobile application can interoperate data stored on different clouds. However, we found issues as follows:

- The cloud abstraction library only supports particular cloud-based services provided by some vendors without any solution addressed to support the integration to other cloud-based services or local data sources.
- The cloud abstraction library requires distinct configuration details to be made for different cloud platforms, so custom source codes for each integration to cloud is still existing in the mobile application.
- Although the development effort decreases, the total effort depends on the complexity of cloud abstraction library and cross-mobile development platform.The total development effort can be calculated as follows:

$$T_e = T_m + \sum_{j=0}^{P_c} T_{c_j} + \sum_{j=0}^{P_c} T_{i_j}$$

T_m is the amount of effort taken to develop a mobile application using cross-mobile development platform, whereas P_c is the number of clouds and T_c is the amount of effort to develop services on different cloud platforms using cloud abstraction library. T_i is the amount of effort to develop integration to each backend service. Therefore, T_i relies on the number of backend services to integrate with: if we need to make any changes in data structure and its involved functionalities, the source code of both backend service and mobile application will need to be substantially modified.

3. Proposed framework

SIMON is a software framework deployed on top of cloud and mobile platform to automate the development and help execution of MAIMC in the heterogeneous

environment, which involves mobile platforms and clouds platforms. The framework supports two fundamental data-oriented functionalities: information acquisition and information presentation (Mahjourian, 2008). SIMON is based on MDE, so it comprises prefabricated components that process high-level model of the application design. With SIMON, the developer defines three essential design aspects of data-oriented application in *application blueprint*, namely, *Data Model*, *Security Policy* and *User Interface Configuration*. *Data Model* defines data structure design which contains specification of data entities and their relationship. *Security Policy* is a set of security rules that define different levels of data access for different user roles. *User Interface Configuration* defines how data are presented and acquired through the user interface. The framework is developed with Java, an object-oriented programming language, so the framework's programming units are extendable and reusable. The framework can be deployed on any J2EE (Java 2 Enterprise Edition) technology (Singh *et al.*, 2002) compatible application server such as cloud-based J2EE runtime of AWS Beanstalk. The performance can be tuned at the application server level. For example, more worker nodes can be added to the application server to support more volumes of transaction.

As depicted in Figure 2, the architecture of SIMON is divided into three layers. The foundation service layer contains components that generally serve data management such as local relational database system and remote cloud-based storage service. The Application layer contains three core components, namely, Data Runtime Engine (DRE), Security Runtime Engine (SRE) and Blueprint Engine (BE). These core components work together to serve functionalities according to designs in the application blueprint. DRE interoperates data at different data sources including cloud-based storage and RDBMS at local infrastructure. DRE provides remote services which are for the purpose of performing typical data-oriented operations such as insert, update, query and delete. SRE provides remote services that enforce *Security Policy* in the application blueprint; it authenticates user credentials, verifies user roles and validates whether a user has enough privilege to access to the data. BE is a main component that manages the application blueprint and serves necessary information of application design to the

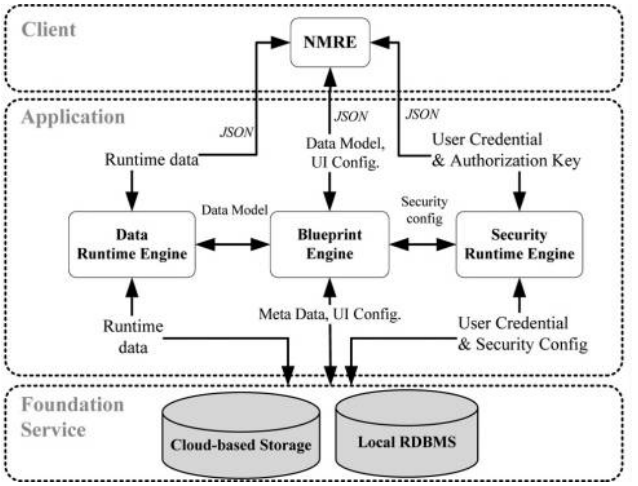

Figure 2.
SIMON's architecture

other components. The components within application layers can be deployed either on public cloud or local infrastructure. The mobile application is executed by native mobile runtime engine (NMRE) located at the client layer. NMRE is a runtime engine developed on top of a native mobile platform such as Android, iOS and Windows Phone. NMRE is composed of prefabricated components that are responsible for presenting application functionalities to the user by using necessary services provided by core components.

3.1 Framework in action

The core components consist of programming units that provide remote services as REST API (Richardson and Amundsen, 2013). These remote services exchange information in the format of JSON (JavaScript Object Notation). JSON is compact size and convenient for transmitting over network, so it is suitable for bandwidth-limited of communication link between remote services and mobile application. Different parts of the application blueprint are managed by different core components. *Data Model* and *User Interface Configuration* are managed by BE, whereas the *Security Policy* is managed by SRE. BE wraps *Data Model* and *User Interface Configuration* together as an artifact referred to as *Client artifact*. During initialization of mobile application, NMRE retrieves *Client artifact* from BE and stored locally on its embedded storage at the mobile device.

During runtime, NMRE renders user interface according to *User Interface Configuration*. SIMON supports data-oriented functionalities with two types of user interface, namely, data grid and data form to serve information presentation and acquisition, respectively. Figure 3 shows the screenshot of a sample mobile application running on Android, Part A is the data grid that is for the purpose of showing list of data records in the tabular format. NMRE uses services provided by DRE to query at the data source. To use DRE's services, NMRE has to send necessary parameters to the service according to the structure specified in the data

Notes: (a) Data grid; (b) data form

Figure 3.
Sample user interface

model. After the result is received, the data grid is dynamically rendered to show a table filled with result records. The gesture controls and buttons are by default presented for data-oriented functions such as insert, update and delete. When creating a new data record or editing an existing record, the data form is presented for the purpose of data entry through various kinds of input as depicted in Part B. After the user completes filling in the data form, NMRE formats information that the user inputs into JSON format and sends to DRE to insert or update data at the data source.

3.2 Matter of security

Figure 4 shows interaction between components when MAIMC is executed using the framework. The user logs on through mobile application by entering credential information for authentication. The authentication can be done by a service provided by SRE. If the credential information is authentic, SRE will generate *Authorization key* as a pair characters of alphanumeric. *Authorization key* is temporarily stored and mapped with a user name on SRE before it is sent to NMRE. NMRE puts *Authorization key* in the header part of every request to DRE services for the following reasons:

- DRE has to verify whether incoming request is trustworthy for further process; and
- DRE needs to be able to identify the user to determine the access privilege.

Authorization key is valid for a limited time (30 min by default) after last request has been received by DRE's services. If *Authorization key* has expired, the user needs to go through authentication process by logging on to the application.

3.3 Software evolution

The functional change is inevitable in the software system, as well as changes at the platform that the application is running on. Therefore, the framework aims to support

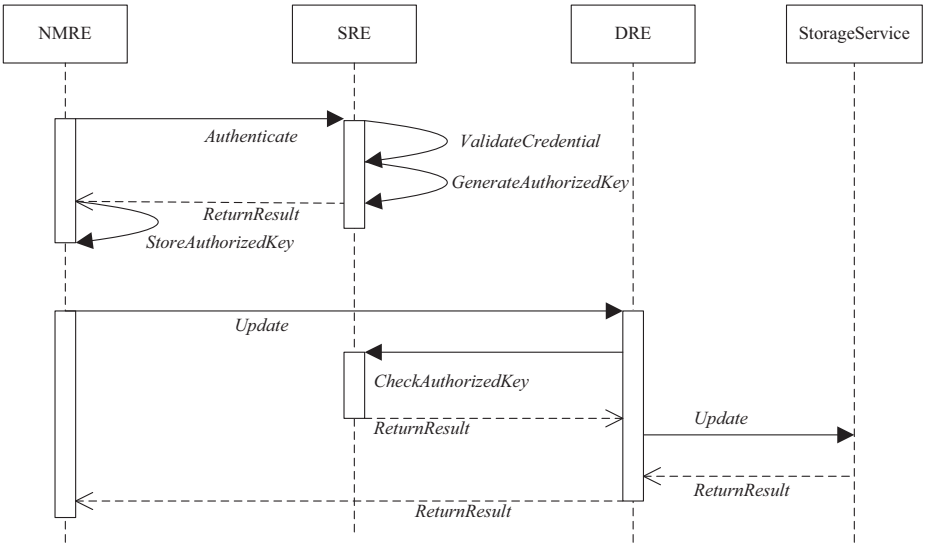

Figure 4.
Components interaction

quality attributes for software evolution, namely, function changeability and platform adaptability. The meta-model gives a scope of what change can be done on the function. To support platform adaptability, the framework provides programming libraries which are extendable to support changes at the platform, as well as migration to a complete new platform in the future.

3.3.1 Function changeability. MDE takes model as a development artifact throughout the software development process. Any changes to the functionality can be done through the model. The meta-model is therefore designed to support modeling of the application blueprint. The meta-model is platform independency model processed by SIMON to automate development and support execution of MAIMC. The application blueprint can be managed using a Web-based application referred to as *Blueprint Manager*. *Blueprint Manager* enables developers to make changes that the structure of meta-model supports such as data model, security policy and user interface. For example, new data entities and attributes can be added, or the security rule can be updated to allow an additional user role to access some data. When there are any changes made to the application blueprint, the application's functionality will also be changed, after the framework reloads the application blueprint. The application blueprint can be reloaded to make functional change without any downtime of the infrastructure.

The meta-model is depicted in Figure 5, it gives a scope of what functional change can be done on the application system. The meta-model is composed of three parts that are specification of three essential design aspects in the application blueprint. Each part has the structure of meta-elements which gives detailed description of how the application should be designed. Because the framework aims to support data-oriented functionality, so meta-data model is a core of meta-model. Meta-data model is a specification that covers essential structure of typical data modeling namely, data entity, data attributes and relationship. The meta-data model coexists and relates with a number of meta-elements that give specification of security policy and user interface configuration. The user interface configuration is designed to define characteristics of user interface

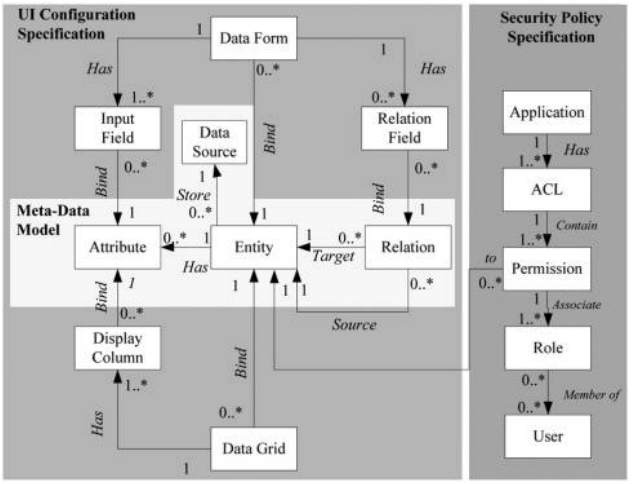

Figure 5.
Application blueprint specification

that support acquisition and presentation of information. The security policy specification is designed for rules that enforce confidentiality and integrity of the data.

XML format of sample application blueprint:

Part A: Data Model

```
<data-model>
  <entity name = "Employee" datasource="AWS">
    <attribute name = "firstname" datatype = "string" />
    <attribute name = "lastname" datatype = "string" />
    <attribute name = "birthday" datatype = "date"/>
  <attribute name = "photo" datatype = "file"/>
  <attribute name = "position" datatype = "string" />
      ....
  </entity<
  <entity name="Phone" datasource = "GAE">
    <attribute name = "label" datatype = "string" />
    <attribute name = "number" datatype = "string" />
  </entity>
  <relationship name = "Employee-Phone">
      <source>Employee</source>
      <target>Phone</target>
      <type>1-n</type>
  </relationship>
</data-model>
```

Part B: UI Configuration

```
<datagrid>
  <name>EmployeeGrid</name>
  <displayName>Employee</displayName>
  <type>generic</type>
  <bindEntity>Employee</bindEntity>
  <displayColumn>
    <bindAttribute>firstname</bindAttribute>
    <displayName>First Name<displayName>
    <width>100</width>
    <visible>true</visible>
    </displayColumn>
      ....
</datagrid>
<dataform>
    <name>EmployeeForm</name>
    <displayName>Employee Form</displayName>
    <type>generic</type>
    <bindEntity>Employee</bindEntity>
    <inputField>
        <bindAttribute>firstname</bindAttribute>
        <displayName>First name</displayName>
        <type>text</type>
          <hint>Required Field</hint>
        <required>true</required>
    </inputField>
      ....
</dataform>
```

```
Part C: Security Policy
  <organization>
    <roles>
      <role>manager</role>
      <role>office</role>
    </roles>
    </organization>
  <application name = "Human Resource Management" code = "HRM">
    <ACL>
        <permission>
          <role>manager</role>
          <entity>Employee</entity>
          <level>3</level>
        </permission>
        <permission>
          <role>officer</role>
          <entity>Employee</entity>
          <level>2</level>
        </permission>
    </ACL>
  </application>
```

The above XML format of an application blueprint can be exported by *Blueprint Manager*. The sample application is a part of human resource management system, and it has data-oriented functions that allow users to manage information of employee and phone.

The meta-data model is presented in the highlighted part of Figure 5. A record of *Entity* represents a data entity in the application which is one-to-many related to *Attribute* because a data entity can have a number of attributes. *Attribute* defines properties of data attributes such as attribute name and its data type. The framework currently supports the following data types: string, date, number and file. *Relation* supports information of the relationship made between data entities in the application; therefore, it has two relationships to *Entity* in the meta-data model, to the handle the source and target of a relationship. *DataSource* aims to keep configuration parameters that are necessary to access the data source. In an application, different entities can be designed to store on different data source located on different clouds or local RDBMS, so *Entity* has a link to *DataSource*. Part A shows the sample of a data model which defines two data entities, namely, *Employee* and *Phone*. The relationship is defined in <relationship>. The type of relationship is specified as "1-n" representing 1-to-many relationship; this expresses that an employee can have one or many phones.

Figure 5 shows the specification of user interface configuration which has meta-elements that are related to the meta-data model. Each *Entity* has one or more *DataForm* for the purpose of defining how data are entered. A *DataForm* is composed of a number of *InputField* which defines how the value of each attribute is entered. *Relationfield* defines how relationships can be made, either to select an existing record or create a new record to relate with. *DataGrid* defines how data are presented in the data grid. A data grid is composed of the number of *DisplayColumn*. Each *DisplayColumn* gives details of format showing data attributes in columns of the data grid. Part B shows a sample data grid configuration defined at <datagrid>. This data grid presents

records of an employee. And <dataform> defines how data are entered for each attribute of an employee.

The security policy is based on *Access Control List* (ACL) pattern (Messaoud, 2006), it contains ACL rules that define user roles and their privileges to access different data. An application has the number of *ACL*. Each of ACLs contains a set of *Permission*. *Permission* has a designated number specified to represent the level of data access (3 = Delete, 2 = Update, 1 = Read and 0 = None). *Permission* has *Entity* and Role defined to specify who can access what. A user can subscribe to one or more Role to get the authorization to access data. Part C shows a sample of security policy.

SIMON uses the application blueprint at runtime to help set up data structure at the data source, as well as providing necessary REST-based services to support functionality on the mobile application. The application blueprint is processed by *model loader* during the framework's initiation. The *model loader* can also be triggered to run anytime when there is any change made in the application blueprint, it fetches data model and sets up data structure on designated data storage through the cloud adaptor's Data Access Object (DAO) class. The DAO class has *createTable()*, a function that helps create data structure at the data source which can be either cloud-based storage service or local RDBMS. Below is pseudo code that explains the algorithm to create data structure:

```
1: Input: Data model
2: for all Entity do
3:    if table for entity is not exists then
4:       Select appropriate adaptor for specified data source
5:       Create a table for entity with entity_id as primary key
6:    end if
7: end for
8: for all Relationship do
9:    if table for relation is not exists then
10:          select appropriate adaptor for source's data source
11:          source_id = append(entity's name of source, "entity_id")
12:          target_id = append(entity's name of target, "entity_id")
13:          create a table for relationship with source_id and target_id as primary key
14:   end if
15:   If Relationship is bi-directional then
16:          select appropriate adaptor for target's data source
17:          source_id = append(entity's name of source, "entity_id")
18:          target_id = append(entity's name of target, "entity_id")
19:          create a table for relationship with source_id and target_id as primary key
20:   end if
21: end for
```

According to the meta-data model described in Section 3.3.1, the model loader processes each *Entity* and creates a corresponding table on designated data storage which is defined at *Entity*'s associated *Datasource*. *entity_id* is automatically inserted into every table for primary key. The cloud adaptors manage data according to the requirement given by cloud vendors. For primary key, *entity_id* is created as *hash key* on AWS's DynamoDB, while it is created as *key* on GAE's Datastore. After all entities are created, a table for each relationship is created according to details defined at *Relationship*. These relationship tables contain two properties, namely, source's entity_id and target's entity_id to record primary keys of associated entities. On AWS's DynamoDB, source's

primary key is *hash key*, and target's primary key is *range key* for enhancing performance when there is a query to find a relationship. On GAE's Datastore, primary key of source and target are created as *key* in the relationship table. If the relationship is bi-directional, an additional table will be created to support reverse relationship from the target back to the source.

To support different designs of data-model, we need to standardize the structure of data exchanging within the frameworks, so that the design changes do not require changes made at the part of the source code that supports data exchanging. Figure 6 presents the structure of data that is used at runtime to syntactically standardize data transported between NMRE and DRE through the cloud adaptor. To perform data operations on different clouds, the cloud adaptor's DAO class has standard functions for typical data operation such as *insert()*, *update()*, *remove()* and *relate()*, which help to insert a new record, update, delete and make a relationship, respectively. These functions process data encapsulated within RuntimeEntity class. The RuntimeEntity class has a structure that fits to any design of data model, it encapsulates the attribute's values and associated relationships. When *insert()* and *update()* are called, the functions extract the attribute's value from RuntimeEntity according to the attribute's name defined in the data model. After that, the attribute's value is later processed by the cloud adaptor through proprietary API corresponding to the data source. The cloud adaptor also has *query()* method which supports different queries to the data source. *query()* takes RuntimeEntityQuery as a parameter which represents the query expression. RuntimeEntityQuery consists of a number of RuntimeAttributeQuery. RuntimeAttributeQuery supports expression to find a matching data by its attribute's value, it defines data attribute's name, value to find and a comparison operator. RuntimeRelationQuery supports expression to find a matching relationship.

Because of the mechanism described above, the framework can dynamically adapt to changes in the model of application blueprint without any changes required to be made in the source code. As a result, the functional changes can be done at runtime without restarting the infrastructure that supports the application.

3.3.2 Platform adaptability. The development of MAIMC has a major problem known as *vendor lock-in* that is an obstacle of platform adaptability. The proposed framework therefore aims to prevent vendor lock-in problem. SIMON is designed with the mechanism that abstracts proprietary programming details of cloud and native mobile platform. The platform's programming details are isolated in the adaptor to prevent ripple effect. If there are any changes on the platform, the modification is only required

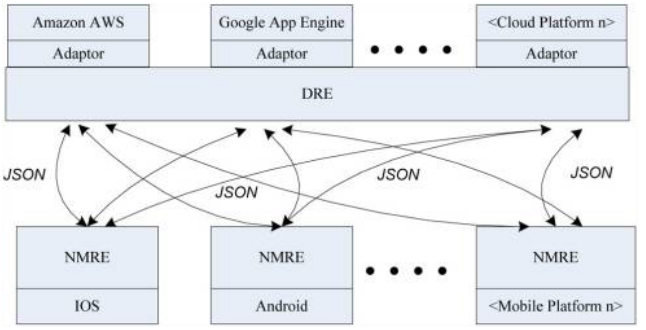

Figure 6.
Interoperability using SIMON

to be made on the adaptor without any impact propagating to the functionality. Moreover, the framework provides programming libraries which are extendable to support new mobile and cloud platforms.

3.4 Mobile platform

The mobile application runs by NMRE. NMRE is designed to be a middleware that supports running the mobile application on native mobile platform. It renders user interface and give functionality to the user according to the application blueprint. Therefore, if there are any changes required to be made on the functionality of the mobile application, the changes can be made at the application blueprint, and it is propagated to the mobile application without the need to make any change at the development artifacts of mobile application. The development of NMRE follows a standard specification, which is composed of three major subcomponents: *ApplicationInitiator*, *UIRenderer* and *ServiceCaller*. Each subcomponent has various methods to support the execution of data-oriented functions on the mobile application. During initiation, *ApplicationInitiator* is triggered to work to load necessary configuration from *Client Artifact* retrieved from BE. *UIRenderer* dynamically renders user interfaces according to the user interface configuration defined in the application blueprint. *ServiceCaller* supports interaction to services provided by core components. *ServiceCaller* prepares the requesting message including necessary parameters in JSON format; it then sets up a network connection to the service and receives back responding messages in JSON format which is later converted into the format that *UIRenderer* can use to present data to the user.

3.5 Cloud platform

The storage service on different clouds must be used through different proprietary APIs, so SIMON abstracts programming details of the storage service's APIs with the cloud adaptor. The cloud adaptor is developed as an intermediate to data storage which can be either public cloud-based service or local data storage on private cloud infrastructure. The cloud adaptor is attached to DRE; it contains any proprietary details that are required to access cloud-based storage service and local RDBMS. Therefore, any changes on the cloud platform only need modification on the cloud adaptor and do not affect the other parts. The cloud adaptor consists of modules which contain source code that uses proprietary data storage's APIs to access data sources. These modules are Java classes that are based on DAO pattern (Matic *et al.*, 2004), and they are inherited from *IDataAccessObject*. a Java interface that has the number of standard methods for the following two purposes. Meta-data administration helps setting up data structure on designated data storage according to the meta-data model, and typical data operation is for the purpose of performing basic data operation on data storage. At runtime, NMRE and DRE's services exchange data in JSON format which follows common structure described previously. For the sample implementation, two adaptors are developed particularly for two cloud-based storages, namely, AWS's *DynamoDB* and GAE's *Data store*. An adaptor is also developed for MySQL database to access data stored locally at private cloud infrastructure.

Figure 7 shows that SIMON supports interoperability in the heterogeneous environment of MAIMC. NMRE and cloud adaptor are shields against various platforms involved in the application system. As a result, mobile applications on

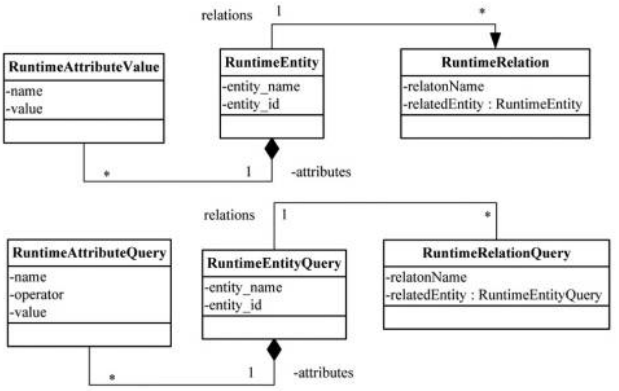

Figure 7.
Runtime class diagram

different operating system can seamlessly interoperate cloud-based services on multiple clouds with platform adaptability.

4. Implementation and evaluation

We compare SIMON with the conventional approach for the development of MAIMC. The conventional development raises vendor lock-in problem and consequently adds complexity to the development. Two scenarios are used to assess function changeability and platform adaptability. This evaluation concentrates on the measurement of effort taken for the development using *Source line of code (SLOC)*. SLOC is a software metric used to measure the size of a software program by counting the number of lines in the source code that the developer has to develop; it is one of the influential factors that determine the amount of development effort (Garcia *et al.*, 1996). The greater SLOC causes more inertia to make any changes in the system (Booch, 2008).

Safety engineering system is selected to be a sample application system used to examine with both approaches. The application system serves the safety engineering department at Electricity Generating Authority of Thailand (EGAT). The safety engineering department is responsible for managing and maintaining various kinds of safety appliances such as fire alarms, springer pipe systems and fire hydrants. The safety appliances are periodically inspected and tested, this requires the engineer team to record information. The information of testing and inspection are later used by the department to prepare several annual summary reports, which are required to be regularly submitted to the government agency for regulation review. The routine inspection and testing must be performed in parallel at multiple locations because numerous appliances are located separately in different rooms at different terminal buildings. Therefore, the application system needs to allow a group of engineers to simultaneously input information on the mobile application, and the information will be remotely stored on the centralized database which can be later queried.

The data model of this application system is shown in Figure 8. There are two types of data entities, namely, master data and transactional data. Master data are generally the core data that is essential to the operation of business such as department, building, room and equipment. These master data are required before making a transactional data which are information recorded for inspection and testing. The data model can be described as follows. *Department, Terminal, Room* and *Equipment* are for the purpose of keeping information of

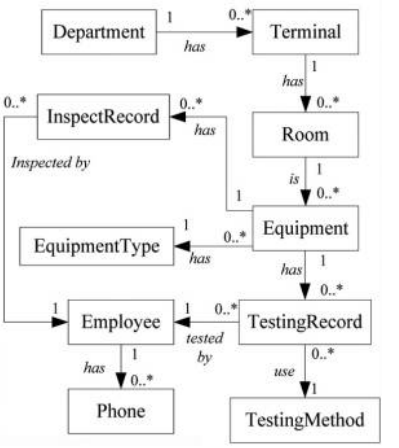

Figure 8.
Data model of safety engineering application system

appliances and their location. The appliances can be categorized into different kinds specified by *EquipmentType*. Because these appliances are periodically inspected and tested, so *InspectionRecord* and *TestingRecord* are used to keep information of inspection and testing. *TestingMethod* keeps method of testing. The information of inspection and testing can also include videos and images of the safety appliances. The system keeps the employee's profiles and contact information in *Employee* and *Phone*. The security policy defines two user roles, namely, *Engineer* and *Team support*. *Team support* is responsible for managing the master data. *Engineer* can browse through the master data to find equipment to record details of inspection and testing. For the master data, *Team support* has permission to write, while *Engineer* has permission to read only. The master data have to be stored on local RDBMS located in the private cloud infrastructure because they are the company's sensitive information that must be highly protected from unauthorized access. The records of inspection and testing are high volume because of the video and image files so they are stored on GAE's Datastore, a cloud-based storage service.

The application system initially includes a mobile application for Android and backend services that helps managing data on MySQL and GAE's Datastore. With SIMON, the development efforts are primarily taken for designing and creating the application blueprint using *Blueprint Manager* Therefore, SLOC is significantly lower than conventional approach (as shown in the first row of Table I) because the conventional approach requires effort to develop the mobile application and backend services from scratch. Two change scenarios are used to evaluate the amount of effort taken for making different changes as follows.

Table I.
Development complexity comparison

Development scenario	Conventional (SLOC)	SIMON (SLOC)
Development	6.224	280
Change Scenario 1	153	16
Change Scenario 2	6.324	4.014

4.1 Change Scenario 1 – functional change

This scenario demonstrates a change in the data model. The inspection requires the method to be specified in the same way that the testing has. Therefore, *InspectionMethod* entity is added to the data model and has a one-to-many relationship to *InspectionRecord*. To make this change, SIMON needs slight changing in the data model part of the application blueprint. The conventional approach requires changing in source code for both backend services and the mobile application. With SIMON, SLOC this change can significantly be reduced by 89 per cent compared to the conventional approach as the result shown in the second row of Table I.

4.2 Change Scenario 2 – platform change

This scenario demonstrates a platform adaptability which requires data that have to be migrated to Microsoft Azure as cloud-based data storage. And a new version of mobile application has to be developed for Windows Phone. With SIMON, a new version of NMRE has to be developed specifically for Windows Phone. The Windows Phone version of NMRE follows the specification described in Section 3.3.2. Similarly, a new adaptor is developed for *SQL Database* service on Microsoft Azure. With the conventional approach, the back-end service and mobile application have to be developed from the very beginning, the existing source code cannot be reused anywhere. With SIMON, SLOC considerably decreases by approximately 37 per cent compared with the conventional approach. The number of SLOC can be seen in the third row of Table I.

SIMON is systematically compared with several existing approaches by examining selected features. Each approach has its own contribution, so the evaluation does not aim to determine which one is the best, but rather how it fits to support the development of MAIMC and adaptability to support additional mobile and cloud platform. The evaluation's results are presented in Table II. PhoneGap and Appcelerator have been used to develop real-world applications, while DIMAG and SIMON has only been the research prototype. SIMON is the only framework that does not generate any code, this cut off effort taken for deployment task and consequently allows function changeability at runtime. All approaches provide native mobile application except PhoneGap. For the development of backend services, Appcelerator and SIMON support REST API as the service interface. DIMAG and SIMON propose a mechanism to extend their framework to support new mobile platforms. SIMON is the only approach that is extensible to support new mobile and cloud platforms.

Feature	DIMAG	PhoneGap	Appcelerator	SIMON
1. Production applications	–	✓	✓	–
2. Code generation	✓	✓	✓	–
3. Native mobile application	✓	–	✓	✓
4. Support REST API service development	–	–	✓	✓
5. Support mobile application development support	✓	✓	✓	✓
6. Extensible to support new cloud platform	–	–	–	✓
7. Extensible to support new mobile platform	✓	–	–	✓

Table II. Feature comparison to existing work

5. Conclusion

This paper presents SIMON, a software framework that supports the development of data-oriented MAIMC. SIMON executes mobile applications that reside in the heterogeneous system environment consisted of various data sources located at private cloud and public cloud. The framework comprises prefabricated components that support function changeability and platform adaptability. SIMON has been used to develop a security engineering system for EGAT and is evaluated with different change scenarios to demonstrate the support of software evolution. The result shows significant decreasing in the amount of efforts taken for making changes.

Although SIMON only supports data-oriented applications, there are other applications that require information sending in the workflow according to business processes and rules. Therefore, the future work of SIMON will be extended to cover the functionality that supports the workflow application.

Notes

1. http://aws.amazon.com
2. https://cloud.google.com/appengine/
3. www.phonegap.com/
4. www.appcelerator.com
5. www.kinvey.com
6. www.parse.com
7. www.firebase.com
8. www.dreamfactory.com
9. http://dasein-cloud.sourceforge.net
10. https://libcloud.apache.org
11. https://jclouds.apache.org

References

Bass, L., Clements, P. and Kazman, R. (2004), *Software Architecture in Practice*, Addison Wesley, Boston, MA.

Booch, G. (2008), "Measuring architectural complexity", *IEEE Software*, Vol. 25 No. 4, pp. 14-15.

Deursen, A.V., Visser, E. and Warmer, J. (2007), "Model-driven software evolution: a research Agenda", *Proceedings of International Workshop on Model-Driven Software Evolution (ECSMR'07), Amsterdam, The Netherlands*.

Erbes, J., Motahari Nezhad, H.R. and Graupner, S. (2012), "The future of enterprise IT in the cloud", *Journal of Computer*, Vol. 45 No. 5, pp. 66-72.

Flores, H. and Srirama, S.N. (2014), "Mobile cloud middleware", *Journal of Systems and Software*, Vol. 92 No. 1, pp. 82-94.

France, R. and Rumpe, B. (2007), "Model-driven development of complex software: a research roadmap", *Proceedings of Future of Software Engineering (FOSE), Minneapolis*.

Garcia, M.A., Ellis, N.C. and Simmons, D.B. (1996), "A knowledge base system used to estimate schedule, effort, staff, documentation and defects in a software development process", *Proceedings of Mexico-USA Collaboration in Intelligent Systems Technologies*, pp. 306-314.

Kemp, R., Palmer, N., Kielmann, T., Seinstra, F., Drost, N., Maassen, J. and Bal, H. (2009), "eyeDentify: multimedia cyber foraging from a smartphone", *Proceedings of 11th IEEE International Symposium on Multimedia*.

Khan, A.R., Othman, M., Madani, S.A. and Khan, S.U. (2014), "A survey of mobile cloud computing application models", *IEEE Communications Survey & Tutorials*, Vol. 16 No. 1, pp. 393-413.

Mahjourian, R. (2008), "An architectural style for data-driven systems", *Proceedings of 10th International Conference on Software Reuse*, pp. 14-25.

Matic, D., Butorac, D. and Kegalj, H. (2004), "Data access architecture in object oriented applications using design patterns", *Proceedings of Electrotechnical Conference (MELECON)*, pp. 595-598.

Messaoud, B. (2006), *Access Control Systems: Security, Identity Management and Trust Models*, Spring Science Business Media Inc, New York, NY.

Miravet, P., Marin, I., Ortin, F. and Rodriguez, J. (2013), "Framework for the declarative implementation of native mobile applications", *IET Journal of Software*, Vol. 1 No. 1, pp. 19-32.

Richardson, L. and Amundsen, M. (2013), *RESTful Web APIs*, O'Reilly Media, CA.

Sanaei, Z., Abolfazli, S., Gani, A. and Buyya, R. (2014), "Heterogeneity in mobile cloud computing: taxonomy and open challenges", *IEEE Communications Survey & Tutorials*, Vol. 16 No. 1, pp. 369-392.

Singh, I., Steans, B. and Johnson, M. (2002), *Designing Enterprise Applications with the J2EE Platform*, Pearson Education, New Jersey.

Szyperski, C. (2002), *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley Professional, Boston, MA.

Unhelkar, B. and Murugesan, S. (2010), "The enterprise mobile applications development framework", *IT Professional*, Vol. 12 No. 3, pp. 33-39.

Corresponding author
Nacha Chondamrongkul can be contacted at: nacha.cho@mfu.ac.th