Sonata: A Workflow Model for Interactive Mobile Cloud Applications

Verdi March, Yan Gu, and Bu Sung Lee

Cloud Intelligence Lab, Hewlett-Packard Laboratories Singapore
{verdi.march,chloe.yan.gu,francis.lee}@hp.com

Abstract. Workflow is a well-established approach to visually compose large and complex applications out of components. However, existing workflow models do not provide high-level abstractions of two recurring user-interaction patterns in mobile cloud applications, namely backtracking and interactive controls. In this paper, we propose *Sonata*, a workflow model that provides high-level abstractions for implicit and structured backtracking, and interactive controls. We prototype a workflow engine for Android devices and another for a RESTful cloud service platform, each of which orchestrates the execution of mobile components and cloud services, respectively. Choreography between the mobile orchestrator and cloud orchestrator is implemented on top of HTTP using REST-style invocations. An example application workflow incorporating all our proposed constructs is further elaborated.

Keywords: implicit backtracking, structured workflow, interactive control, orchestration, choreography.

1 Introduction

The convergence of mobile and cloud leads to the emergence of mobile cloud applications [6]. A mobile cloud application is composed of cloud services and tasks running on end-point devices (e.g., smartphones, tablets, laptops, and future smart devices). As the number of features in applications increase, the complexity of application development needs to be managed. At present, the industries largely adopt the workflow paradigm to simplify the design, configuration, customization, management, and maintenance of complex applications such as in SOA (service-oriented architecture) [9,8] and big data processing [3,1,5,4]. In this paradigm, an application is viewed as a composition of independent components with a pre-defined execution flow among the components. A component is a self-contained building blocks of an application which, depending on the domain, can range from fine-grained objects (e.g., Java beans and Microsoft COM) to loosely-coupled services (e.g., web or cloud services).

To simplify the application development, high-level abstractions should be provided for two interaction patterns which are commonly found in mobile cloud applications, namely *backtracking* and *interactive control*. Backtracking, supported in all generations of mobile devices, enables the user of an application to

A. Ghose et al. (Eds.): ICSOC 2012, LNCS 7759, pp. 59–71, 2013.

instruct the application to go from one component to another. Interactive control enables the user to directly instruct which component to execute. As an illustration, consider an application for uploading an image to a social-networking website. This application consists of a *gallery* component, a *site selector* component, and a number of *image uploader* components. When the application is invoked, the *gallery* component displays a list of images stored on the phone. The user interacts with the application by selecting an image. This selection triggers a transition to the *site selector* component, which displays the selected image and the list of available social-networking websites. At this point, the user can press the back button of her device to backtrack to the *gallery* component. Alternatively, the user selects the website to publish the selected image (i.e., the interactive control pattern), which further trigger the execution of the appropriate *image uploader* component.

To the best of our knowledge, existing workflow models do not provide high-level abstractions for the above mentioned interactive patterns. *Forward-only models* such as MapReduce [3], Hadoop [1], and Dryad [5] do not support backtracking. *Explicit models* such as BPMN [9] and SPL [4] require the patterns to be implemented explicitly using low-level constructs such as backward edges and gateway nodes. Although Sarasvati supports *arbitrary backtracking* whereby any component can backtrack to any ancestor [2], it may lead to an ambiguous execution state (see Section 4), let alone interactive controls.

This paper proposes *Sonata*, our proposed approach to model the backtracking and interactive control in mobile cloud applications. It is motivated by a simple premise whereby the probability for composition error can be decreased by reducing the number of explicit constructs. Thus, instead of backward edges and gateway nodes, Sonata indicates backtracking by assigning tags to components. We also propose a structure of mobile cloud applications that are free from arbitrary backtracking to guarantee that workflow execution does not enter an ambiguous state. Lastly, Sonata proposes two specific interactive controls, namely *chooser* and *iterator*. We demonstrate the feasibility of Sonata by composing a face recognition application using Android components and RESTful cloud services. Our prototype runtime platform consists of a workflow engine for Android (written in Python) to orchestrate mobile components, a server-side workflow engine (written in Java and MySQL) to orchestrate cloud services, and a REST-based scheme to choreograph these two engines.

The remainder of this paper is organized as follows. Firstly, we compare Sonata with existing workflow models (Section 2). We then describe the interaction patterns in mobile cloud applications(Section 3), followed by our proposed work (Section 4). Afterwards, we describe our prototype (Section 5) and an application use case (Section 6). Lastly, we conclude this paper (Section 7).

2 Related Work

In terms of backtracking, existing workflow models can be classified as forward-only models, explicit models, and arbitrary models.

By design, forward-only models such as MapReduce [3], Hadoop [1], and Dryad [5] do not support backtracking. Hence, they are not suitable for mobile cloud applications in which users may request the application to transition from the current GUI state to a preceding GUI state.

Explicit models such as BPMN [9] and SPL [4] require backtracking to be explicitly specified for every pair of nodes where backtracking may occur. Hence, design scalability is a significant issue since the number of backward edges and gateway nodes (see Section 3.1) increases with the number of components in a workflow. Because the semantic of workflow is left to workflow designers, it is possible to inadvertently create a syntactically correct workflow yet semantically incorrect (see Section 4 for the detail discussion).

Arbitrary models such as Sarasvati [2] do not guarantee that workflow execution remains unambiguous. Sarasvati is programming toolkit to develop Java-based workflow [2]. It does not come with a model to visually compose a new workflow. Sarasvati supports only unstructured backtracking whereby any node can backtrack to an arbitrary ancestor. Therefore, it is prone to the time-travel paradox which is discussed in detail in Section 3.2. Lastly, Sarasvati does not support interactive controls which are an integral part of mobile cloud applications.

Existing workflow models target batch executions, whereby the next component to execute in a conditional execution is automatically triggered based on messages or events generated. No human intervention is required. Hence, conditional executions are implemented using programming-like constructs such as if-else, loop, scoping and fork-and-join. Scripting is typically involved, e.g., to specify the conditional logic of if-else or loop. However, mobile cloud applications are interactive. As such, they include use cases where the next component to execute is solely determined by users, rather being automatically inferred. Hence, implementing the inherently high-level user interactions is unnecessarily complicated. For example, interactive iteration is modeled using a combination of backward edges, explicit marker of the scope of iteration, and filtering rules scripted by application designer in the design time, so that during runtime, applications can automatically decide when to an iteration completes. In contrast to these existing approach, Sonata simply represents an interactive control as a single component, thereby, simplifying the resulted workflow.

3 Design Objectives

Mobile cloud applications are interactive: they wait for input from users, and then perform action based on the input. We identify two common patterns of interactions between users and mobile devices, namely backtracking and interactive controls. A desirable property of high-level abstraction is to minimize the number of explicit constructs to implement the patterns, since this reduces the probability of composition errors. In the followings, we discuss this subject in details.

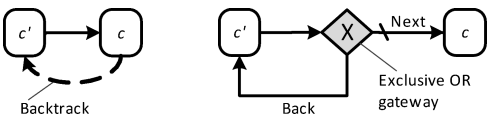
Fig. 1. Backtracking Interface in Mobile Devices

3.1 Backtracking

Backtracking is inherent in interactive mobile applications, as evident throughout device generations (Figure 1). Backtracking enables applications to go back from one GUI-enabled state to a previous GUI-enabled state. A high-level abstraction of backtracking should address the following issues.

Firstly, it should minimize the number of mandatory constructs required to represent a backtracking step. In explicit models, every backtracking from c to c' (Figure 2a) requires one explicit backward edges and a gateway node (Figure 2b). The gateway node represents the step whereby devices wait for user input. Hence, the number of backward edges and gateway nodes increase as the number of components where backtracking is possible grow.

Secondly, the high-level abstraction must enforce valid backtracking whereby backtracking occurs only between interactive nodes which wait for user input (Figure 3). A backtracking from c to c' can be triggered only if c waits for users to press the back button. Hence, c must be interactive which implies that c has a GUI. When c' is visited, the application displays the state pertaining to c' and waits for user input. This also implies that c' must also be interactive. Should c' is non-interactive, e.g., a cloud service or a logic-only mobile component, then users will perceive that the back button is ignored as c' will execute and immediately the application state returns to c. A low-level abstraction may produce workflows that are syntactically correct but semantically incorrect (i.e., the three invalid



(a) Logical (b) BPMN Implementation

Fig. 2. Backtracking and its Explicit Implementation

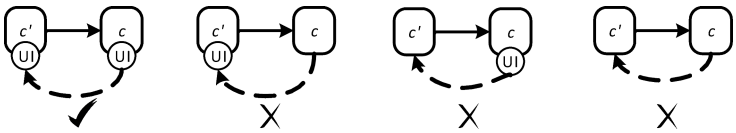


Fig. 3. Valid versus Invalid Backtracking (with GUI Symbols Added for Clarity

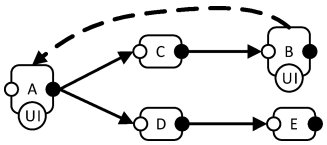


Fig. 4. Time Travel Paradox

backtracking illustrated in Figure 3. It is desirable that such workflows can be prevented by using a high-level abstraction.

Lastly, the high-level abstraction must prevent *time-travel paradox*. This paradox occurs when backtracking causes only a subset of a node's parents to be re-executed. Figure 4 shows a workflow which generates the time-travel paradox. Assume that a backtracking path is defined from B to A. The sequence of workflow execution that leads to the paradox is as follows:

1. Node A completes its execution.
2. Node C and D start their execution.
3. Suppose that before node D completes, node C already finished its execution and node B starts to execute.
4. Node B receives a backtracking command from user. Thus, we backtrack to node A. All the while, node D is still executing.
5. Node A is re-executed. After its completion, node C and D should be re-executed. However, recall that node D is still executing (i.e., from the previous iteration/wave). At this point, time-travel paradox occurs on node D.

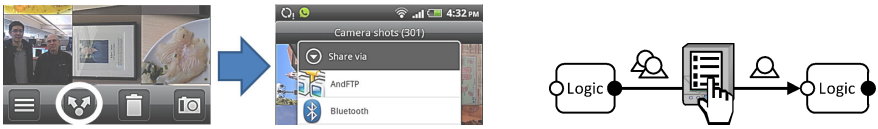
To solve the paradox, either:

1. node D cancels its execution from the previous iteration, or
2. after node D completes its execution from previous iteration, node E is executed. Then, node D is re-executed again (i.e., the current/new iteration), which implies that later node E will also be re-executed.

However, the appropriate solution depends on the context of the applications, and explicit notations increases the workflow complexity. Hence, a simpler solution based on preventive strategy is required(Section 4).

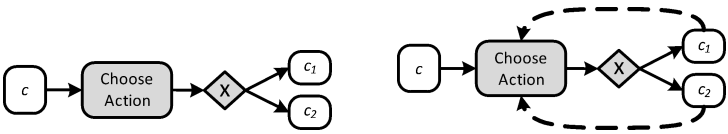
3.2 Interactive Controls

Interactive control enables application users to control the execution path of applications. Two types of interactive controls are identified in mobile applications (Figure 5). Firstly, when a workflow forks into multiple disjointed execution paths, users may want to select only one particular path. This is illustrated by the example in (Figure 5a). To other interactive control is iteration (Figure 5b). Iteration enables a list of data items of the same type to be consumed by a node designed to process only one item at a time. When a producer outputs outputs a list of data items, a mobile device will request its users to select a particular data



(a) Chooser (b) Iteration

Fig. 5. Interactive Control in Mobile Applications



(a) Chooser (b) Iterator

Fig. 6. Modeling Interactive Controls with BPMN

item from the list. Then, the selected data item is forwarded to the consumer which is designed to accept only one data item per invocation.

The motivation for a high-level abstraction is similar as in the case of backtracking. Firstly, we aim to minimizes the number of edges and gateway nodes (Figure 6). Secondly, we aim to prevent workflows that are syntactically correct but semantically incorrect. For examples, the exlusive OR gateway may be inadvertently replaced with a parallel fork/join, or when the backward edges are removed from an iterator such that it degenerates to a chooser.

4 Proposed Workflow Model

We propose Sonata, a workflow model for interactive mobile cloud applications which are composed of cloud services and tasks running on mobile devices. A workflow is modeled as a directed graph where each node represents a cloud service or a task on a mobile device, and each directed edge represents control and data flow between nodes. The key novelties of Sonata are: (i) implicit backtracking based on node types, (ii) structured workflow to prevent the time-travel paradox, and (iii) interactive control nodes.

Sonata infers backability from the types of nodes (Figure 7); thus, obviating the need for backward directed edges and additional gateway nodes (see Figure 2). Nodes are classified along two dimensions: (i) *interactive* versus *non-interactive*, and (ii) *backtrackable* versus *non-backtrackable*. Interactive components provides a GUI to allow user interaction during execution. On the contrary, non-interactive components do not allow user interaction during their execution. Only mobile components are interactive because users directly interact with mobile devices (i.e., the client-side). On the other hand, cloud services, by definition, are server-side components accessed programmatically by mobile components. Only backtrackable nodes can be re-visited by its interactive successor. Backtrackable are further sub-classed into *backable* and *bookmarked*. A backable node

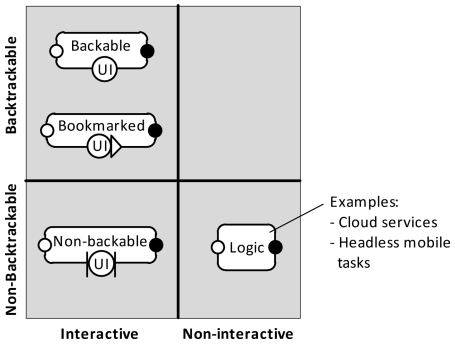


Fig. 7. Taxonomy and visual notation of nodes

can be re-visited only by its *immediate* interactive successor. Backable is the default type for interactive nodes. However, this behavior can be overriden by tagging an interactive node as a bookmarked node so that it can be re-visited by any of its interactive descendants. The bookmark sub-type is intended for larger-screen devices (e.g., tablets, notebooks and desktops) as additional navigation index can be displayed without obscuring the GUI content of a component. With our proposed approach, mobile cloud application workflows remain succinct and elegant, and are clean from redundant backward edges. Notice that in using our classification, the location of a component (i.e., mobile or cloud) is optional so that we do not need to explicitly mark whether a component is mobile or cloud.

To address the time-travel paradox, Sonata adopts a prevention strategy to prevent such an ambiguity to occur. The time-travel paradox occurs due to the structure of a directed graph is arbitrary. Hence, to prevent the paradox, Sonata enforces a *structure* whereby a Sonata workflow is structured as a critical path consisting of a sequence of interactive nodes and region of logic nodes (Figure 8). Each interactive node has at most one interactive successor (see the left-side workflow in Figure 8). Logic nodes between consecutive interactive nodes are grouped in a region, and synchronization barriers are imposed at the entrance and exit of each region (see the right-hand-side workflow in Figure 8). Sonata structure is reasonable for mobile cloud applications. The critical path denotes that interactive nodes must be executed in sequence. This makes sense in mobile devices since concurrent interactive nodes will compete for the device screen to display their GUI, and yet device screen is relatively limited for sophisticated UI mash-up.

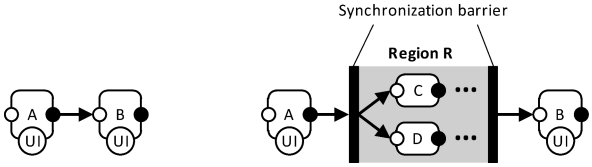


Fig. 8. Structure of Sonata Workflow

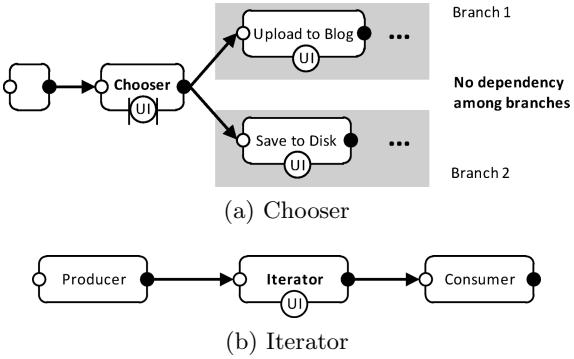


Fig. 9. Interactive Control Nodes

Sonata defines two interactive control structures, namely chooser (Figure 9a) and iterator (Figure 9b). Both accept user inputs from GUI during runtime, thereby, completely obviate the need for scripting and scoping in design time. Chooser is non-backable and equivalent to the XOR boolean operator. Chooser restricts users to select exactly one and only one execution path out of many during runtime to preserve the critical-path structure. Iterator enables a collection of data items of the same type to be processed by a node designed to process only one item at a time. As illustrated in Figure 9b, when face detection outputs a list of detected faces, the iterator will request users to select a particular face from the list. Then, the selected face is forwarded to face recognition which is designed to recognize one face at a time.

5 Preliminary Prototype

A workflow graph, implemented as a JSON object, is partitioned into a mobile partition and a cloud partition which contains Android-based mobile components and Java-based cloud components, respectively. Barriers and interactive controls within a workflow graph are implemented as built-in components (Table 1). The mobile partition and cloud partition is then executed by a mobile orchestrator and a cloud orchestrator, respectively (Figure 10). Choreography between these engines are implemented using RESTful invocations over HTTP. Within an orchestrator domain, component execution follows the master-and-slave paradigm whereby the orchestrator invokes one component after another,

Table 1. Built-in Components

Built-in Component	Type	Location
Barrier	Logic	Mobile
Chooser	Non-backable	Mobile
Iterator	Backable	Mobile

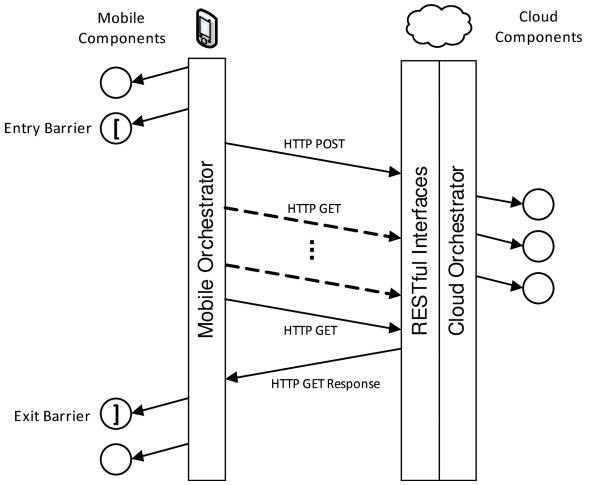


Fig. 10. Choreography between Mobile Orchestrator and Cloud Orchestrator

and data from one component must go through the orchestrator to the next component. The cloud orchestrator is implemented in Java and supports the execution of Java components. Intra-cloud invocations and data forwarding are implemented as native Java method invocations. The mobile orchestrator is implemented in Python using the SL4A[1] (Scripting Languages For Android) environment. Intra-device invocations and data forwarding are implemented via the Android's intent mechanism.

Each component specifies zero or more input and output parameters using `get(key)` and `set(key, value)` operation, respectively. The skeleton of a component is shown in Figure 11. The specific implementations of cloud components and mobile components are as follows:

– *Cloud Components* — We have developed a Java-based SDK to ease the development of cloud components. The SDK provides a framework for component life cycle and input/output APIs, and guarantees that components are re-entrant. A new cloud component is implemented by sub-classing the

```
Component c
    String i1 = get("i1");
    String i2 = get("i2");
    String o1 = do_something(i1, i2);
    put("o1");
    put("o2");
```

Fig. 11. Skeleton of Component with Input $\{i_1, i_2\}$ and Output $\{o_1, o_2\}$

[1] `http://code.google.com/p/android-scripting`

Table 2. RESTful Interfaces of Cloud Component c with Workflow w

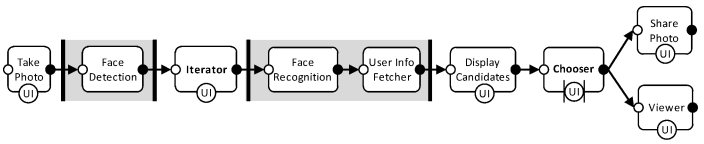
Method	URL	Description
POST	`/w`	Request a new session. Mandatory prior to executing w. Return handle `s`.
POST	`/w/s`	Start or resume the execution of cloud-portion of w.
POST	`/w/s/c`	Execute workflow w, starting from component c.
GET	`/w/s/c/k`	Retrieve output `k` emitted by component c. If the output is a blob object, returns a URL to the object to facilitate subsequent downloads.

`CloudComponent` class, and then overriding its `activate()` method with the specific functionality to be provided. Presently, keys are *strings*, whereas values can be *strings* or *blobs*. String values are transient and thus, they can be garbage collected when no longer referred to by any component. On the other hand, blob values are persisted as a file stored in a cloud storage. Components are transparently decorated with RESTful interface to ease their development (Table 2).

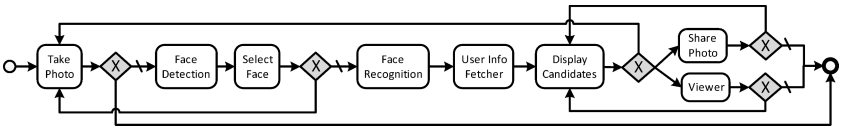
– *Mobile Components* — A mobile component is implemented as a native Android application (i.e., an `apk` package). Each mobile component is a subclass of `Activity`. Component input and output is implemented using the *Intents* mechanism[2]. In particular, the *get* and *set* operations are achieved using the intent's *extra* APIs in the Android application framework.

To handle state transitions from a mobile device to cloud and vice versa, we adopt the choreography approach whereby interactions between the orchestrator follow a peer-to-peer model (Figure 10), rather than being governed by a centralized entity. The transition from a mobile orchestrator to a cloud orchestrator is encapsulated in a multi-part/form-data HTTP POST request [7] `/w/s` or `/w/s/c` shown in Table 2. On the other hand, the transition from a cloud orchestrator to a mobile orchestrator is currently implemented using a pull mechanism. In this scheme, the mobile orchestrator periodically polls whether the cloud orchestrator has completed its execution. Once the cloud execution completes, the mobile orchestrator pulls the necessary data by sending an HTTP GET request `/w/s/c/k` (see Table 2) to the cloud orchestrator. In response, the cloud orchestrator serializes the data into JSON objects, then serves the serialized data to the mobile orchestrator. The mobile orchestrator then de-serializes the JSON objects before injecting the data to the appropriate mobile components. Polling the cloud orchestrator may be simpler to implement, but may increase the number of round trips. Thus, at the moment, a push-based mechanism is being considered. However, the issue of polling-vs-not is just a non-functional aspect of our implementation which does not affect the functionality the Sonata model.

[2] `http://developer.android.com/guide/topics/intents/intents-filters.html`

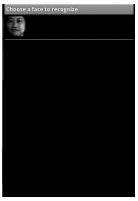


(a) Sonata Workflow



(b) BMPN Workflow

(c) Camera



(d) Select Detected Face

(e) Display Candidates

(f) Actions for Match

(g) View Match

(h) Share Match

Fig. 12. Face Recognition Application

6 Use Case

Face Recognition enables a user to automatically retrieve the profile of a face taken with an Android device. As can be seen in Figure 12a–12b, the Sonata workflow is simpler than the BPMN one. The *Photo Snapper* snaps a photo by engaging the embedded camera on the mobile phone (Figure 12c). The captured photo will be sent to and processed by the *Face Detection* cloud service. This cloud service detects faces in a photo based on the face detection algorithm proposed by Tong et. al. [10]. The list of detected face images is forwarded to the mobile device, and then displayed by the *Iterator* component Figure 12d. Once a user selects the face image, the face image is forwarded to the *Face Recognition* component. The *Face Recognition* component compares the face image to available images in a database based on the face clustering algorithms in

Tong et al. [10]. It returns three images with the highest match scores, and their corresponding profile (i.e., name, designation, and email) are further retrieved by *User Info Fetcher*. The face image and profile of the three candidates (i.e., image faces and their profile) is forwarded to the mobile device to be displayed by the *Display* component (Figure 12e). Once a user select a candidate, the *Chooser* component displays two possible actions corresponding to the two successors of *Chooser*, namely *Share* and *Viewer* (Figure 12f. If the user chooses to view, then the *Viewer* component displays the selected candidate image in full-screen mode Figure 12g. Alternatively, if the user chooses to share, then the *Share* component will be invoked to (Figure 12h).

7 Conclusion

Our proposed workflow model is based on the simple premise whereby the probability for composition errors can be decreased by reducing the number of explicit constructs. Our workflow model supports implicit backtracking and interactive controls, which are two recurrent interactive patterns in mobile devices. Our workflow model imposes structured backtracking to prevent workflows that are syntactically correct but semantically incorrect. Interactive controls (i.e., chooser and iterator) are targeted at use cases where the next component to execute is solely determine by users, rather being automatically triggered from messages or events generated by a workflow engine. Therefore, during the design time, the application designer does not need to write the filtering rules with programmatic expressions. When Sonata is implemented into a workflow designer GUI, then mobile cloud application can be visually assembled in a drag-and-drop fashion. Our ongoing works include investigating more high-level constructs into Sonata, including abstractions for life-cycle management and streaming (i.e., tightly-coupled) operations between components. In addition, we are working on quantitative analysis of design scalability and workflow's runtime performance.

References

1. Apache Hadoop, `http://hadoop.apache.org`
2. Sarasvati, `http://code.google.com/p/sarasvati/`
3. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: Proc. of OSDI (December 2004)
4. Hirzel, M., Andrade, H., Gedik, B., Kumar, V., Losa, G., Nasgaard, M.M.H., Soule, R., Wu, K.-L.: SPL stream processing language specification. Technical Report RC24897 (W0907-066), IBM Research Division (November 2009)
5. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed data-parallel programs from sequential building blocks. In: Proc. of EuroSys (March 2007)
6. March, V., Gu, Y., Leonardi, E., Goh, G., Kirchberg, M., Lee, B.S.: μCloud: Towards a new paradigm of rich mobile applications. In: Proc. of MobiWIS (September 2011)

7. Nebel, E., Masinter, L.: RFC1867: Form-based file upload in HTML, http://www.ietf.org/rfc/rfc1867.txt
8. OASIS Standard Committee. Web Services Business Process Execution Language version 2.0 (2007), http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html
9. Object Management Group. Business Process Model and Notation (BPMN) version 2.0 (January 2011), http://www.omg.org/spec/BPMN/2.0/PDF
10. Zhang, T., Xiao, J., Wen, D., Ding, X.: Face based image navigation and search. In: Proc. of the 17th ACM Intl. Conf. on Multimedia (MM), pp. 597–600 (March 2009)