A Data Mashup Engine for Smartphones Based on XProc

Harald Hofstätter
Vienna University of Technology
Vienna, Austria
e0927651@student.tuwien.ac.at

Amin Anjomshoaa
Vienna University of Technology
Vienna, Austria
anjomshoaa@ifs.tuwien.ac.at

A Min Tjoa
Vienna University of Technology
Vienna, Austria
amin@ifs.tuwien.ac.at

ABSTRACT
One of smartphones' core factor of success is the vast amount of available mobile applications. Despite this numerous number of apps there cannot exist one for every single use case. If a user does not find an app fulfilling the requirements, he rarely is able to write one of its own because of (1) missing programming skills, (2) too much time effort for simple tasks and (3) expenses on particular systems. This work proposes a mobile workflow engine on Android to empower users without much technical knowledge to build their solution in a platform-neutral and standardized language called XProc which is the W3C recommendation for data processing via XML technologies. For this purpose one of the XProc implementations is ported to Android and extended with smartphone specific features and services such as location, short messages, mobile UI, etc. Also a trigger mechanism has been implemented to activate workflows based on the user's context like time, location and reception of a short messages. A performance measurement done in this work demonstrates no execution drawback compared to native development after a workflow is compiled to its object representation.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: Domain-specific architectures, Languages, Patterns, D.2.13 [**Reusable Software**]: Reusable libraries

General Terms
Algorithms, Management, Design, Languages

Keywords
Data Mashup, Workflow, Smartphone Programming

1. INTRODUCTION
The term Ubiquitous Computing which was coined out in 1991 [1] envisions a large variety of devices gain more intelligence through built-in computers. They are interconnected and aware of their context like location or the other devices in their surroundings. Those enhanced devices should support the human's life while staying in background so that the user may not even be aware that he is using a computer.

In this context ubiquitous computing devices are defined as "cheap, low-power computers that include equally convenient displays, a network that ties them all together, and software systems implementing ubiquitous applications". Today's smartphones and tablets are definitely a step into this direction and combine all these requirements. They feature sensors like accelerometer, gyroscope, magnetic field sensor, etc. and are equipped with common input and output devices such as camera, microphone and some audio output.

Smartphones would not be that successful without the vast amounts of mobile applications (apps) for them. There is an ever increasing number of apps for different mobile operating systems in different app markets. An app can be developed in many different ways having varying abstraction levels to the underlying operating systems. They can be built by using some drag and drop IDEs (e.g. Red Foundry's Fusion Studio [7]), web frameworks (e.g. PhoneGap, jQuery Mobile, Titanium, Rhodes) or by doing some native implementation in the respective programming language. The fewer the abstraction level is, the more possibilities exist to build apps. For example building a mobile frontend for some REST service will be easily possible with web technologies, namely HTML and Javascript. On the other hand, implementing an image manipulation application will only be possible with native development.

Despite this huge number of apps, a lot of use cases are not tackled by today's apps and the mobile users are striving for custom-tailored tools for addressing the long tail of requirements [2]. If the user's required app does not exist, the user should either wait until the required app is implemented by community or develop the app himself which does not seem to be feasible for the average smartphone users.

A sample use case for personal requirements can be formulated as follows: if I am at the railway station in Vienna on Friday, buy an SMS-ticket from Vienna to Salzburg. Building an application for this would be quite trivial, but can only be accomplished by the fewest smartphone users. Apart from that, who would spend some hours developing time to save a minute every Friday? Even doing so, adjusting the app so that it is useful for others can become complicated very soon.

One method for creating such situational applications is to use mashups which are defined as "user-driven micro-integration of web accessible data" and they owe their popularity and fast improvements to two fundamental elements: Web 2.0 and SOA [3][4]. The objective of mashups is to build effective and lightweight information-processing solutions based on the accessible Web services of organizations. Mashups can be applied

MoMM2013, 2-4 December, 2013, Vienna, Austria.

to a broad spectrum of use cases ranging from simple data widgets to more complex ones such as task automation and system integration [5]. The lightweight resource composition of mashups makes them appropriate for many domains such as business and enterprise applications, where different data sources can be combined together in a "quick and dirty" manner. The main components of the mashup stack solutions are APIs and widgets.

To resolve the problem stated in this section, we have adopted a workflow engine and ported it successfully to Android platform. This engine is able to execute user-generated workflows which are described in W3C's XProc language.

2. STATE OF THE ART

The communication facilities of smartphones also can enable them to cooperate to create mobile-based mashups for combining data from different devices. Prutsachainimmit et al [6] tackle this challenge in their research and created a platform to gather information from several smartphones to feed a web API for finding the requested location (e.g. a restaurant) in a collaborative way. Their approach enables to build mobile mashups that use the shared information among different devices. Therefore several small service applications have to run on the smartphones which handle communication. In this research, mashups are described in a flow-based XML language called Cooperation Mobile Application Interface Description Language (C-MAIDL).

In this context, there are also a number of mobile platforms that enable users to create mobile apps without dealing with programming challenges. One such solution is the Red Foundry [7] that enables users to create native mobile applications without programming skills. Apps are assembled by drag and drop in a Rich Internet Application (RIA) from a lot of different components which can be grouped into layout, input, output, etc. There also exist a handful of implementations for calling website APIs of Facebook, Twitter, Flickr or fetching RSS feeds. The app is saved in a proprietary XML format called Red Foundry Markup Language (RFML), which also allows building reusable components for other users. The created apps can be either previewed through an online service or compiled to native Android and iOS applications.

Another mashup solution for mobiles is Microsoft's on{x} which is a service that enables users to program their devices to react on environmental changes. Android was chosen as the first target platform, others should follow in the future [8]. The key component in on{x} is a rule. It is possible to load several rules on the smartphone which are executed in the background. Running in background is a vital, because the smartphone should react to its environment without user interaction. A rule is composed of a trigger and an action. If the trigger is satisfied, the action is executed. Rules are programmed in Javascript, hence programming skills are needed to write new rules.

Tasker [9] is a very powerful Android application from Crafty Apps which automates tasks based on contexts. Task automations are called profiles and are created on the smartphone directly. Everything is done by clicking and drag & drop, no programming skills are needed. A profile consists of one or several contexts, an enter-task and an optional exit-task.

Locale [10] is also an Android application and pretty similar to Tasker though it is much simpler. In Locale situations are configured which trigger system settings. Situations can be a certain time, location, incoming call, phone orientation, and so on. The app can only set some settings like background image, ringtone volume, switch Wi-Fi on or off, or launch an application. It is not possible to load data or display a message. But Locale has also implemented a plugin architecture, hence developers can extend it with custom situations and actions. A lot of plugins exist already for different tasks such as reacting upon receiving a text message or when wired headphones are connected.

JBoss implemented a light-weight and open-source Business Process Management Suite for Java called jBPM [11]. The heart of the project is the Core Process Engine supporting Business Process Model and Notation (BPMN) 2.0 for modelling and executing business processes. The jBPM can be extended by domain specific nodes to simplify development. This mechanism not only allows to wire any custom nodes into business processes, but also the jBPM Designer (a graphical tool to design processes) supports these nodes. Verlaenen [12] successfully ported jBPM to Android.

Table 1, summarizes the existing mashup solutions for smartphones and compares them based on different indicators such as coding requirement, extendibility, data processing support, context support, etc.

Feature	Red Foundry	on{x}	Tasker	Locale	jBPM
No coding required	✓		✓	✓	✓
Open Source				✓	✓
Extendible	✓		✓	✓	✓
Supports data processing	✓	✓			✓
Context support		✓	✓	✓	
Cross-platform	✓	in Future			

Table 1. Summary of Mashup Solutions for Smartphones.

It is important to note that none of the aforementioned projects is capable of offering a rapid process development without coding skills in an open and extendible approach for different platforms. In our proposed approach we try to address these issues by providing a generic and extendable mashup solution for smartphones.

3. PROPOSED SOLUTION

In order to realize the mashup goals, we have used XProc [13] as the standard language for capturing and describing the data processing and data flow steps of the target situational solutions. XProc is based on the XML pipelines that are formed by connecting different XML processes so that the output of one serves as input for its successor. According to XProc specifications, "An XML Pipeline specifies a sequence of operations to be performed on a collection of XML input documents. Pipelines take zero or more XML documents as their input and produce zero or more XML documents as their output."

Typically several processing steps like XML transformation and validation are needed to transform one or several input documents into one or more output documents. Steps are connected by ports and one step can have zero or more uniquely named input and output ports. Each step has an implicit output port for reporting errors which must not be declared explicitly. Loops are not allowed, neither direct nor indirect ones.

XProc is developed by the W3C's XML Processing Model Working Group and since May 2010 it is a W3C Recommendation. XProc is a powerful tool providing different processing steps and control structures. Its native support loading XML documents over HTTP makes it a potential technology for building mashups based on the data pipe paradigm.

Another reason for choosing Xproc is that it follows the dataflow-paradigm, hence an XProc Pipeline can be illustrated as graph which is more comprehensible for non-technical users.

In order to execute and process the XProc instructions a number of execution engines are available. One of these engines is Calabash [14] which is an open source implementation by Norman Walsh who is also part of the XProc specification team. It is implemented in Java, uses XPath 2.0 as its expression language and can be extended easily by implementing custom steps. Because of these features, Calabash was chosen as XProc execution engine for this work and we have adopted it for running on Android devices. Tis adoption process required a number of changes to the core classes and also required libraries of Calabash.

Furthermore the standard XProc steps are extended to include platform specific features such as location, SMS, etc. Following the XProc definition, all these platform specific steps are atomic steps, i.e. they do not contain any sub-pipelines. The platform specific steps that have been implemented can be used inside the mashups as data providers or data consumers. These extended steps are as follows:

- **Toast**: is used to display any message to the user for a few seconds.
- **Notification**: creates a notification window for showing the relevant information of the workflow.
- **SMS**: sends a text message to the given number
- **Location**: provides the current location.
- **Orientation sensor**: provides the current values of the orientation sensor.
- **Intent**: starts an Android Activity via calling a specific Intent
- **UI**: dynamically creates a user interface for showing information or requesting user input.
- **Execution time**: provides the execution time of the current pipe.

To clarify the platform specific steps we will explore two of them in more details. Listing 1, shows the SMS step and the input and output ports. Similar to smashup widgets, this components can receive a number of parameters such as phone number and a message as input and return the results of sending the SMS as output.

```
<p:declare-step type="hx:sms">
  <p:option name="phone-number" required="true" />
  <p:option name="message" required="true" />
  <p:input port="source" sequence="true" />
  <p:output port="result" sequence="true" />
</p:declare-step>
```

Listing 1. The implemented SMS step

Another interesting step is calling an Android Intent which facilitates the interaction with all available Android services and third party apps with public intents. Intents provide the ability to start Android activities and the implemented Intent step can be used also to launch third party applications. It takes the intent's action and type as options and any number of parameters on its extras port. This step is non-blocking, meaning that the workflow continues execution immediately after the Intent is started. From the workflow's perception, starting the Activity can be seen as a *fire and forget* call. In some future work it would be possible to also support blocking calls to use the Activity's return value in subsequent steps. Listing 2, shows the structure of Intent step which can be combined with other built-in XProc steps to accomplish user tasks.

```
<p:declare-step type="hx:intent">
  <p:option name="action" required="true" />
  <p:option name="type" required="false" />
  <p:input port="extras" kind="parameter"/>
  <p:input port="source" sequence="true" />
  <p:output port="result" sequence="true" />
</p:declare-step>
```

Listing 2. The implemented Intent step

In addition to XProc steps, a number of triggers have been also implemented to activate workflows in a proactive manner. A workflow can have at most one trigger which invokes the workflow automatically if it is satisfied. A trigger is composed of at least one condition, whereas satisfaction of all conditions entail the trigger's validity and hence lead to the execution of the associated workflow. Therefore a trigger can be seen as a container for concrete condition implementations. Four different simple conditions have been implemented in this work:

- **Location condition** which is satisfied if a certain location is entered, defined by its latitude, longitude and radius in meters for these coordinates.
- **Time condition** which evaluates a specific time of day. At midnight the condition is set to unsatisfied waiting for the same moment on the day to get satisfied again.
- **Screen unlock condition** which becomes and stays satisfied as soon as the screen gets unlocked by the user.
- **Short message received condition** which becomes and stays satisfied as soon as a short message is received.

A trigger with its containing conditions is stored as a simple XML document and not only invokes its pipe automatically, but also injects the data which leads to its satisfaction.

Listing 3, shows a sample trigger which will be fired when the user enters the 100m neighborhood of a certain locations after 5pm.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<trigger>
  <time value="17:00" />
  <location latitude="48.347568" longitude="15.647808" radius="100" />
</trigger>
```

Listing 3. A sample trigger

In addition to mobile implementation, we have also implemented a synchronization mechanism to load the workflow definitions from a central workflow repository (CWR) on the web. In this way, the users will be able to share their mashups and also load the created solutions of other users. Workflow definitions are synchronized with the central workflow repository over a provided REST interface which is accessed by the CWR REST client component. Retrieved workflow definitions are stored in a local SQLite database on the mobile client.

Trigger and data pipelines are defined as XML documents, hence they can be easily stored as SQLite text fields without the need of any serialization. The local workflow repository manages synchronization and persistence of personal workflows.

Figure 1, depicts a snapshot of the Mashup platform and a number of installed workflows. It also shows that the end user may activate or deactivate workflows and directly run the workflows that are not depending on a trigger to start.

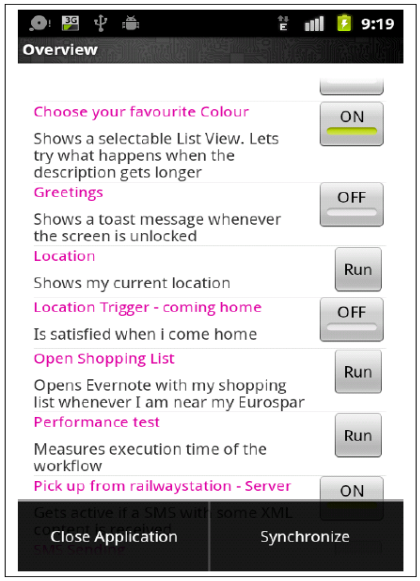

Figure 1. A snapshot of mashup platform on Android

4. PERFORMANCE EVALUATION

Describing workflows in a platform neutral way introduces an abstraction layer above native programming. This adds additional execution overhead in contrast to native development. To execute a workflow, the XML definition has to be transformed into an XPipeline object. Furthermore an XProcRuntime object is needed to execute the XPipeline. These two objects are called runtime objects. Creating them is computationally expensive, especially on smartphones having only low CPU power. The objects can be reused for further executions of the same workflow, hence they have to be created only once. At the current implementation runtime objects are created just before the first execution request and stored afterwards. The time between the first execution request and the beginning of the first step's execution, including creation of runtime objects, can be seen as warm-up time for the workflow. The actual execution of a workflow (that is executing the individual steps) runs with native speed because it uses pure Java and native Android classes. Hence the most time consuming action is the transformation process. This section evaluates the warm-up time on different Android smartphones and proposes some improvements.

4.1 Set-up

To measure the warm-up time, a simple workflow with only two steps was created and then the execution time on different devices was measured. Table 2, shows the test devices and their configuration.

Device	Android Version	Processor	RAM
ZTE Blade	2.3.7	600 MHz Qualcomm	512 MB
Samsung Galaxy Ace	2.3.6	800 MHz Qualcomm Snapdragon	278 MB
Sony Xperia U	2.3.7	1 GHz STE-U8500-Dual-Core	512 MB
Sony Xperia S	4.0.4	1.5 GHz Qualcomm-Dual-Core	1 GB
Samsung Galaxy Tab 10.1	4.0.4	1GHz 2 Dual-Core Nvidia Tegra 2	1 GB

Table 2. The target test devices and their configuration

4.2 Execution

The workflow measures the time between the execution request and the beginning of the first step's execution. Consequently it indicates how long a user has to wait until the desired workflow is really executed. Caching was disabled for all performance measurement in order to include the required costs for creating the runtime objects.

The workflow was cloned to be able to execute several instances of it and they have been started in parallel to measure how concurrent workflow executions influence each other. To start workflows (nearly) at the same time, a screen unlock trigger was used. First only one workflow instance was executed. Afterwards two, three, four and finally five instances have been executed in parallel. The arithmetic mean was calculated when workflows have been executed in parallel because every workflow has its own warm-up time. The results are shown in Figure 2.

As the result shows, the warm-up time on the slowest device (ZTE Blade) takes over 15 seconds which can be seen as unacceptably long. Starting five workflows in parallel takes nearly a minute until their execution. However the ZTE Blade is a very weak and old device. The Samsung Galaxy Ace is not doing much better. The Sony Xperia U reflects an averaged performance smartphone nowadays. Warm-up time varies from about 3 to 22 seconds. The high-end smartphone Sony Xperia S needs just above 12 seconds for executing five workflows in parallel. The Samsung Galaxy Tab 10.1 as the only tablet device in the test is the fastest one by staying below 10 seconds for five workflows.

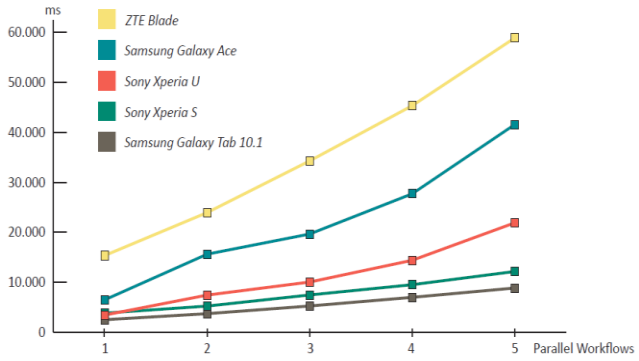

Figure 2. Comparing the workflow execution time on different devices.

As the Smartphone hardware gets more powerful, the warm-up times get shorter which is reflected by the measured values. Subsequent executions of the same workflow reuse the previously created runtime objects. As a result the execution starts instantly which significantly reduces the warm-up time.

To show this difference, we have executed the previous workflows again, but this time a caching strategy was applied to reuse the runtime objects of executed workflows. Table 6.3 shows the execution times for one workflow and also five workflows in parallel which clearly demonstrates the significant reduction of warm-up times. Subsequent executions start in milliseconds for all devices in contrast to seconds for the first execution. To reduce warm-up times for the first execution, runtime objects should be created in advance and pooled afterwards so that workflows have no performance drawback against native development.

Device	One Workflow	5 Workflows
ZTE Blade	110 ms	160 ms
Samsung Ace	2 ms	15 ms
Sony Xperia U	50 ms	75 ms
Sony Xperia S	5 ms	10 ms
Samsung Galaxy Tab 10.1	5 ms	8 ms

Table 4. The required time for execution of workflows after applying the object caching strategy

Memory consumption was measured for the created runtime objects. The average consumption for all devices was nearly the same because always the same workflow was executed. The XProcRuntime object consumes about 350kBytes memory and the XPipeline object about 330kBytes.

5. CONCLUSIONS

This work aimed to create a toolbox enabling users to automate simple tasks on their smartphone. Some requirements for a reasonable solution have been identified, which are all fulfilled by the implemented work. Workflows based on XProc build the center of the proposed solution containing steps to solve a user's problem. Workflows can have a trigger which automatically starts the workflow if a certain event occurs. To execute the workflows and as proof of concept, an open and extensible workflow engine for Android smartphones was implemented.

Since it is not possible to create workflows on the smartphone directly. They are created via a web application hosted at Google's App Engine (GAE. The current implementation does not contain a graphical workflow designer, however as future work we are planning to build a web-based editor for creating XProc workflows. The central workflow repository allows to publish and share workflows with a group of users. Users can search the repository for useful workflows and adapt them to their own needs. Despite the platform-neutral definition of workflows, some platforms (iOS, WP7/WP8) implement tighter security policies and prevent steps like sending text messages. Though implementing an XProc engine for those systems seems to be feasible. By doing this, the created situational applications and automated tasks can be exchanged among different smartphones.

6. REFERENCES

[1] Weiser, M., The computer for the 21st century, Scientific American, vol. 3, no. 3, 1991.

[2] Bader, G., He, W., Anjomshoaa, A., & Tjoa, A. M. (2012). Proposing a context-aware enterprise mashup readiness assessment framework. Information Technology and Management, 13(4), 377-387.

[3] JackBe Corporation, "A Business Guide to Enterprise Mashups," JackBe, 2008.

[4] Koschmider, A., Torres, V., & Pelechano, V. (2009, April). Elucidating the mashup hype: Definition, challenges, methodical guide and tools for mashups. In Proceedings of the 2nd Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web at WWW.

[5] Anjomshoaa, A., Tjoa, A. M., & Hubmer, A. (2010). Combining and integrating advanced IT-concepts with semantic web technology mashups architecture case study. In Intelligent Information and Database Systems (pp. 13-22). Springer Berlin Heidelberg.

[6] Prutsachainimmit, K., Chaisatien, P., & Tokuda, T. (2012). A mashup construction approach for cooperation of mobile devices. In Current Trends in Web Engineering (pp. 97-108). Springer Berlin Heidelberg.

[7] Red foundry, mobile made easy - red foundry is a complete solution for building and managing mobile apps. http://www.redfoundry.com , 2013.

[8] Microsoft onX, automate your life, https://www.onx.ms , 2013.

[9] Tasker, Android apps on google play, https://play.google.com/store/apps/details?id=net.dinglisch.android.taskerm , 2013.

[10] Two forty four a.m. LLC, "Locale for android." http://www.twofortyfouram.com/ , 2013.

[11] J. C. team, "jbpm - jboss community." http://www.jboss.org/jbpm , 2013.

[12] K. Verlaenen, Drools & jbpm: jbpm5 lightweight? Running on android!, http://blog.athico.com/2011/03/jbpm5-lightweight-running-on-android.html , 2011.

[13] W3C XProc, http://www.w3.org/XML/XProc/docs/langspec.html , 2013

[14] Calabash, http://xmlcalabash.com/, 2013