# Adaptive Service-Oriented Mobile Applications: A Declarative Approach*

Gianpaolo Cugola, Carlo Ghezzi, Leandro Sales Pinto, and Giordano Tamburrelli

DeepSE Group @ DEI - Politecnico di Milano, Italy
{cugola,ghezzi,pinto,tamburrelli}@elet.polimi.it

**Abstract.** Modern society increasingly relies on mobile devices and on distributed applications that use them. To increase development efficiency and shorten time-to-market, mobile applications are typically developed by composing together ad-hoc developed components, services available on-line, and other third-party mobile applications. To cope with unpredictable changes and failures, but also with the various settings offered by the plethora of devices, mobile applications need to be adaptive. We address this issue by proposing a declarative approach. The advantages of the proposed solution are demonstrated through an example inspired by an existing worldwide distributed mobile application.

## 1 Introduction

Mobile applications, commonly referred to as *apps*, are small-sized, efficient, modular and loosely coupled aggregates of software components developed with specific programming frameworks that depend on the target mobile platform. Their development imposes several challenges to modern software engineering. In particular, to achieve the desired efficiency in terms of development time and to exploit existing well established software solutions, apps are typically developed by composing together: (1) ad-hoc developed components, (2) existing services available on-line, (3) third-party apps, and (4) platform-dependent components to access device-specific hardware (e.g., camera, GPS, etc.).

The typical approach to develop such heterogeneous software artifacts follows a three step approach. Developers first start by conceiving the list of needed functionalities and they organize them in a suitable workflow of execution. Secondly, they evaluate the trade-offs between implementing such functionalities directly or resorting to existing services or third-party apps. Finally, they implement the app by integrating all the components together. Building apps as orchestrations of components, services and/or other third-party applications, however, introduces a direct dependency of the system with respect to external software artifacts which may evolve over time, fail, or even disappear, thereby compromising the application's functionality. Moreover, differently from traditional software

* This research has been funded by the EU, Programme IDEAS-ERC, Project 227977-SMScom and FP7-PEOPLE-2011-IEF, Project 302648-RunMore.

C. Liu et al. (Eds.): ICSOC 2012, LNCS 7636, pp. 607–614, 2012.

systems, the development of mobile apps is characterized by an increased explicit dependency with respect to hardware and software settings of the deployment environment. Indeed, even if developed for a specific platform (e.g., Android, iPhone, etc.), apps may be deployed on a plethora of different mobile devices characterized by heterogeneous hardware and software configurations (e.g., available sensors, firmware version, etc.). To cope with these peculiarities apps need to be *adaptive* [8] with respect to the heterogeneous deployment environments and with respect to the services and external apps they rely upon. The traditional way to achieve this goal is by explicitly programming the needed adaptations by heavily using exception handling techniques to manage unexpected scenarios when they occur. This is quite hard per-se and cannot be done by inexperienced users. This paper precisely address this issue by proposing a different approach. We abandon the mainstream path in favor of a strongly declarative alternative, called *SelfMotion*[1], which allows apps to be modeled in terms of the abstract functionalities they provide and the overall goal they have to met. SelfMotion apps are then executed by a middleware that leverages automatic planning techniques to elaborate, at run-time, the best sequence of activities to achieve the goal. Whenever a change happens in the external environment (e.g., a service becomes unavailable), which prevents successful completion of the execution, the middleware tries to find an alternative path toward the goal and continues executing the app, which results in a nice and effective self-healing behavior.

## 2 A Motivating Example: The ShopReview App

Let us now introduce *ShopReview* (SR), the mobile app we will use throughout the paper to explain our approach. SR is inspired by an existing application (i.e., ShopSavvy[2]). It allows users to share data concerning a commercial product or query for data shared by others. Users may use SR to publish the price of a product they have found in a certain shop (chosen among those close to their current location). In response, the app provides the users with alternative, nearby places where the same product is sold at a more convenient price. The unique mapping between the price signaled by the user and the product is obtained by exploiting the product barcode. In addition, users may share their opinion concerning the shop and its prices on a social network such as Twitter.

As introduced in the previous section, the development process for an app like SR starts by listing the needed functionalities and by deciding which of them will be implemented through an ad-hoc component and which will be implemented by re-using existing solutions. For example, the communication with social networks may be delegated to a third party app, while geo-localization of the user may be performed by a ad-hoc component which exploits the GPS sensor on the device.

Table 2 illustrates the abstract components uses as the main building blocks for the SR app. For the `BarcodeReader`, consider we decide to implement its code as for the original ShopSavvy app, which runs an ad-hoc developed component in

[1] Self-Adaptive Mobile Application.
[2] `http://shopsavvy.mobi/`

**Table 1.** ShopReview Components

| Name | Description |
|---|---|
| *BarcodeReader* | Allows the user to insert the barcode of the product |
| *GetProductName* | Translates the barcode into the product name |
| *GetPosition* | Retrieves the current user location |
| *LocalSearch* | Retrieves other shops in the neighborhood which offer the product at a more convenient price |
| *SharePrice* | Shares the price of a product on a given shop on Twitter |
| *InputPrice* | This component collects from the user the product's price |

```
if(manager.hasSystemFeature(PackageManager.FEATURE_CAMERA_AUTOFOCUS){
  //Run local barcode recognition
}else{ //Invoke remote service with blurry decoder algorithm }
//....
Location location = null;
if(manager.hasSystemFeature(PackageManager.FEATURE_LOCATION_GPS){
  LocationProvider provider = LocationManager.GPS_PROVIDER;
  try{
    //Return null if the GPS signal is currently not available
    location = locationManager.getLastKnownLocation(provider);
  }catch(Exception e){ location = null; }
}
if(location==null){
  //Device whitout GPS or an excpetion was raised invoking it. We show up a map
  //to allow the user to indicate its location manually
  showMap();
}
```

**Listing 1.1.** Adaptive Code Example

charge of acquiring a picture of the barcode from the mobile camera. Since such component may execute correctly only on devices with an autofocus camera and does not work properly on other devices, our choice would limit the usability of our app. To overcome this limitation and allow a correct barcode recognition also on devices with fixed focus cameras, SR needs to provide a form of adaptivity. Indeed, it has to detect if the camera on the current device is autofocus and, if not, it has to invoke an external service to process the acquired image with a special blurry decoder algorithm. A similar approach can be used to get the user location (i.e., GetPosition component), which requires a GPS sensor[3]. To execute SR on devices without GPS we may offer a different implementation, which shows a map to the user for a manual indication of the current location.

The code snippet reported in Listing 1.1 describes a possible implementation of the described adaptive behavior for the Android platform. Although this is just a small fragment of the SR app, which is by itself quite a simple example, it is easy to see how convoluted and error prone the process of defining all possible alternative paths may turn out to be. Things become even more complex considering run-time exceptions, like an error while accessing the GPS or invoking an external service, which have to be explicitly managed through ad-hoc code. We argue that the main reason behind these problems is that the mainstream platforms for developing mobile applications are based on traditional imperative languages in which the flow of execution must be explicitly programmed. In this setting, the adaptive code—represented in our code fragment by all the *if-else* branches—is intertwined with the application logic, reducing the overall readability and maintainability of the resulting solution, and hampering its fu-

[3] Network Positioning System is not precise enough for our needs.

ture evolution in terms of supporting new or alternative features, which requires additional branches to be added to the implementation.

## 3 The SelfMotion Approach

The SelfMotion approach comprises activities at design-time as well as at run-time. Initially, at design time, it requires the intervention of domain experts and software engineers, while at run-time it executes autonomously. Design-time activities are supported by a *declarative language*, while at run-time activities are supported by a *middleware*. At design time, domain experts and engineers must declare the following elements: (1) the app's *Goal*, expressed as a set of facts that are required to be true at the end of the app's execution; (2) the *Initial State*, which models the set of facts one can assume to be true at app invocation time; (3) a set of *Abstract Actions*, which models the primitive operations that can be executed to achieve the goal; (4) A set of *Concrete Actions*, one or more for each abstract action. Concrete actions map abstract ones to executable snippets that define the actual steps required for realizing them, e.g., by invoking an external service. At run-time, the SelfMotion middleware comes into play to actually execute the app. It comprises two distinct components: a *Planner* and an *Interpreter*. The Planner analyzes the goal, the initial state, and the abstract actions to build an *Abstract Execution Plan*, which lists the logical steps to reach the goal. The Interpreter is in charge of enacting this plan by associating each step (i.e., each abstract action) with the concrete action to execute, possibly invoking external components where specified. If something goes wrong (e.g., an external service returns an exception), the Interpreter first tries a different concrete action for the abstract action that failed. If no alternative action can be found or all alternatives have been tried unsuccessfully, it invokes the Planner again to build an alternative plan. From a deployment viewpoint the Interpreter is installed on the mobile device, since it is in charge of actually executing the app. The Planner, instead, may be deployed either locally or remotely.

### 3.1 The SelfMotion Declarative Language

**Abstract Actions.** Abstract actions are high-level descriptions of the primitive actions used to accomplish the app's goal. They represent the main building blocks of the app. Listing 1.2 illustrates the abstract actions for the SR reference example: they correspond to the high level components listed in Table 2. In some cases, the same functional component may correspond to several abstract actions, depending on some contextual information (e.g., if the device has a camera with autofocus or not). For example, we split the GetPosition functionality into two abstract actions `getPosWithGPS` and `getPosManually`. We also introduced an `enableGPS` abstract action, which encapsulates the logic to activate the sensor. Similarly, the `blurryDecoder` abstract action represents a remote component in charge of recognizing barcodes from pictures taken with fixed focus cameras. Together with the `blurryBarcodeReader` action it can read the barcode when an autofocus camera is not available.

```
action barcodeReader          action blurryBarcodeReader  action enableGPS     action inputPrice(Name)
pre:  hasAutoFocusCamera      pre:  hasFixedFocusCamera   pre:  ~isGPSEnabled  pre:  prodName(Name)
post: barcode(prodBarcode)    post: image(blurryImage)    post: isGPSEnabled   post: price(prodPrice)

action blurryDecoder(Image)   action getProdName(Barcode) action localSearch(Barcode, Pos)
pre:  image(Image)            pre:  barcode(Barcode)      pre:  barcode(Barcode), position(Pos)
post: barcode(prodBarcode)    post: prodName(name)        post: listOfLocalPrices

action getPosWithGPS          action getPosManually       action sharePrice(Name, Price)
pre:  hasGPS, isGPSEnabled    pre:  true                  pre:  prodName(Name), price(Price)
post: position(gpsPos)        post: position(manualPos)   post: sharedPrice
```

**Listing 1.2.** SR Abstract Actions

```
goal (listOfLocalPrices and sharedPrice and position(gpsPos)) or
     (listOfLocalPrices and sharedPrice and position(userDefinedPos))

start (hasFixedFocusCamera and hasGPS and ~isGPSEnabled)
```

**Listing 1.3.** SR Goal and Initial State

Abstract actions are modeled with an easy-to-use, logic-like language, in terms of: (1) *signature*, (2) *precondition*, and (3) *postcondition*. Signatures include a name and a list of arguments. For instance, the `localSearch` action has the following signature: `localSearch(Barcode, Pos)`. The precondition is expressed as a list of facts that must be true in the current state for the action to be enabled. For `localSearch` we use the expression `barcode(Barcode), position(Pos)` to denote the fact that the `Barcode` parameter is a product barcode, while the `Pos` parameter represents the user's position. The postcondition models the effects of the action on the current state of execution by listing the facts to be added to and the ones to be removed from the state. In our example, when `inputPrice` is executed the fact `price(prodPrice)` is added to the state, while no facts are deleted (deleted facts, when present, are designed by using the "~" symbol). Facts are expressed as propositions, characterized by a name and parameters, which represent relevant objects of the domain. Parameters that start with an uppercase letter denote *unbound objects*, which must be bound to instances, whose name starts with a lowercase letter, to generate an execution plan. For instance, if at any point the fact `position(gpsPos)` is added to the state, the object `gpsPos` becomes available to be bound to the `Pos` parameter in the `localSearch` action.

**Goal and Initial State.** Besides abstract actions, the goal and initial state are also needed to build and execute apps. The goal specifies the desired state after executing the app. It may actually include a set of states, which reflect all the alternatives to accomplish the app's goal, listed in order of preference. As an example, in the SR app (see Listing 1.3) we have two alternative goals. The first one requires the GPS sensor and the second relies on the user input to retrieve the location. The initial state complements the goal by asserting the facts that are true at app invocation time. It is partially generated at run time by the SelfMotion Middleware, which detects the features of the mobile device in which it has been installed. In our example, assuming the device has a fixed-focus camera and a disabled GPS, it generates the initial state shown in Listing 1.3. Developers may add application specific facts to this auto-generated initial state, if needed. By relying on abstract actions, goal, and initial state, the Planner can build an Abstract Execution Plan. The Planner starts trying to build an Abstract Execution Plan to satisfy the first goal; if it does not succeed

```
1: blurryBarcodeReader                  5: getProdName(prodBarcode)
2: enableGPS                            6: inputPrice(name)
3: blurryDecoder(blurryBarcodeImage)    7: localSearch(prodBarcode, gpsPos)
4: getPosWithGPS                        8: sharePrice(name, price)
```

**Listing 1.4.** A Possible Abstract Execution Plan

```
@Action(name="getProdName", priority=1)
public String getProdNameViaService(Barcode barcode){
  String barcodeValue = barcode.getValue();
  //Use remote Web service (e.g., searchupc.com)
  String productName = ...;
  return productName;
}
```

```
@Action(name="getProdName", priority=2)
public String getProdNameFromUser(Barcode barcode){
  String barcodeValue = barcode.getValue();
  //Ask the user for the product name
  String productName = ...;
  return productName;
}
```

**Listing 1.5.** `getProdName` Concrete Actions

it tries to satisfy the second goal, and so on. Listing 1.4 reports a possible plan of the SR example for a device without autofocus (i.e., `hasFixedFocusCamera` is set to true) and with a GPS sensors available but not enabled (i.e., `hasGPS` set to true, `isGPSEnabled` set to false). This Abstract Execution Plan is a list of abstract actions that lead from the initial state to a state that satisfies the goal. Notice that: (1) when several sequences of actions could satisfy the goal, the Planner chooses one non-deterministically; (2) although the plan is described as a sequence of actions, the middleware is free to execute them in parallel, as soon as the respective precondition becomes true.

**Concrete Actions.** Concrete actions are the executable counterpart of abstract actions. Currently, concrete actions are implemented through Java methods. We use the annotation `@Action` to refer to the abstract actions they implement. In general, several concrete actions may be bound to the same abstract action. This way, if the currently bound concrete action fails (i.e., it returns an exception) the SelfMotion middleware has other options to accomplish the app's step specified by the failed abstract action. For example, the `getProdName` abstract action may have two concrete actions: one which exploits a Web service (e.g., `searchupc.com`) to map the barcode value to the product name, and another which asks it to the user. Listing 1.5 reports the code used to define the concrete actions. Notice that, in presence of multiple concrete actions for the same abstract action, it is possible to specify a preferred ordering through the `priority` attribute.

### 3.2 Advantages of the SelfMotion Approach

**Decoupled Design.** SelfMotion achieves a clear separation among different aspects of the app: from the more abstract ones, captured by goals, initial state, and abstract actions, to those closer to the implementation, captured by concrete actions. In defining abstract actions developers may focus on the features they want to introduce in the app, ignoring how they are implemented (e.g., ad-hoc developed components, services, or third party apps). This choice is delayed until run-time binding. Consider the `GetProductName` component of the SR app. In the inception phase of the app, developers only focus on the features it requires – the preconditions – and the features it provides – the postconditions. Later on, they can implement a first prototype that leverages an ad-hoc component (i.e.,

manual input of the product name). This solution may gradually evolved, by adding other alternative concrete actions.

**Enable Transparent Adaptation.** By separating abstract and concrete actions and supporting one-to-many mappings we solve two typical problems of mobile apps: (1) how to adapt to the plethora of devices available, and (2) how to cope with failures happening at run-time. As an example of problem (1), consider the implementation of component `GetPosition` of Listing 1.1 with its SelfMotion counterpart, which relies on several abstract actions with different preconditions (see Listing 1.2). The former requires to explicitly hard-code the various alternatives (e.g., to handle the potentially missing GPS), and any new option introduced by new devices would increase the number of possible branches. Conversely, SelfMotion just requires a separate abstract (or concrete) action for each option, leaving to the middleware the duty of selecting the most appropriate ones, considering the current device capabilities and the order of preference provided by the app's designer. As for problem (2), consider the example of `GetProductName`, which is implemented in SelfMotion by a single abstract action mapped to two different concrete actions (Listing 1.5). The middleware initially tries the first concrete action that invokes an external service: if this returns an exception, the second concrete action is automatically tried. Furthermore, if none of the available concrete actions succeeds, SelfMotion may rely on its *re-planning* mechanism to build an *alternative plan* at run-time. As an example, consider the case in which the middleware is executing the plan reported in Listing 1.4 and assume that the GPS sensor fails to retrieve the user location, throwing an exception. The middleware automatically catches the exception and recognizes the `getPosWithGPS` as faulty, which has no alternative concrete actions. Thus, the Planner is invoked to generate a new plan that avoids the faulty step. The new plan would include the `getPosManually` abstract action.

**Improve Code Quality.** SelfMotion promotes a clean modularization of the app's functionality into a set of abstract actions and their concrete counterparts and avoids contorted code through cascaded *if-else*s and exception handling constructs. As a result, code is easy to read, maintain, and evolve. By encapsulating all the features in independent actions and by letting the actual flow of execution to be automatically built at run-time by the middleware, SelfMotion increases reusability, since the same actions can be reused across different apps.

## 4 Related Work

Many existing works focus on the effective and efficient development of mobile applications, as summarized in [5,11]. They cover a wide range of approaches: from how to achieve context-aware behavior (e.g., [6]) to how to apply agile methods in the mobile domain (e.g., [1]).

Context-aware frameworks aim at supporting the development of mobile applications that are sensitive to their deployment context (e.g., the specific hardware platform) and their execution context (e.g., user location). For example, the EgoSpaces middleware [6] can be used to provide context information extracted from data-rich environments to applications. Another approach to mo-

bile computing middleware is presented in [3], which exploits the principle of reflection to support adaptive and context-aware mobile capabilities. In general these approaches provide developers with abstractions to query the current context and detect context changes; i.e., they directly support context-dependent behavior as first-class concept. In the same direction, approaches like [2,10] provide specific context-aware extensions to the Android platform. The aforementioned approaches do not directly compete with ours, but rather they can be viewed as orthogonal. SelfMotion may benefit from their ability to detect context information, for example, to generate plans whose initial state depends on the surrounding context. The added value of SelfMotion is instead its ability to automatically build an execution flow based on the context and the overall design approach it promotes. Last, we would like to mention the foundational work on a three-layer architecture for software adaptation, described in [7,9], which shares with our work the motivation to provide sound architectural principles to the development of adaptive systems.

## 5 Conclusions and Future Work

SelfMotion is part of a long running research stream on declarative languages [4]. Future work includes building an IDE, possibly integrated in a widely adopted tool such as Eclipse, to further simplify the definition of abstract/concrete actions and goals. As for the SelfMotion middleware, while the current prototype is operational and publicly available, there is still space to further improve performance and robustness.

## References

1. Abrahamsson, P., Hanhineva, A., Hulkko, H., Ihme, T., Jäälinoja, J., Korkala, M., Koskela, J., Kyllönen, P., Salo, O.: Mobile-D: An Agile Approach for Mobile Application Development. In: OOPSLA 2004 (2004)
2. Appeltauer, M., Hirschfeld, R., Rho, T.: Dedicated Programming Support for Context-Aware Ubiquitous Applications. In: UBICOMM 2008 (2008)
3. Capra, L., Emmerich, W., Mascolo, C.: CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. IEEE Trans. Software Eng. (2003)
4. Cugola, G., Ghezzi, C., Sales Pinto, L.: DSOL: a declarative approach to self-adaptive service orchestrations. Computing (2012)
5. Dehlinger, J., Dixon, J.: Mobile application software engineering: Challenges and research directions. In: Workshop on Mobile Software Engineering (2011)
6. Julien, C., Roman, G.C.: Egospaces: Facilitating rapid development of context-aware mobile applications. IEEE Trans. Software Eng. (2006)
7. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: FOSE 2007 (2007)
8. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing adaptive software. Computer (2004)
9. Sykes, D., Heaven, W., Magee, J., Kramer, J.: From goals to components: a combined approach to self-management. In: SEAMS 2008 (2008)
10. van Wissen, B., Palmer, N., Kemp, R., Kielmann, T., Bal, H.: Contextdroid: an expression-based context framework for android. In: PhoneSense 2010 (2010)
11. Wasserman, T.: Software engineering issues for mobile application development. In: FoSER 2010 (2010)