SelfMotion: a Declarative Language for Adaptive Service-Oriented Mobile Apps

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

In this demo we present SelfMotion: a declarative language and a run-time system conceived to support the development of adaptive, mobile applications, built as compositions of ad-hoc components, existing services and third party applications. The advantages of the approach and the adaptive capabilities of SelfMotion are demonstrated in the demo by designing and executing a mobile application inspired by an existing, worldwide distributed, mobile application.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Service-oriented architecture (SOA)*

General Terms

Languages

Keywords

Mobile applications, Self-adaptive systems, Declarative orchestration language

1. INTRODUCTION

Mobile applications, commonly referred to as *apps*, are small-sized, efficient, modular and loosely coupled software artifacts typically developed by composing together: (1) ad-hoc developed components, (2) existing services available on-line, (3) third-party apps, and (4) hardware features (e.g., camera, GPS, etc.). Building apps as an orchestration 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. This kind of dependency increases when the app relies on hardware features, usually not present in all devices. To cope with these peculiarities apps need to be *adaptive* with respect to the heterogeneous deployment environments and with respect to the services and external apps they rely upon [1]. The traditional way to achieve this goal is by explicitly programming the needed adaptations, heavily using exception handling techniques to manage unexpected scenarios when they occur. Using this traditional approach it is hard to separate the alternative ways to achieve the app's goal from the exception handling code. This brings further complexity and results in hard to read and maintain apps. We tackle this problem by abandoning the mainstream path and by proposing an innovative approach called *SelfMotion*[1][2]. This approach comprises an interpreted *Declarative Language* and a *Middleware*, both presented in this demo. SelfMotion not only supports adaptivity but also provides an effective framework to decouple the business logic from the adaptation logic, facilitating code reuse and refactoring.

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.

2. SELFMOTION EXPLAINED

The SelfMotion language allows apps to be modeled by describing: (1) a set of *Abstract Actions*, (2) a set of *Concrete Actions*, and (3) the overall *Goal* to be met.

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. Abstract actions are modeled with an easy-to-use, logic-like language, in terms of: (1) *signature*, (2) *precondition*, and (3) *postcondition*. The precondition is expressed as a list of facts that must be true in the current state for the action to be enabled. The postcondition models the effects of the action on the current state of execution by listing the facts to be added to (and those to be removed from the state). For example, Listing 3 reports two abstract actions. The first one corresponds to a component called `getPositionWithGPS` which retrieves the user location via GPS, it requires as precondition an enabled GPS sensor and asserts with its postcondition the user location. The second abstract action corresponds to another component (i.e., `getProductName`) which converts an acquired barcode to a product name (we use these actions in our example, later on).

Goal and Initial State. Besides abstract actions, the goal and initial state are also needed to build and execute an app. 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. The SelfMotion middleware will start trying to satisfy the first goal; if it does not succeed it will

[1] Self-Adaptive Mobile Application.

```
action   getPositionWithGPS
pre :    hasGPS, isGPSEnabled
post:    position(gpsPosition)

action   getProductName(Barcode)
pre :    barcode(Barcode)
post:    productName(name)
```

Listing 1: Abstract Action Example.

try to satisfy the second goal, and so on. The initial state complements the goal by asserting the facts that are true at app's 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.

Concrete Actions. Concrete actions are the executable counterpart of abstract actions. In the current implementation of SelfMotion, concrete actions are implemented through Java methods. 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.

```
@Action(name="getProductName", priority=1)
public String getProductNameViaService(Barcode barcode){
  String barcodeValue = barcode.getValue();
  //Invoke remote service (e.g., http://searchupc.com/)
  return productName;
}

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

Listing 2: `getProductName` Concrete Actions.

For example, for the `getProductName` abstract action we may have two corresponding concrete actions: one exploiting a Web service (e.g., `http://searchupc.com/`) to map the barcode value to a product name, the other asking it to the user. The latter is automatically selected by the middleware when the former fails. Listing 2 reports the code used to define these concrete actions.

The Middleware. SelfMotion apps are executed by a middleware that leverages automatic planning techniques to elaborate, at run-time, the best sequence of abstract actions 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 SelfMotion middleware tries to find an alternative path toward the goal and continues executing the app, which results in a nice and effective adaptive behavior. The middleware's architecture comprises two distinct components: a Planner and an Interpreter. The Planner analyzes the goal, the initial state, and the abstract actions provided at design-time by the developer and builds an *Abstract Execution Plan*. Such plan lists the logical steps to reach the expected goal (i.e., a list of abstract actions that lead from the initial state to a state that satisfies the goal). The Interpreter is in charge of enacting the generated plan by associating each step (i.e., each abstract action) with a concrete action that is executed, possibly invoking external components where specified. If something goes wrong (e.g., an external service is unable to accomplish the expected task), the Interpreter first tries a different concrete action for the abstract action that failed. Afterwards, if none of the available concrete actions is able to complete successfully, the Interpreter invokes the Planner invoked again to build an alternative plan that avoids the failing step.

Enable Adaptation. By separating abstract and concrete actions and supporting one-to-many mappings we solve two key typical problems of mobile apps: (1) how to adapt the app to the plethora of devices available, and (2) how to cope with failures happening at run-time. Concerning problem (1), traditional approaches require to explicitly hard-code (using *if-else* constructs) the various alternatives (e.g., to handle the potentially missing sensor on a certain device), and any new option introduced by new devices would increase the number of possible branches. Conversely, SelfMotion just requires a separate 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 (Listings 1 and 2). 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. If none of the available concrete actions succeeds, SelfMotion may rely on its *re-planning* mechanism. Consider the case in which the middleware is executing a plan which include the location retrieval via GPS and let us assume that the GPS sensor fails throwing a system exception. The middleware catches the exception and recognizes the `getPositionWithGPS` as faulty, which has no alternative concrete actions. Thus the Planner is invoked to generate a new plan that avoids the faulty step for example by using an alternative abstract actions which asks the user to manually indicate its current location (i.e., `getPositionManually`). These examples show how SelfMotion relieves programmers from the need for explicitly handling the intertwined exceptional situations that may happen at run-time.

Decouple Design from Implementation. SelfMotion achieves a clear separation among the different aspects of the app: from the more abstract ones, captured by goals, initial state, and abstract actions, to those closer to the implementation domain, captured by concrete actions. In defining abstract actions developers may focus on the features hey want to introduce in the app, ignoring how they will be implemented (e.g., ad-hoc developed components, services, or third party apps). This choice is delayed until run-time binding. This way the app may be gradually evolved, by adding new concrete actions that implement the additional features, e.g., exploiting a Web service. This process, in which the system design is decoupled from the implementation, is not currently supported by mainstream app development environments. SelfMotion is an attempt to fill this gap. In addition the modularization of the app's functionality avoids the typical, spaghetti-like code required to merge all possible alternatives and exception handling fragments. As a result, code is easy to read, write, maintain, and evolve. In addition, SelfMotion increases reusability, since the same actions can be reused across different apps.

Table 1: ShopReview Components.

Name	Description
BarcodeReader	It allows to scan the product barcode
GetProductName	It maps the barcode in a product name the barcode into the product name
GetPosition	It retrieves the user location
WebSearch	It retrieves more convenient prices offered online
LocalSearch	It retrieves online other shops in the neighborhood which offer the product at a more convenient price exploiting the data provided by other users of the app.
SharePrice	It allows users to share the product's price on Twitter
InputPrice	It collects from the user the product's price

3. THE DEMO

The demo illustrates SelfMotion by designing and executing a realistic mobile application for the Android platform. In particular, the demo is organized in two parts: (1) *App Design* and (2) *App Execution*. In the first part we design and develop the app, highlighting the SelfMotion peculiarities and emphasizing the advantages of the approach with respect to the state-of-the-art. Subsequently, in the App Execution part, we run the app on an emulator as well as on a real mobile device to illustrate the SelfMotion middleware. The combination of the two parts of the demo clearly provides a typical usage scenario of the language and the tool.

The mobile app we develop in the demo is called *ShopReview* (SR) and is inspired by an existing application (i.e., ShopSavvy[2]). SR 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.

The development process for an app like SR typically 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 (i.e., services or third party apps). Table 1 lists all the building blocks that compose the SR app.

3.1 App Design

The demo starts by modeling the SR app in terms of abstract actions as reported in Listing 3 where each action corresponds to the high level components listed in Table 1. In some cases, the same functional component corresponds to several abstract actions, depending on some contextual information. For example, we split the GetPosition functionality into two abstract actions `getPositionWithGPS` and `getPositionManually`. We also introduced an `enableGPS` abstract action, which encapsulates the logic to activate the sensor. At this step we also discuss the details related to the language syntax, such as parameters of pre/postconditions (e.g., *Position* or *Barcode*). These parameters start with an uppercase letter, which denote *unbound objects*. Unbounded objects are bound at planning time to instances, whose name starts instead with a lowercase letter.

[2] `http://shopsavvy.mobi/`

```
action  barcodeReader
pre :   true
post :  barcode(productBarcode)

action  getProductName(Barcode)
pre :   barcode(Barcode)
post:   productName(name)

action  inputPrice(Name)
pre :   productName(Name)
post :  price(productPrice)

action  sharePrice(Name, Price)
pre :   productName(Name), price(Price)
post :  sharedPrice

action  getPositionWithGPS
pre :   hasGPS, isGPSEnabled
post:   position(gpsPosition)

action  getPositionManually
pre :   true
post:   position(userDefinedPosition)

action  enableGPS
pre :   ~isGPSEnabled
post:   isGPSEnabled

action  webSearch(Name)
pre :   productName(Name)
post :  listOfOnlinePrices

action  localSearch(Barcode, Position)
pre :   barcode(Barcode), position(Position)
post :  listOfLocalPrices
```

Listing 3: SR Abstract Actions.

```
goal
listOfLocalPrices and listOfOnlinePrices and sharedPrice
and position(gpsPosition)
or
listOfLocalPrices and listOfOnlinePrices and sharedPrice
and position(userDefinedPosition)

start hasGPS and ~isGPSEnabled
```

Listing 4: SR Goal and Initial State.

At this stage the demo proceeds by declaring the goal for the SR app (see Listing 4), which is composed in turn by two alternative goals. The first one requires the GPS sensor and the second one relies on the user input to retrieve the location (this way the app will be adaptive w.r.t. the GPS sensor, which may be potentially missing on certain devices). As described in the previous section the initial state is 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 GPS sensor which is currently disabled, it generates the initial state shown in Listing 4, which will prevent the Planner to generate a plan including the `getPositionWithGPS` action. After these steps, the demo focuses on illustrating

```
enableGPS
barcodeReader
getPositionWithGPS
getProductName(barcode)
inputPrice(name)
webSearch(name)
localSearch(barcode, gpsPosition)
sharePrice(name, price)
```

Listing 5: A Possible Abstract Execution Plan.

the concrete actions that implement the abstract actions, similarly to what reported in Listing 2 in which we showed the concrete actions associated to the `getProductName` abstract action. Notice that during this part of the demo we highlight the difference and the advantages of our approach with respect to the traditional imperative paradigm. Indeed, to execute SR on devices without GPS, its implementation include an adaptive behavior which shows a map to the user for a manual indication of the current location if GPS is missing. The code snippet reported in Listing 6 describes a possible implementation of the described adaptive behavior for the Android platform [4]. 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.

```
if(manager.hasSystemFeature(FEATURE_CAMERA_AUTOFOCUS){
  //Run local barcode recognition
}else{
  //Invoke remote service with blurry decoder algorithm
}
Location location = null;
if(manager.hasSystemFeature(FEATURE_LOCATION_GPS){
  LocationProvider provider =
                    LocationManager.GPS_PROVIDER;
  try{
    /* The following line returns null if the
       GPS signal is 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*/
  showMap();
}
```

Listing 6: Traditional Adaptive Code Example

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 future evolution in terms of supporting new or alternative features, which requires additional branches to be added to the implementation. In the demo we actually compare this solution with the SelfMotion counterpart, which relies on several abstract actions with different preconditions (see Listing 3). We also demonstrate these concepts to the case of the third-party apps invoked to obtain specific functionalities, like those used by SR to access the different social networks. These apps are typically installed by default on devices but they can be removed by users, thus jeopardizing the app's ability to accomplish its tasks.

3.2 App Execution

The second part of the demo illustrates the SelfMotion middleware by running the SR app. Indeed, given abstract actions, goal, and initial state, the Planner can build an Abstract Execution Plan. Listing 5 reports a possible plan for the SR example with a GPS sensors available but not enabled (i.e., `hasGPS` set to true, `isGPSEnabled` set to false). 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. In the demo we launch the middleware which executes the app in the Android emulator as well as on a real device. The execution shows that the middleware does not introduce any significant delay or overhead at run-time.

In addition, in this part of the demo we illustrate the adaptive capabilities of the planner. For example we artificially inject a fault in the concrete action `getProductName` which invokes the external Web service (see Listing 2) simulating an unresponsive or faulty service and we show how the planner automatically switches to the alternative concrete action. This way we point out how the SelfMotion approach is able to adapt to unexpected scenarios without requiring the convoluted code exemplified in Listing 6. We also deploy the app with two different settings of the emulator representing two devices with and without the GPS to demonstrate the adaptivity of the approach with respect to the plethora of devices available today.

SelfMotion is available, together with the ShopReview example, at `http://www.dsol-lang.net/self-motion.html`. This work is part of a long running research stream, which aims at investigating declarative approaches to enforce adaptive capabilities in software systems (e.g., [2, 3]).

4. REFERENCES

[1] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, et al. Software engineering for self-adaptive systems: A research roadmap. *Software Engineering for Self-Adaptive Systems*.

[2] G. Cugola, C. Ghezzi, L. P. Pinto, and G. Tamburrelli. Adaptive service-oriented mobile applications: A declarative approach. In *Service-Oriented Computing-ICSOC 2012 (to appear)*.

[3] G. Cugola, C. Ghezzi, and L. S. Pinto. DSOL: a declarative approach to self-adaptive service orchestrations. *Computing*, pages 1–39.

[4] R. Rogers. *Android application development*. O'Reilly, 2009.