SelfMotion: A Declarative Approach for Adaptive Service-Oriented Mobile Applications

Gianpaolo Cugola[a], Carlo Ghezzi[a], Leandro Sales Pinto[a], Giordano Tamburrelli[b]

[a] *Politecnico di Milano, Dipartimento di Elettronica e Informazione*
Piazza Leonardo da Vinci 32, Milan, Italy
[b] *University of Lugano, Faculty of Informatics*
Via Buffi 13, Lugano, Switzerland

Abstract

Modern society increasingly relies on mobile devices. This explains the growing demand for high quality software for such devices. To improve the efficiency of the development life-cycle, shortening time-to-market while keeping quality under control, mobile applications are typically developed by composing together ad-hoc developed components, services available on-line, and other third-party mobile applications. Applications are thus built as *heterogeneous compositions*, whose characteristics strongly depend on the components and services they integrate. To cope with unpredictable changes and failures, but also with the various settings offered by the plethora of available devices, mobile applications need to be as adaptive as possible. However, mainstream adaptation strategies are usually defined imperatively and require complex control strategies strongly intertwined with the application logic, yielding to applications that are difficult to build, maintain, and evolve. We address this issue by proposing a declarative approach to compose adaptive heterogeneous mobile applications. The advantages of this approach are demonstrated through an example inspired by an existing worldwide distributed mobile application, while the implementation of the proposed solution has been validated through a set of simulations and experiments aimed at illustrating its performance.

Keywords: Mobile applications, Self-adaptive systems, Declarative Language

1. Introduction

Software is a fundamental asset of modern society. Nowadays, most human activities are either software enabled or entirely managed by software. The recent massive adoption of mobile devices—such as smartphones and tablet PCs—which support people in their daily tasks, makes this phenomenon even more relevant. Mobile devices make software literally ubiquitous and pervasive, creating an increasing demand for high quality mobile applications to meet societal needs.

"Invented" by Apple for its iOS operating system and successively adopted by Google for the Android OS, *apps* are driving the growth of this mobile phenomenon. They are usually small-sized, often distributed and single-task applications, which the user may easily download (often for free) and install on her device to empower it with new capabilities with respect to those that come pre-installed.

The mobile market that enables this interaction is an extremely dynamic and vibrant ecosystem characterized by thousands of new apps published worldwide every week. This is posing new challenges to modern Software Engineering, first and foremost the need for effective development strategies centered around strong time-to-market constraints. To answer this challenge while keeping the various qualities of developed software under control, a component-based development process is usually adopted. This is enabled by the same development frameworks that come with modern mobile OSs, which allow components installed on the same device to easily communicate and invoke each other. As a result, most mobile apps are 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 (possibly iterative) three-step approach. Developers first conceive the list of needed functionality and they organize them in a suitable workflow of execution. Secondly, they evaluate the trade-offs between implementing such functionality directly or resorting to existing services or third-party apps. Finally, they build the app by implementing the needed components and integrating all the pieces 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 systems, the development of mobile apps is characterized by an increased, often explicit dependency with respect to hardware and software settings of the deployment environment. Indeed, even if devel-

Email addresses: gianpaolo.cugola@polimi.it (Gianpaolo Cugola), carlo.ghezzi@polimi.it (Carlo Ghezzi), pinto@elet.polimi.it (Leandro Sales Pinto), giordano.tamburrelli@usi.ch (Giordano Tamburrelli)

oped for a specific platform (*e.g.,* iOS or Android), the same app may be deployed on a plethora of different devices characterized by heterogeneous hardware and software configurations (*e.g.,* available sensors and networking hardware, list of pre-installed components, OS version, etc.). As an example, consider the case of an iPhone application using the built-in camera. The current iPhone has an auto focus camera while previous versions, still in widespread use, were equipped with fixed focus cameras. As we will show in our running example, this difference, albeit apparently minor, if left unmanaged may impact the application's ability to satisfy its requirements.

To cope with these peculiarities apps need to be *adaptive* (Cheng et al. (2009); McKinley et al. (2004)), both with respect to the heterogeneous deployment environments and with respect to the external services and apps they rely upon. The traditional way to achieve this goal is to explicitly program the needed adaptations by heavily using branches in the execution flow and exception handling techniques to manage unexpected scenarios when they occur. This is not easy to do and results in complex code that intertwines the application logic with the logic to cope with the peculiarities of each device and with unexpected situations that may happen at run-time. This brings further complexity, resulting in hard to read and maintain code.

This paper precisely addresses 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 by describing: (1) a set of *Abstract Actions*, which provide a high-level description of the elementary activities that realize the desired functionality of the app, (2) a set of *Concrete Actions*, which map the abstract actions to the actual steps to be performed to obtain the expected behavior (*e.g.,* invoking an external service or calling a pre-installed, third-party application), (3) a *QoS Profile* for each concrete action that models its non-functional characteristics (*e.g.,* energy and bandwidth consumption), and (4) the overall *Goal* to be met and the *QoS Policy* to be adopted in reaching such goal (*e.g.,* minimizing power consumption).

SelfMotion apps are then executed by a middleware that leverages automatic planning techniques to elaborate, at runtime, the best sequence of abstract actions to achieve the goal, mapping them to the concrete actions to execute in accordance with the specified QoS Policy. Whenever a change happens in the external environment (*e.g.,* a service becomes unavailable) that prevents successful completion of the defined plan of execution the middleware automatically – and transparently with respect to the user – builds an alternative plan toward the goal. This reifies in a nice and effective self-healing behavior that allows the app to seamlessly continue its execution.

In this paper we describe our approach in details, and we show, through a set of experiments, its effectiveness and its performance, showing how the approach based on a planner scales well even when the goal becomes complex and requires, to be satisfied, several activities (*i.e.,* abstract and concrete actions) to be called in the correct order.

SelfMotion brings several contributions and advantages with respect to the existing solutions in the area of self-adaptive and context-aware mobile apps:

1. The proposed solution represents the first attempt to support the design and development of adaptive mobile apps that relies on planning as well as on a declarative language.

2. SelfMotion represents a novel approach to adaptive mobile apps that conjugates functional adaptivity and non-functional awareness. More precisely, the former is achieved through planning, while the latter is obtained with QoS profiles and policies.

3. We contribute to the area of mobile development investigating the intersection of mobile app and services and in particular shedding light on the adaptivity of mobile apps achieved via service re-binding.

SelfMotion was initially introduced in Cugola et al. (2012b,c). Beyond a significantly more detailed description of the approach, this paper reports on several new contributions and experiments. First, we extended SelfMotion by introducing the support for QoS policies and profiles. This is the subject of Sections 3.2.4 and 3.2.5. Second, we extended and improved our validation of the approach, using not only a real-world mobile application to qualitatively evaluate the approach, but also running several synthetic simulations aimed at stressing its scalability and performance. This is the subject of Section 5. More precisely, for a clear and effective explanation of the proposed approach, we rely on a realistic mobile app illustrated in Section 2 and used as a reference example throughout the paper. The SelfMotion approach is described in detail in Section 3, while Section 4 discusses its advantages with respect to the state of the art. Section 5 evaluates the performance of SelfMotion in several scenarios of growing complexity, while Section 6 discusses related work. Finally, Section 7 draws some conclusions and briefly illustrates future work.

2. A Motivating Example: The ShopReview App

Let us now introduce *ShopReview* (SR), the mobile app we will use throughout the paper. SR is inspired by an existing application (*i.e.,* ShopSavvy[2]) and it allows users to share various information concerning a commercial product. In particular, an SR user may use the app to publish the price of a product she found in a certain shop (chosen among those close to her current location). In response, the app provides the user 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 its barcode. In addition, users may share their opinion concerning the shop where they bought the product and its prices on a social network, such as Twitter.

[1] Self-Adaptive Mobile Application.

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

As already mentioned, the development process for an app like SR starts by listing the needed functionality and by deciding which of them have to be implemented through an ad-hoc component and which can be realized by re-using existing solutions (*i.e.,* external services available online or third party apps that can be found pre-installed on the device or that can be installed on demand). For example, the communication with social networks may be delegated to a third party app to be installed on demand, while geo-localization of the user may be performed by exploiting a pre-existing component that accesses the GPS sensor on the device.

In making these choices developers have to remember that run-time conditions may change and may subvert design-time assumptions, impacting on the ability of the app to operate correctly. As an example, developers must consider the differences in the various devices that will run their app to let it *adapt* to these different devices. Similarly, they have to make the right choices to minimize the impact of changes in the external services they rely upon, either letting the app adapt to those changes or not using them at all, with the result of being forced to re-implement a functionality that may be easily found on line.

Given these premises, let us assume we choose the functionalities listed in Table 1 as the main building blocks for the SR app. Let us also assume we decide to realize the `ReadBarcode` functionality as an ad-hoc developed component that extracts the product's barcode from a picture taken using the mobile camera.[3] Since such component may execute correctly only on devices with an auto focus 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 supports auto-focus; if it does 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.,* to implement the `GetPosition` functionality), which in principle requires a GPS,[4] a hardware component that may not be available on every device. To execute SR on devices lacking a GPS we may offer a different implementation of the `GetPosition` functionality, which shows a map to the user for a manual indication of the current location.

The code snippet reported in Listing 1 describes a possible implementation of the described adaptive behavior for the Android platform (Rogers et al. (2009)). Although this is just a small fragment of the SR app, which is by itself quite a simple app, 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 Listing 1 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.

```
1 PackageManager mng = getPackageManager();
2 if(mng.hasSystemFeature(PackageManager.FEATURE_CAMERA_AUTOFOCUS)){
3   //Run local barcode recognition
4 }else{
5   //Invoke remote service with blurry decoder algorithm
6 }
7
8 Location location = null;
9 if(mng.hasSystemFeature(PackageManager.FEATURE_LOCATION_GPS)){
10   LocationProvider provider = LocationManager.GPS_PROVIDER;
11   LocationManager locManager =
12         (LocationManager) getSystemService(Context.LOCATION_SERVICE);
13   try{
14     //Return null if the GPS signal is currently not available
15     location = locManager.getLastKnownLocation(provider);
16   }catch(Exception e){
17     location = null;
18   }
19 }
20
21 if(location==null){
22   //Device whitout GPS or an excpetion was raised invoking it.
23   //We show up a map to allow the user to indicate
24   //its location manually
25   showMap();
26 }
```

Listing 1: Adaptive Code Example.

Notice that these concepts apply also to the case of the third-party apps invoked to obtain specific functionality, like those used by SR to access the various 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. The SelfMotion Approach

Here we introduce the SelfMotion approach and explain how to design an app like SR to achieve a form of self-adaptation that overcomes the problems discussed above.

3.1. *Introducing* SelfMotion

To help developing *adaptive* mobile applications SelfMotion adopts a novel approach, which includes several steps both at design-time and run-time. At design-time it supports the activities of domain experts and software engineers through a multi-layer *declarative language*, which supports the design of an application through different abstraction levels, while at run-time it offers a *middleware*, which uses planning techniques to reach the app's goals, adapting to the various situations that that may be encountered. More specifically, as shown in Figure 1, a SelfMotion application includes the following layers:

1. The app's *Goals*, expressed as a set of facts that are required to be fulfilled by the app's execution;

[3] This is the choice made by the original ShopSavvy app.

[4] We are assuming that a Network Positioning System is not precise enough for our needs.

Name	Description
GetPosition	Retrieves the current user location
InputPrice	Collects the product's price from the user
ReadBarcode	Acquires the barcode of the product
GetProductName	Translates the barcode into the product name
SearchTheWeb	Retrieves, through the Internet, more convenient prices offered on e-commerce sites
SearchTheNeighborhood	Retrieves, through the Internet, other nearby shops which offer the product at a more convenient price
SharePrice	Lets the user share the price of a product found on a given shop on Twitter

Table 1: ShopReview functionality.

2. the *Initial State*, which models the set of facts one can assume to be true at app's invocation time. It includes application-specific facts specified at design-time and context-specific facts, automatically derived by the SelfMotion middleware at run-time, like the availability of a GPS device or the presence of an auto-focus camera;

3. A set of *Abstract Actions*, which specify the primitive operations that can be executed to achieve the goal;

4. A set of *Concrete Actions*, one or more for each abstract action, which map them to the executable snippets that implement them (*e.g.,* by invoking an external service);

5. A *QoS Profile* for each concrete action, which models its non-functional characteristics (*e.g.,* energy and bandwidth consumption);

6. The *QoS Policy* to be adopted in reaching the goal (*e.g.,* minimizing energy consumption).

At run-time, the *Interpreter* translates the goal, the initial state, and the abstract actions into a set of rules and facts, used by the *Planner* to build an abstract execution plan, which lists the logical steps through which the desired goal may be reached. This plan is taken back by the Interpreter to be enacted by associating each step (*i.e.,* each abstract action) with the concrete action that may better satisfy the given QoS policy. These concrete actions are then executed, possibly invoking external services, third-party apps, or ad-hoc components. If something goes wrong the SelfMotion middleware adapts to the new situation by looking for alternative concrete actions to accomplish the failed step of execution or by invoking the Planner again to avoid that step altogether.

3.2. *The* SelfMotion *Declarative Language*

This section provides a detailed description of the fundamental concepts behind the SelfMotion declarative language.

3.2.1. *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 2 illustrates the abstract actions for the SR reference example: they correspond to the high level functionalities listed in Table 1. Note that, in some cases, the same functionality may correspond to several abstract actions, depending on some contextual information (*e.g.,* if the device has an auto focus camera or not). For example, we split the `GetPosition` functionality into two abstract actions `getPositionWithGPS` (lines 1-3) and `getPositionManually` (lines 9-11). We also introduce an `enableGPS` abstract action (lines 5-7), which encapsulates the logic to activate the GPS. Similarly, the `blurryReadBarcode` abstract action (lines 25-27) represents a component in charge of recognizing barcodes from pictures taken with fixed focus cameras.

Abstract actions are modeled with an easy-to-use, logic-like language, in terms of their *signature*, a *pre-condition*, and a *post-condition*. The signature provides the action name and its list of arguments. The precondition is expressed as a list of facts that must be true in the current state for the action to be enabled. As an example, for the `searchTheNeighborhood` action (lines 37-39) we use the expression `barcode(Barcode), position(Position)` to denote the fact that the `Barcode` argument must be a product barcode, while the `Position` argument must represent the user's position. The post-condition 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 current state. As an example, when the `inputPrice` action (lines 13-15) is executed the fact `price(productPrice)` is added to the state, while no facts are deleted (deleted facts, when present, are denoted by using the "~" symbol).

Facts are expressed as *propositions*, characterized by a name and a set of parameters, which represent relevant objects of the domain. By convention, parameters that start with an uppercase letter denote *unbound objects*; they must be bound to instances, whose name starts with a lowercase letter, to generate a valid execution plan. For instance, if at any point the fact `image(barcodeImage)` is added to the state, the object `barcodeImage` becomes available to be bound to the `Image` parameter in the `readBarcode` or `blurryReadBarcode` actions.

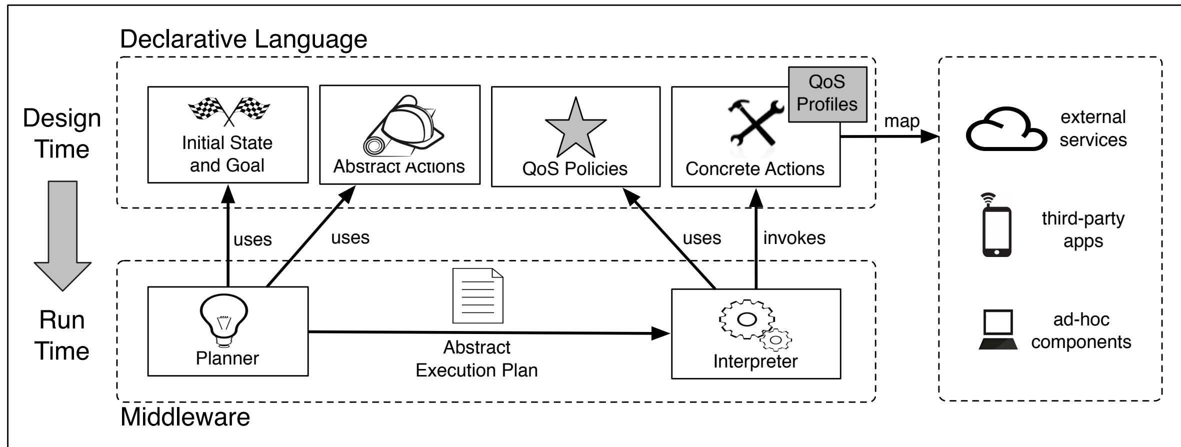


Figure 1: SelfMotion Conceptual Architecture.

Name	Description
hasGPS	The device has a GPS sensor
isGPSEnabled	The device has a GPS sensor and it is enabled
hasCamera	The device has a camera
hasAutoFocusCamera	The device has a camera and it supports auto-focus
hasFixedFocusCamera	The device has a camera but it does not support auto-focus
lowBattery	The device's battery level is low

Table 2: Example of facts automatically added to the initial state by the SelfMotion middleware.

3.2.2. Goal and Initial State

Besides abstract actions, the goal and initial state are also used to model apps in SelfMotion. The goal specifies the desired state resulting from the app's execution. One may actually specify a set of states, which reflect all the alternatives to accomplish the app's goal, listed in order of preference. The Planner will start by trying to build an execution plan to satisfy the first goal; if it does not succeed it will try to satisfy the second goal, and so on. As an example, in the SR app (see Listing 3) we have two alternative goals. The first one requires the GPS sensor and the second relies on the user input to retrieve the current location.

The initial state complements the goal by asserting the facts that are true at app's invocation time. It includes application-specific facts asserted by app's designers at design-time and context-specific facts automatically added at run-time by the SelfMotion middleware, which detects the features of the mobile device in which it has been installed. Table 2 illustrates some examples of the latter. Note that they are added in negated form if a given fact is not true, *e.g.,* `~hasGPS` is included to the initial state if the current device does not have a GPS sensor.

For the SR app, no application-specific fact is included in the initial state, which is fully populated by the SelfMotion middleware. Assuming that SR is deployed in a device equipped with a fixed-focus camera and with a GPS sensor that is currently disabled, the initial state becomes the one shown in Listing 4.

3.2.3. Concrete Actions

Concrete actions are the executable counterparts of abstract actions. In general, several concrete actions may be bound to the same abstract action. For example (see Table 3), in our SR app we have different implementations for some of the abstract actions. The `getProductName` abstract action can be mapped to three concrete actions: two of them exploit a remote Web service (*i.e.,* `searchupc.com` and `simpleupc.com`) to map the barcode to a product name, while the third one explicitly asks the product name to the user. Having multiple concrete actions for the same abstract one allows the SelfMotion middleware to choose the one that better satisfies the QoS policy (more on this later) but, most important, it allows the Interpreter to overcome unexpected situations in which a given concrete action does not execute successfully (*e.g.,* a web service fails) by invoking an alternative concrete action.

As for the actual code of concrete actions, in our current SelfMotion prototype, which runs on Android, they are implemented as Java methods, extended with ad-hoc annotations. For instance, we use the annotation `@Action` to refer to the implemented abstract action, as in Listing 5, which shows the three concrete actions that reify the `getProductName` abstract action (we will come back later to this example, explaining the exact meaning of the other annotations).

3.2.4. QoS Profiles

The concrete actions mapped to the same abstract one are functionally equivalent but they may differ in several non-functional aspects. For instance, consider the `getProductName` abstract action and the three corresponding concrete actions reported in Table 3. Those that rely on a remote service are characterized by a higher energy consumption with respect to the one that rely on the input manually provided

Abstract Actions	Concrete Actions
getPositionWithGPS	Ad-hoc Component (user localization via GPS)
enableGPS	Ad-hoc Component (enable GPS sensor)
getPositionManually	Ad-hoc Component (manual user localization)
inputPrice	Ad-hoc Component (textual input from the user)
acquirePhoto	Ad-hoc Component (photo acquisition from the mobile camera)
readBarcode	Ad-hoc Component (local barcode recognition)
burryReadBarcode	WebService (remote barcode recognition)
getProductName	Web service (`http://searchupc.com/`) Web service (`http://simpleupc.com/`) Ad-hoc Component (textual input from the user)
searchTheWeb	Web service (`http://www.kelkoo.it/`) Web service (`http://www.buscape.com/`)
searchTheNeighborhood	Web service (`http://shopsavvy.mobi/`)
sharePrice	Third-party app (UberSocial: `http://ubersocial.com/`) Third-party app (Twicca: `http://twicca.r246.jp/`) Web service (`https://dev.twitter.com/`)

Table 3: ShopReview Concrete Actions.

by the user. Thus, from an energy perspective, the last option is preferable. Conversely, considering usability, the concrete action that needs the user intervention is less preferable. Finally, considering cost, one of the three alternative relies on a Web service that charges a fee on a per-request basis (*i.e.,* `simpleupc.com`), while the others do not have any associated cost.

SelfMotion allows developers to declare all these non-functional aspects by relying on the `@QoSProfile` annotation, as illustrated in Listing 5. In particular, this annotation contains two lists of parameters: `metrics` and `values`. The list of metrics allows developers to declare the QoS attributes they are interested in. In the example, the list of metrics includes *usability*, *cost*, and *energy*. The second list contains the value associated with each metric. For example, concerning energy consumption, the actions that invoke remote services are annotated with -1, while the action that performs a local computation is annotated with 0. With these values we express the fact that remote invocations affect the battery usage more than local computation. Similarly, concerning usability, we annotate the three actions with different impact values to indicate that the automatic alternatives are preferable over those which bother the user asking for an explicit input. Finally, concerning cost, we annotated with 1 the action that invokes the `simpleupc.com` service since it charges a fee for each invocation.

Summing up, by relying on the `@QoSProfile` annotation, we are able to characterize the non-functional behavior of concrete actions. In particular, it is important to notice that, using the described approach we do not need to necessarily know the real QoS values of each alternative concrete action but only their *relative difference* (this also depends on the way the QoS Policy is specified, see later). In other words, considering for example the energy consumption, we do not need to know the actual energy consumed by each action but only the fact that those actions that use the network consume more energy than those that only perform local computations. This brings two significant advantages. First, we may ignore the real QoS values, which may be difficult to measure and dependent on the specific device. Second, this approach allow us to express application-specific QoS values, such as usability, which can hardly be measured to produce an absolute value, but rather may be more easily stated in relative terms with respect to different alternatives.

3.2.5. QoS Policies

Given the QoS characterization as described so far, it is also necessary to instruct the SelfMotion middleware about the different policies used to guide, at run-time, the Interpreter in prioritizing metrics, comparing their associated values, and choosing the best concrete actions to execute.

A QoS policy is defined in the SelfMotion language with the keyword `qos` followed by the name of the policy. In addition, each policy definition contains: (1) a pre-condition, similar to that of abstract actions, and (2) an ordered list of QoS preferences decorated with the `min` and `max` keywords.

Since a SelfMotion application may have multiple QoS Policies, pre-conditions are used to enable or disable each policy. In particular, at start-up, the Interpreter evaluates the policies in order and adopts the first one whose pre-condition is enabled in the initial state.

Let us consider Listing 6, which reports two possible QoS policies for the SR example: `energySaver` and `default`. Imagine the scenario in which the middleware set in the initial state the fact `lowBattery`, indicating the current low state of the battery. In this case, the first policy with a valid pre-condition is `energySaver` and, as a consequence, the SR application will be executed using this specific policy. In particular, `energySaver` is composed by three ordered constraints: (1) `min: energy`, (2) `max: usability`, and (3) `min: cost`. The three constraints will be applied in order. Every time the Interpreter must execute an abstract action with many corre-

```
1  action   getPositionWithGPS
2  pre  :   hasGPS, isGPSEnabled
3  post :   position(gpsPosition)
4
5  action   enableGPS
6  pre  :   ~isGPSEnabled
7  post :   isGPSEnabled
8
9  action   getPositionManually
10 pre  :   true
11 post :   position(userDefinedPosition)
12
13 action   inputPrice(Name)
14 pre  :   productName(Name)
15 post :   price(productPrice)
16
17 action   acquirePhoto
18 pre  :   hasCamera
19 post :   image(barcodeImage)
20
21 action   readBarcode(Image)
22 pre  :   image(Image), hasAutoFocusCamera
23 post :   barcode(productBarcode)
24
25 action   blurryReadBarcode(Image)
26 pre  :   image(Image), hasFixedFocusCamera
27 post :   barcode(productBarcode)
28
29 action   getProductName(Barcode)
30 pre  :   barcode(Barcode)
31 post :   productName(name)
32
33 action   searchTheWeb(Name)
34 pre  :   productName(Name)
35 post :   prices(onlinePrices)
36
37 action   searchTheNeighborhood(Barcode, Position)
38 pre  :   barcode(Barcode), position(Position)
39 post :   prices(localPrices)
40
41 action   sharePrice(Name, Price)
42 pre  :   productName(Name), price(Price)
43 post :   priceShared
```

Listing 2: ShopReview Abstract Actions.

sponding concrete actions, it will invoke the one with minimum required energy. If this criterion does not result in the selection of a unique concrete action (*i.e.,* many actions have the same minimum energy value), the Interpreter applies the second constraint (*i.e.,* the maximum usability) to the set of actions with the minimum energy value. If even this criterion is not able to identify a unique candidate, the Interpreter applies the third constraint (*i.e.,* minimum cost). If neither this is enough to find a unique concrete action to invoke, the Interpreter chooses non-deterministically among the available actions. The same occurs if all actions do not have an associated QoS Profile or if none of the existing QoS policies has a valid precondition. Conversely, if the battery is fully charged, the middleware set in the initial state the fact `~lowBattery` and thus the Interpreter will discard the `energySaver` and will apply the `default` profile that prioritizes usability over cost and energy.

```
1 prices(localPrices) and prices(onlinePrices) and
2 priceShared and position(gpsPosition)
3
4 or
5
6 prices(localPrices) and prices(onlinePrices) and
7 priceShared and position(userDefinedPosition)
```

Listing 3: ShopReview Goal.

```
1 hasFixedFocusCamera and hasGPS and ~isGPSEnabled
```

Listing 4: ShopReview Initial State.

```
1  @Action(name="getProductName")
2  @ReturnValue("name")
3  @QoSProfile(metrics={"usability", "cost", "energy"},
4              values={1,0,-1})
5  public String getProductNameViaSearchUPC(Barcode barcode){
6    String barcodeValue = barcode.getValue();
7    //Invoke http://searchupc.com/
8    String productName = searchupc(barcodeValue);
9    return productName;
10 }
11
12 @Action(name="getProductName")
13 @ReturnValue("name")
14 @QoSProfile(metrics={"usability", "cost", "energy"},
15             values={1,1,-1})
16 public String getProductNameViaSimpleUPC(Barcode barcode){
17   String barcodeValue = barcode.getValue();
18   //Invoke http://simpleupc.com/
19   String productName = simpleupc(barcodeValue);
20   return productName;
21 }
22
23 @Action(name="getProductName")
24 @ReturnValue("name")
25 @QoSProfile(metrics={"usability", "cost", "energy"},
26             values={-1,0,0})
27 public String getProductNameFromUser(Barcode barcode){
28   String barcodeValue = barcode.getValue();
29   //Ask the user for the product name
30   String productName = ...;
31   return productName;
32 }
```

Listing 5: `getProductName` Concrete Actions.

Given these premises, if we consider, for example, the `getProductName` abstract action, with its concrete counterparts reported in Listing 5, and the `default` QoS policy, the Interpreter first selects the `getProductNameViaSearchUPC` and `getProductNameViaSimpleUPC` actions, which have the maximum usability value. Then it applies the second constraint (*i.e.,* minimum cost) selecting only the `getProductNameViaSearchUPC`, the one that is invoked.

Let us consider now a more complex scenario to explain how SelfMotion may consider and satisfy more articulated QoS requirements. Let us consider in particular a scenario where the profiles of concrete actions include an additional QoS profile called *network* that indicates the amount of bandwidth consumed by each concrete action and let us add – on top of the QoS policies reported in Listing 6 – the additional `slowConnection` policy reported in Listing 7.

In this setting, the `slowConnection` policy will be enabled by the Middleware on every device in which the WiFi connection is not available. By relying on this profile, the app designers may prioritize concrete actions that consume less bandwidth when a reliable and fast connection is not available in order to guarantee a smoother user experience. Conversely, if the `slowConnection` profile is disabled (i.e., a WiFi connection is available) the remaining other two QoS profiles that predicate

```
1 qos: energySaver
2 pre: lowBattery
3 min: energy
4 max: usability
5 min: cost
6
7 qos: default
8 pre: true
9 max: usability
10 min: cost
11 min: energy
```

Listing 6: QoS Policy Definitions.

```
1 qos: slowConnection
2 pre: ~hasWiFi or ~wiFiEnabled
3 min: network
4 max: usability
5 min: cost
6 min: energy
```

Listing 7: Additional QoS Policy Definition.

over the battery state are considered for execution.

It is important to notice that the SelfMotion mechanism based on policies and profiles represents a well balanced trade-off among the simplicity and efficiency required by the mobile domain and the expressiveness needed by developers. Indeed, profiles and policies allow application designers to effectively prioritize multiple and conflicting QoS requirements as illustrated in this example in which effective bandwidth management is prioritized over battery management. Summing up, by specifying one or more QoS policies developers encode a hierarchical system of priorities among available concrete actions, which in turn allows an adaptive behavior of the resulting app, as discussed later on in Section 4.2. Finally, it is important to mention that in our previous work Cugola et al. (2012d) we investigated a more comprehensive and expressive approach to QoS for the specific domain of service orchestrations.

3.3. The SelfMotion *middleware*

As previously introduced, the SelfMotion middleware is in charge of executing the app. First of all, at start-up it analyzes the current device and populates the initial state with the set of facts that describe the device's features (*i.e.,* the available sensors, the battery state, etc.). Second, it invokes its two internal components: the *Planner* and the *Interpreter*.

The Planner analyzes the goal, the initial state, and the abstract actions and produces an *Abstract Execution Plan*, which lists the logical steps (*i.e.,* the abstract actions) to reach the goal. The Interpreter, takes this plan and executes it by associating each abstract action with a concrete one, chosen according to the QoS policy that is currently active, invoking external components where specified.

During execution of the plan, the actual state of the app is represented by the abstract objects manipulated by the Planner and by the concrete (*i.e.,* Java) objects manipulated by the Interpreter at run-time. Both are kept by the Interpreter into the *Instance Session*: a key-value database that maps each abstract object used by the Planner and referenced inside the plan with a corresponding concrete object. When the Interpreter must invoke a concrete action to execute the next step of the plan, it uses the Instance Session to retrieve the concrete objects to be passed to the action, while the value returned by the action, if any, is stored into the Instance Session, mapped to the abstract object whose name is given through the `@ReturnValue` annotation (see Listing 5 for an example). This way the abstract plan produced by the Planner is concretely executed by the Interpreter, step by step.

If something goes wrong during this process (*e.g.,* an external service returns an exception), the Interpreter first tries a different concrete action for the abstract action that failed (following the order of precedence established by the QoS policy in use). If no alternative actions can be found or all alternatives have failed, it invokes the Planner again to build an alternative plan that skips the abstract action whose concrete counterparts have all failed. This approach allows SelfMotion to automatically adapt to the situations (and failures) it encounters at run-time, maximizing reliability. All of this occurs without requiring designers to explicitly code complex exception handling strategies. Everything is managed by the SelfMotion middleware, which uses the set of alternative concrete actions associated to the same abstract action as backups of each other, while the Planner is in chrge of automatically determining the sequence of steps that satisfies the goal under the circumstances actually faced at run-time.

As far as the implementation is concerned, the current SelfMotion prototype uses an ad-hoc planner, built as an extension of JavaGP (Meneguzzi and Luck (2009); JavaGP (2010)), a Java open-source version of the Graphplan (Blum and Furst (1997)) planner. In particular, we extended the JavaGP planner to support multiple goals and the possibility of setting the initial state of the plan at run-time. The JavaGP planner was also modified to introduce the ability of inhibiting the use of some steps in the plan, *i.e.,* those that are mapped to concrete actions whose invocation failed at the previous round.

Listing 8 reports a possible plan of the SR example for a device with fixed focus camera (*i.e.,* `hasFixedFocusCamera` is set to true) and with a GPS sensor available but not enabled (*i.e.,* `hasGPS` set to true, `isGPSEnabled` set to false). As mentioned, this plan is a list of abstract actions that lead from the initial state to a state that satisfies the goal, as in Listing 3. Notice that: (1) when several sequences of actions could satisfy the goal the Planner chooses one non-deterministically;[5] (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.

From a deployment point of view, 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. In the first case, plan generation and interpretation take place in the same execution environment, while in the

[5] As a consequence of the use of the Graphplan planning algorithm, the current implementation prioritizes plans with the smaller number of actions

```
1 acquirePhoto
2 blurryReadBarcode(barcodeImage)
3 enableGPS
4 getPositionWithGPS
5 getProductName(productBarcode)
6 inputPrice(name)
7 searchTheWeb(name)
8 searchTheNeighborhood(productBarcode, gpsPosition)
9 sharePrice(name, price)
```

Listing 8: A Possible Abstract Execution Plan.

second case the Planner is deployed on a remote server and the Interpreter invokes it as a service when needed. The two strategies differ in their performance, as we will discuss in Section 5.

4. Advantages of the SelfMotion Approach

This section describes the main advantages of our approach with respect to the development process usually adopted for apps. The discussion refers to the SR example.

4.1. 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 functionalities the app has to provide, ignoring how they will be implemented (*e.g.,* through ad-hoc developed components, invoking external services, or launching third party apps). This choice is delayed to the time when concrete actions are defined. Moreover, if different concrete actions are associated with the same abstract one, the actual choice of how a functionality is implemented is delayed to run-time, when abstract actions are bound to concrete ones. For example, consider the `GetProductName` functionality of the SR app. In the initial phase of the app's design, developers may focus on the features it requires —the precondition— and the features it provides —the post-condition. Later on, they can implement a first prototype of this functionality (a concrete action) that leverages an ad-hoc developed component (*i.e.,* the manual input of the product name) and they may realize that this solution needs to be improved in terms of usability. After this first try, the app may gradually evolve by adding other concrete actions that implement the same functionality, *e.g.,* exploiting a Web service. This approach, that decouples system design from its implementation, is typical of mature engineering domains but it is not currently supported by mainstream apps' development environments. SelfMotion is an attempt to address this issue.

4.2. Enable Transparent Adaptation.

By separating abstract and concrete actions (with their QoS profile) and by supporting one-to-many mappings among abstract and concrete actions we solve two key problems of mobile apps: (1) how to adapt the app to the plethora of devices available today, and (2) how to cope with failures happening at run-time.

As an example of problem (1), consider the implementation of the `GetPosition` functionality given in Listing 1 and compare it with its SelfMotion counterpart, which relies on several abstract actions with different preconditions (see Listing 2). The former requires to explicitly hard-code (using *if-else* constructs) 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 one, given the current device capabilities and the order of preference provided by the app's designers.

As for problem (2), consider the example of the `GetProductName` functionality, which is implemented in SelfMotion by a single abstract action mapped to three different concrete actions (Listings 2 and 5). The middleware initially tries the first concrete action, which invokes an external service. If this returns an exception, the second concrete action is automatically tried. In the unfortunate case this also fails, the third concrete action is tried. Finally, if none of the available concrete actions succeeds, SelfMotion may rely on its *replanning* mechanism to build an *alternative plan* at run-time. As an example, consider the case in which the Interpreter is executing the plan reported in Listing 8 and let us assume that the GPS sensor fails to retrieve the user location (*e.g.,* because we are indoor) and throws a system exception. The middleware automatically catches the exception and recognizes the `getPositionWithGPS` as faulty, which has no alternative concrete actions. In this setting the Planner is invoked to generate a new plan that avoids the faulty step. The new plan will include the `getPositionManually` abstract action, whose concrete counterpart will ask the position to the user through an ad-hoc pop-up. Again, obtaining the same behavior using conventional approaches would require a complex usage of exception handling code, while SelfMotion does everything automatically, relieving programmers from the need of explicitly handling the intertwined exceptional situations that may happen at run-time.

Finally, the possibility of specifying multiple QoS policies also reveals the adaptive nature of SelfMotion apps. Indeed, let us recall the policy example in Listing 6. In the previous section we considered the case of a device with a fully charged battery, which would select the `default` policy. If we consider now the alternative scenario in which `batteryLow` is set true in the initial state, the `energySaver` policy would be selected. This change results in a different behavior of the Interpreter (and consequently a different behavior of the app), which will prioritize the energy efficient actions. As an example, the `GetProductName` functionality this time would be realized by executing the `getProductNameFromUser` concrete action. In other words, through an accurate use of QoS policies, SelfMotion allows developers to easily build apps that adapt to the execution context.

4.3. *Improve Code Quality and Reuse.*

As a final advantage of SelfMotion we observe that by promoting a clean modularization of the app's functionality into a set of abstract actions and their concrete counterparts, and by avoiding convoluted code using cascaded *if-else*s and exception handling constructs, SelfMotion improves readability and maintainability of apps' code.

Moreover, by encapsulating the various features of an app into 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 action can be easily reused across different apps. This advantage is fundamental to shorten the development life-cycle, which is crucial in the mobile domain.

5. Validating the SelfMotion Approach

To validate the SelfMotion approach, we implemented a publicly available open-source tool where the implementation the SR app can also be found (see Section 7). Although our approach is general and applies with limited technological modifications to several existing mobile frameworks, we focused on the Android mobile platform Rogers et al. (2009) for our prototype.

The initial validation we report in this section consists of a testing campaign we performed, exploiting the Android emulator as well as several real mobile devices, to measure the overhead introduced by SelfMotion w.r.t. conventional approaches. The experiments showed that this overhead exists but it is practically negligible. More specifically, we measured how the plan generation step performed at run-time by the Planner represents the major element of overhead and the potential bottleneck of SelfMotion. The time to execute this step depends on two factors: (1) the plan length, and (2) the number of abstract actions in the domain, while it is not affected by the number of available concrete actions, as the binding between concrete and abstract actions is performed separately, by the Interpreter. As far as this aspect is concerned, we measured that it does not add a measurable overhead to the overall running time.

Before showing the results we obtained, we describe the testing platforms we chose. For the experiments involving a local deployment of the Planner we used two different hardware settings: an *LG Nexus 4*, which represents the typical Android-enabled device available today, and a netbook equipped with 2*GB* of RAM, an Atom processor, Ubuntu Linux 32 bit, and OpenJDK 1.6.0 The latter represents next generation Android devices (*e.g.,* the *Lava Xolo X900*) powered by the new Intel SOC for smartphones, which integrates the same Atom CPU. For the experiments involving a remote deployment, we installed the Planner on a remote server equipped with an *AMD Phenom II X6 1055T* processor, 8*GB* of RAM, Ubuntu Linux 64 bit, and Sun Java Virtual Machine 1.7.0. Moreover, we repeated all experiments discussed hereafter at least thirty times, varying the seeds to generate the workload, for each described scenario. The figures shown below provide the average results we obtained, report through error bars the 95% confidence interval, and indicate their interpolation with a second grade polynomial trend line.

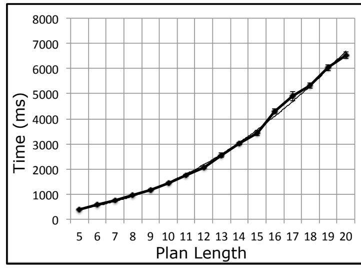
Moving from the consideration above, we started analyzing how the plan length impacts performance. In particular, we developed a scenario in which we had fifty abstract actions and a goal definition satisfiable through a plan composed of five of these actions. We measured the time needed to obtain the plan and we repeated the experiment changing the goal definition in order to obtain plans of increasing length—from five to twenty—recording the time needed to compute them, both with a local and with a remote deployment of the Planner. Figure 2(a) shows that, by running this testbed with a local Planner and with an initial plan composed by five actions, the Planner takes around 385*ms* to complete. The time needed to generate the plan gradually increases up to 6530*ms* for a plan composed by twenty actions.

Figure 2(b) shows instead how the Atom-based platform provides improved performance, reducing the times by an order of magnitude. Finally, if we choose to rely on a remote execution, the plan generation time decreases of another order of magnitude, as reported in Figure 2(c). Notice that the results we report for the remote case—here and in the following experiments—do not include the time required to invoke the Planner remotely, as the time to traverse the network strongly depends on the actual connection type of the device (*e.g.,* gprs vs. WiFi), and the characteristics of the deployment in general.

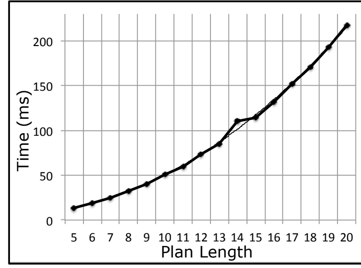
Our second test set focuses on the impact of the number of abstract actions on the plan generation time. For this we built a scenario in which there is an increasingly large set of abstract actions and a goal definition that generates a plan using ten of them. Figure 3(a) shows that, with twenty abstract actions and a local deployment on the LG Nexus 4, the SelfMotion Planner takes about 1136*ms* to complete. This time gradually increases up to 1778*ms* when eighty abstract actions are available. As in the previous scenario, the Atom platform and the remote deployment provide further advantages, as reported in Figure 3(b) and 3(c).

In general these results show an acceptable overhead even on today's devices: an overhead that should not affect the overall app usability. This is especially true if we consider that loading a typical mobile app on today's devices may require one or more seconds —not milliseconds— and executing it requires tens of seconds. Moreover, our implementation, albeit efficient, is just a prototype, and significant performance improvements may be achieved by introducing ad-hoc features, such as plan caching. Finally, we observe that our experiments considered plans of length up to twenty and up to eighty abstract actions. These are overestimates of the values we may encounter on real apps, which are typically characterized by a limited number of abstract functionality as shown by the example described in Section 2. Indeed, the plan length of the SR app includes eight or nine abstract actions (depending on the device capabilities) and the Planner generates the most complex of these plans in 214*ms* (LG Nexus 4), 114*ms* (Atom), and 32*ms* (remote execution).

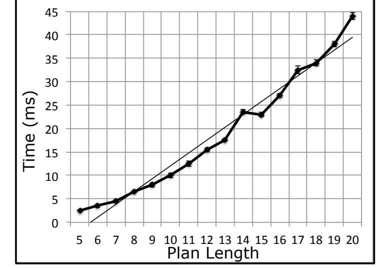
Finally, we briefly report some considerations on local ver-



(a) LG Nexus 4.

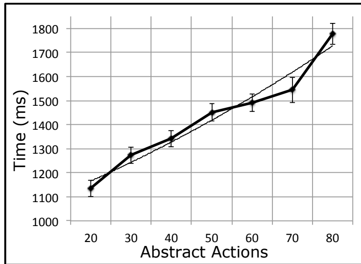


(b) Atom Platform.

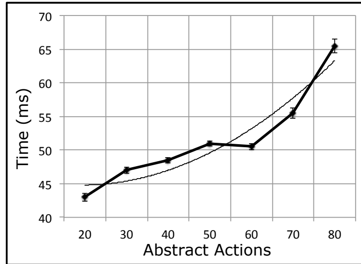


(c) Remote Evaluation.

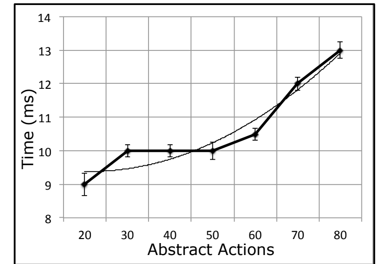
Figure 2: Plan Generation Time over Plan Length.



(a) LG Nexus 4.



(b) Atom Platform.



(c) Remote Evaluation.

Figure 3: Plan Generation Time over Abstract Actions.

sus remote plan generation. The choice among them essentially depends on: (1) the number of abstract actions —which represents an upper-bound of the plan length— and (2) the computational capability of the device. The more powerful a device is, the larger the set of abstract actions it is able to handle successfully in a reasonable time. Since the computational power is known only at run-time, the decision between local or remote plan generation cannot be made statically but it has to be delayed to execution time. Clearly, a local plan generation is generally preferable, since it allows the app to execute successfully even if the device is not connected to the Internet. Notice that SelfMotion is adaptive even in choosing between these two alternatives, which are affected by the device on which the prototype actually runs. Indeed, at design-time, given the set of abstract actions available, SelfMotion estimates the length of the plan. Depending on this value, at run-time, knowing the characteristics of the device where it is running, the middleware autonomously decides whether the plan generation must be performed locally or remotely.

6. Related Work

The recent massive adoption of mobile devices generated an increasing interest on engineering mobile applications. A lot of research is focusing on the effective and efficient development of such systems, as summarized by Dehlinger and Dixon (2011) and Wasserman (2010). Existing works span a wide range of approaches: from how to achieve context-aware behavior (*e.g.,* Gonzlez et al. (2011)) to how to apply agile methods in the mobile domain (*e.g.,* Abrahamsson et al. (2004)).

6.1. Context-aware Frameworks

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) (Hirschfeld et al. (2008)). For example, Subjective-C (Gonzlez et al. (2011)) provides context-oriented abstractions on top of Objective-C, a mainstream language used for programming iOS applications. The EgoSpaces middleware (Julien and Roman (2006)) can be used to provide context information extracted from data-rich environments to applications. Another approach to mobile computing middleware is presented in Capra et al. (2003), 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 Appeltauer et al. (2008); van Wissen et al. (2010) provide specific context-aware extensions to the Android platform.

From our point of view, 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 is populated with information related to 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.

6.2. Multi-platform Frameworks

Other existing related approaches (*e.g.,* Ohrt and Turau (2012)) provide solutions for multi-platform app development. Approaches like PhoneGap (2012) and Appcelerator (2012) allow developers to code using standard technologies (*e.g.,* Javascript and HTML5) and deploy the same codebase on several platforms, including as iOS or Android. These frameworks have a great potential but at the same time they currently suffer from the same limitations as traditional app development, such as the intertwined business logic with adaptation code and limited support for code maintainability.

None of the above efforts specifically deals with service-oriented mobile applications, which instead represent a significant portion of the apps developed so far. The work by Chakraborty et al. (2005) describes an approach for service composition in mobile environments and evaluates criteria for judging protocols that enable such composition. They mainly concentrate on a distributed architecture that facilitates service composition and do not focus on the application layer nor on its adaptation capabilities, as instead SelfMotion does. Generally speaking, the existing approaches to service-oriented mobile app on mobile environments focus on enabling the service composition, without considering the associated consequences, such as the need of adaptation as motivated in Section 1.

6.3. Service Compositions

SelfMotion models mobile applications as composition of ad-hoc developed components and remotely invoked services. Indeed, SelfMotion brings BPEL-like service orchestration to mobile app development, allowing the app programmers to define high-level processes (abstract actions and goals) separately from low-level details (concrete actions), making it easier for programmers to compose apps by combining concrete actions in a declarative way. From this viewpoint SelfMotion shares foundational concepts with traditional service compositions in which applications are designed and implemented by combining the functionality of external services provided by third-party organizations (Erl (2005)). For this reason it is important to relate our approach even with existing solutions in this area as discussed hereafter. During the last years, various proposals have been made to reduce the complexity inherent in defining service compositions, with the goal of further increasing the diffusion of this technology. As an alternative to traditional languages for service compositions such as BPEL (Alves et al. (2006)) and BPMN (White (2008)), other languages like JOpera (Pautasso and Alonso (2005)), Jolie Montesi et al. (2007), and Orc Kitchin et al. (2009), were proposed. While easier to use and often more expressive than BPEL and BPMN, they do not depart from the imperative paradigm, and consequently they share with them the same limitations that motivated our work.

The complexity in defining Web service compositions is also being tackled through *Automated Service Composition* (ASC) approaches. While our research is motivated by the desire of providing abstractions for the development of adaptive applications, overcoming the limitations of mainstream languages in terms of flexibility and adaptability to unexpected situations, ASC is grounded on the idea that the main problem behind service composition is given by the complexity in selecting the right services in the open and large scale Internet environment. The envisioned solution is to provide automatic mechanisms to select the right services to compose, usually based on a precise description of the semantics of the services available. For example, in Rao et al. (2006), user requirements and Web services are both described in DAML-S (Burstein et al. (2002)), a semantic Web service language, and linear logic programming is used to automatically select the correct services and generate a BPEL or DAML-S process that represents the composite service. Similarly, McIlraith and Son (2002) presents an extension of Golog, a logic programming language for dynamic domains, to compose and execute services described in DAML-S, based on high-level goals defined by users. Both approaches requires the exact semantics of services to be defined formally (*e.g.,* in DAML-S) and they do not support dynamic redefinition of the orchestration at run-time to cope with unexpected situations.

Similar considerations hold for those ASC proposals that adopt planning techniques similar to those adopted in SelfMotion. In these approaches the planning domain is composed by the semantically described services and goals are defined by end-users. For example, Wu et al. (2003) uses the SHOP2 planner to build compositions of services described in DAML-S. Similarly, Bertoli et al. (2010) proposes an algorithm, based on planning via model-checking, that takes an abstract BPEL process, a composition requirement and a set of Web services also described in BPEL and produces a concrete BPEL process with the actual services to be invoked. In SWORD (Ponnekanti and Fox (2002)), the to-be composed services are described in terms of their inputs and outputs, creating the "service model". To build a new service the developer should specify its input and output, which SWORD use to decide which services should be chosen and how to combine them. XSRL, a language to express service requests, is presented in Lazovik et al. (2006). Users can use this language to specify how services should be chosen for a given request. A planner is responsible for choosing the services based on the specified request, augmenting an abstract BPEL process with the selected services.

Other ASC approaches start from an abstract "template process", expressed either in BPEL, e.g., Ardagna and Pernici (2007); Aggarwal et al. (2004), or as a Statechart, *e.g.,* Zeng et al. (2004) and, taking into consideration QoS constraints and end-user preferences, select the best services among those available to be actually invoked. As mentioned in the introduction, these approaches focus on a relatively simpler problem than SelfMotion , as they focus on "selecting the right services at run-time", leaving to the service architect the (complex) task of defining the abstract "workflow" to follow. Moreover, as they use traditional, procedural languages as the tool to model this abstract workflow, they suffer from the limitations and problems that we identified in Section 1. In addition, most of the ASC approaches proposed so far operate before the orchestration starts, while SelfMotion includes advanced mechanisms to automatically adapt the app to the situations encountered at execution time. This is particularly evident if we consider

the problem of compensating actions to undo some already performed steps before following a different workflow that could bypass something unexpected. A problem that, to the beast of our knowledge, is not considered by any of these approaches.

A quantitative comparison among our approach based on planning and declarative languages with respect to the existing solutions in the domain of service compositions can be found in Cugola et al. (2012a).

6.4. Declarative Frameworks

To overcome the limitations of imperative solutions, other researchers followed the idea of adopting a declarative approach. Among those proposals, Declare (Montali et al. (2010); van der Aalst and Pesic (2006)) is the closest to our work. In Declare service compositions are defined as a set of actions and the constraints that relate them. Both actions and constraints are modeled graphically, while constraints have a formal semantics given in Linear Temporal Logic (LTL). There are several differences between Declare and SelfMotion . First of all, Declare focuses on modeling service choreographies to support verification and monitoring. Although it could also be applied to mobile applications, our focus is not restricted to modeling mobile applications but specifically on enacting them. This difference motivates the adoption of LTL as the basic modeling tool, as it enables powerful verification mechanisms but introduces an overhead that can be prohibitive for an enactment tool (Montali et al. (2010)), in particular for the mobile domain. The SelfMotion approach to modeling offers less opportunities for verification but it can lead to an efficient enactment tool. Secondly, SelfMotion emphasizes re-planning at run-time as a mechanism to support self-adaptive applications that maximize reliability even in presence of unexpected failures and changes in the external services. This is an issue largely neglected by Declare, as it focuses on specification and verification and it does not offer specific mechanisms to manage failures at run-time.

GO-BPMN (Greenwood and Rimassa (2007); Burmeister et al. (2008); Calisti and Greenwood (2008)) is another declarative language, designed as a Goal-Oriented extension for traditional BPMN. In GO-BPMN business processes are defined as a hierarchy of goals and sub-goals. Multiple BPMN plans are attached to the "leaf" goals. When executed, they achieve the associated goal. These plans can be alternative or they can be explicitly associated with specific conditions through guard expressions based on the context of execution. Although this approach also tries to separate the declarative statements from the way they can be accomplished, the alternative plans to achieve a goal must be explicitly designed by the service architect and are explicitly attached to their goals. The engine does not automatically decide how the plans are built or replaced; it just chooses between the given options for each specific goal, and it does so at service invocation time. The SelfMotion ability to build the plan dynamically and to rebuild it if something goes wrong at run-time, improves self-adaptability to unexpected situations.

The approach described in Van Riemsdijk and Wirsing (2007) defines a goal-oriented service composition language inspired by agent programming languages, like AgentSpeak(L) (Rao (1996)). One of the main motivations of this approach is the possibility of following different plans of execution in the presence of failures. The main difference with our approach is that the alternative plans need to be explicitly programmed based on the data stored into the Knowledge Base and the programmer needs to explicitly reason about all the possible alternatives and how they are related, in a way similar to that adopted by traditional approaches. In the presence of faults, the facts that compose the Knowledge Base are programmatically updated to trigger the execution of specific steps that have to be specified in advance to cope with that situation. No automatic re-planning is supported.

6.5. Other Relevant Related Work

We observe that the three-layered architectural model for self-management described by Kramer and Magee (2007); Sykes et al. (2008) was also used as an inspiration for SelfMotion language and its middleware. In particular, the layers defined by this architecture are: the *goal management layer*, which is based on model checking from the domain model and goals for the generation of plans (in our approach, the Planner); the *change management layer*, which is concerned with using the generated plans to construct component configurations and direct their operation to achieve the goal addressed by the plan (in our approach, the SelfMotion Interpreter, which interacts with the Planner and executes the generated plan); at last, the *component layer*, which includes the domain specific components (in our approach, the abstract and concrete actions, used to build and enact the plan). SelfMotion inherits from these works, but differs in the way adaptation is achieved (via abstract and concrete actions) and in the focus on the openness required by mobile applications.

7. Conclusions and Future Work

We presented SelfMotion, a declarative approach supporting systematic development of mobile apps, modeled in terms of goals, abstract and concrete actions. The approach exploits automatic planning techniques to elaborate, at run-time, the best sequence of activities to achieve the app's goal. In addition, the proposed approach also allows to annotate the actions that compose the final apps with a description of their non-functional behavior (*i.e.,* their QoS profile). By exploiting such annotation, it is possible for engineers to express various QoS policies that maximize or minimize certain QoS metrics (*e.g.,* the energy consumption) depending on the actual conditions encountered at run-time.

The paper contributes to the research in adaptive software systems and services in two principal ways. First, it investigates a declarative approach for the effective and efficient development of adaptive apps conceived as hybrid compositions of services and components. Secondly, it provides a fully functional middleware, which supports adaptivity and enforces a decoupling of the business logic from the adaptation logic, facilitating code reuse, refactoring, and code evolution.

To demonstrate the advantages of SelfMotion in terms of: (1) ease of use, (2) adaptation capabilities, and (3) quality of

the resulting code, we used the proposed approach to implement a realistic mobile app inspired by an existing worldwide distributed mobile application. In addition, we assessed the overhead introduced by the approach and its scalability by performing a validation campaign, which demonstrated the applicability of the approach.

To encourage the adoption of the proposed approach and to allow the replication of experiments, the SelfMotion implementation has been released as an open-source tool for the Android platform, publicly available[6].

SelfMotion is part of a long running research stream, which aims at investigating declarative approaches to enforce adaptive capabilities in software systems addressing specific domains that span services (*e.g.,* Cugola et al. (2011, 2012a,d)), mobile apps (*e.g.,* Ghezzi et al. (2013b)), and the interaction among the two (*e.g.,* Ghezzi et al. (2013a)). 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 its performance and robustness. Finally, our future work also includes a user study aimed at demonstrating the SelfMotionś usability, its moderate learning curve, and the improvements it brings in terms of productivity.

Acknowledgments

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

References

Abrahamsson, P., Hanhineva, A., Hulkko, H., Ihme, T., Jäälinoja, J., Korkala, M., Koskela, J., Kyllönen, P., Salo, O., 2004. Mobile-D: An Agile Approach for Mobile Application Development. In: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. OOPSLA '04. ACM, New York, NY, USA.
URL http://doi.acm.org/10.1145/1028664.1028736

Aggarwal, R., Verma, K., Miller, J., Milnor, W., 2004. Constraint Driven Web Service Composition in METEOR-S. In: Proceedings of the 2004 IEEE International Conference on Services Computing. SCC '04. IEEE Computer Society, Washington, DC, USA, pp. 23–30.
URL http://dl.acm.org/citation.cfm?id=1025130.1026125

Alves, A., Arkin, A., Askary, S., Bloch, B., Curbera, F., Goland, Y., Kartha, N., Liu, C. K., Konig, D., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P., Yiu, A., eds., 2006. Web Services Business Process Execution Language Version 2.0. Tech. rep., OASIS.
URL http://www.oasis-open.org/apps/org/workgroup/wsbpel/

Appcelerator, 2012. Appcelerator — Titanium Mobile Development Platform.
URL http://www.appcelerator.com/

Appeltauer, M., Hirschfeld, R., Rho, T., 2008. Dedicated programming support for context-aware ubiquitous applications. In: Proceedings of the 2008 The Second International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies. UBICOMM '08. IEEE Computer Society, Washington, DC, USA, pp. 38–43.
URL http://dx.doi.org/10.1109/UBICOMM.2008.56

[6]http://www.dsol-lang.net/self-motion.html

Ardagna, D., Pernici, B., 2007. Adaptive service composition in flexible processes. IEEE Transactions on Software Engineering 33, 369–384.
URL http://doi.ieeecomputersociety.org/10.1109/TSE.2007.1011

Bertoli, P., Pistore, M., Traverso, P., Mar. 2010. Automated composition of web services via planning in asynchronous domains. Artif. Intell. 174 (3-4), 316–361.
URL http://dx.doi.org/10.1016/j.artint.2009.12.002

Blum, A. L., Furst, M. L., Feb. 1997. Fast planning through planning graph analysis. Artif. Intell. 90 (1-2), 281–300.
URL http://dx.doi.org/10.1016/S0004-3702(96)00047-1

Burmeister, B., Arnold, M., Copaciu, F., Rimassa, G., 2008. Bdi-agents for agile goal-oriented business processes. In: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: industrial track. AAMAS '08. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 37–44.
URL http://dl.acm.org/citation.cfm?id=1402795.1402803

Burstein, M. H., Hobbs, J. R., Lassila, O., Martin, D., McDermott, D. V., McIlraith, S. A., Narayanan, S., Paolucci, M., Payne, T. R., Sycara, K. P., 2002. Daml-s: Web service description for the semantic web. In: Proceedings of the First International Semantic Web Conference on The Semantic Web. ISWC '02. Springer-Verlag, London, UK, UK, pp. 348–363.
URL http://dl.acm.org/citation.cfm?id=646996.711291

Calisti, M., Greenwood, D., 2008. Goal-oriented autonomic process modeling and execution for next generation networks. In: van der Meer, S., Burgess, M., Denazis, S. (Eds.), Modelling Autonomic Communications Environments. Vol. 5276 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 38–49, 10.1007/978-3-540-87355-6_4.
URL http://dx.doi.org/10.1007/978-3-540-87355-6_4

Capra, L., Emmerich, W., Mascolo, C., 2003. Carisma: Context-aware reflective middleware system for mobile applications. IEEE Transactions on Software Engineering 29, 929–945.
URL http://doi.ieeecomputersociety.org/10.1109/TSE.2003.1237173

Chakraborty, D., Joshi, A., Finin, T., Yesha, Y., Aug. 2005. Service composition for mobile environments. Mob. Netw. Appl. 10 (4), 435–451.
URL http://dx.doi.org/10.1145/1160162.1160168

Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Mller, H., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J., 2009. Software engineering for self-adaptive systems: A research roadmap. In: Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (Eds.), Software Engineering for Self-Adaptive Systems. Vol. 5525 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 1–26, 10.1007/978-3-642-02161-9_1.
URL http://dx.doi.org/10.1007/978-3-642-02161-9_1

Cugola, G., Ghezzi, C., Pinto, L., 2012a. Dsol: a declarative approach to self-adaptive service orchestrations. Computing 94, 579–617, 10.1007/s00607-012-0194-z.
URL http://dx.doi.org/10.1007/s00607-012-0194-z

Cugola, G., Ghezzi, C., Pinto, L. S., 2011. Process programming in the service age: Old problems and new challenges. In: Tarr, P. L., Wolf, A. L. (Eds.), Engineering of Software. Springer Berlin Heidelberg, pp. 163–177, 10.1007/978-3-642-19823-6_10.
URL http://dx.doi.org/10.1007/978-3-642-19823-6_10

Cugola, G., Ghezzi, C., Pinto, L. S., Tamburrelli, G., 2012b. Adaptive service-oriented mobile applications: A declarative approach. In: Service-Oriented Computing. Springer, pp. 607–614.

Cugola, G., Ghezzi, C., Pinto, L. S., Tamburrelli, G., 2012c. Selfmotion: a declarative language for adaptive service-oriented mobile apps. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. ACM, p. 7.

Cugola, G., Pinto, L. S., Tamburrelli, G., 2012d. Qos-aware adaptive service orchestrations. In: Web Services (ICWS), 2012 IEEE 19th International Conference on. IEEE, pp. 440–447.

Dehlinger, J., Dixon, J., 2011. Mobile application software engineering: Challenges and research directions. In: Workshop on Mobile Software Engineering.
URL http://www.mobileseworkshop.org/papers/7-Dehlinger_Dixon.pdf

Erl, T., 2005. Service-oriented architecture: concepts, technology, and design. The Prentice Hall Service-Oriented Computing Series from Thomas Erl Series. Prentice Hall Professional Technical Reference.

Ghezzi, C., Pezzè, M., Tamburrelli, G., 2013a. Improving interaction with services via probabilistic piggybacking. In: Service-Oriented Computing. Springer.

Ghezzi, C., Pinto, L. S., Spoletini, P., Tamburrelli, G., 2013b. Uncertainty management via model-driven adaptivity. In: Proceedings of the 35rd International Conference on Software Engineering. ACM.

Gonzlez, S., Cardozo, N., Mens, K., Cdiz, A., Libbrecht, J.-C., Goffaux, J., 2011. Subjective-c. In: Malloy, B., Staab, S., van den Brand, M. (Eds.), Software Language Engineering. Vol. 6563 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 246–265, 10.1007/978-3-642-19440-5_15.
URL http://dx.doi.org/10.1007/978-3-642-19440-5_15

Greenwood, D., Rimassa, G., 2007. Autonomic goal-oriented business process management. Autonomic and Autonomous Systems, International Conference on 0, 43.
URL http://doi.ieeecomputersociety.org/10.1109/CONIELECOMP.2007.61

Hirschfeld, R., Costanza, P., Nierstrasz, O., Mar. 2008. Context-oriented programming. Journal of Object Technology 7 (3), 125–151.
URL http://www.jot.fm/contents/issue_2008_03/article4.html

JavaGP, 2010. Java GraphPlan.
URL http://emplan.sourceforge.net

Julien, C., Roman, G.-C., 2006. Egospaces: Facilitating rapid development of context-aware mobile applications. IEEE Transactions on Software Engineering 32, 281–298.
URL http://doi.ieeecomputersociety.org/10.1109/TSE.2006.47

Kitchin, D., Quark, A., Cook, W., Misra, J., 2009. The orc programming language. In: Proceedings of the Joint 11th IFIP WG 6.1 International Conference FMOODS '09 and 29th IFIP WG 6.1 International Conference FORTE '09 on Formal Techniques for Distributed Systems. FMOODS '09/FORTE '09. Springer-Verlag, Berlin, Heidelberg, pp. 1–25.
URL http://dx.doi.org/10.1007/978-3-642-02138-1_1

Kramer, J., Magee, J., 2007. Self-managed systems: an architectural challenge. In: 2007 Future of Software Engineering. FOSE '07. IEEE Computer Society, Washington, DC, USA, pp. 259–268.
URL http://dx.doi.org/10.1109/FOSE.2007.19

Lazovik, A., Aiello, M., Papazoglou, M., Jun. 2006. Planning and monitoring the execution of web service requests. Int. J. Digit. Libr. 6 (3), 235–246.
URL http://dx.doi.org/10.1007/s00799-006-0002-5

McIlraith, S. A., Son, T. C., 2002. Adapting golog for composition of semantic web services. In: Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR-02). pp. 482–496.

McKinley, P. K., Sadjadi, S. M., Kasten, E. P., Cheng, B. H. C., Jul. 2004. Composing adaptive software. Computer 37 (7), 56–64.
URL http://dx.doi.org/10.1109/MC.2004.48

Meneguzzi, F., Luck, M., 2009. Declarative agent languages and technologies vi. Springer-Verlag, Berlin, Heidelberg, Ch. Leveraging New Plans in AgentSpeak(PL), pp. 111–127.
URL http://dx.doi.org/10.1007/978-3-540-93920-7_8

Montali, M., Pesic, M., Aalst, W. M. P. v. d., Chesani, F., Mello, P., Storari, S., Jan. 2010. Declarative specification and verification of service choreographiess. ACM Trans. Web 4 (1), 3:1–3:62.
URL http://doi.acm.org/10.1145/1658373.1658376

Montesi, F., Guidi, C., Lucchi, R., Zavattaro, G., Jun. 2007. Jolie: a java orchestration language interpreter engine. Electron. Notes Theor. Comput. Sci.
URL http://dx.doi.org/10.1016/j.entcs.2007.01.051

Ohrt, J., Turau, V., 2012. Cross-platform development tools for smartphone applications. Computer 99 (PrePrints).
URL http://doi.ieeecomputersociety.org/10.1109/MC.2012.121

Pautasso, C., Alonso, G., Jan. 2005. Jopera: A toolkit for efficient visual composition of web services. Int. J. Electron. Commerce 9 (2), 107–141.
URL http://dl.acm.org/citation.cfm?id=1278095.1278101

PhoneGap, 2012. PhoneGap.
URL http://www.phonegap.com/

Ponnekanti, S. R., Fox, A., 2002. SWORD: A developer toolkit for web service composition. In: Proceedings of the 11th International WWW Conference (WWW2002). Honolulu, HI, USA.

Rao, A. S., 1996. Agentspeak(l): Bdi agents speak out in a logical computable language. In: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away: agents breaking away. MAAMAW '96. Springer-Verlag New York, Inc.
URL http://dl.acm.org/citation.cfm?id=237945.237953

Rao, J., Küngas, P., Matskin, M., Jun. 2006. Composition of semantic web services using linear logic theorem proving. Inf. Syst. 31 (4), 340–360.
URL http://dx.doi.org/10.1016/j.is.2005.02.005

Rogers, R., Lombardo, J., Mednieks, Z., Meike, G., 2009. Android Application Development: Programming with the Google SDK. Oreilly Series. O'Reilly Media.

Sykes, D., Heaven, W., Magee, J., Kramer, J., 2008. From goals to components: a combined approach to self-management. In: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems. SEAMS '08. ACM, New York, NY, USA, pp. 1–8.
URL http://doi.acm.org/10.1145/1370018.1370020

van der Aalst, W. M. P., Pesic, M., 2006. Decserflow: towards a truly declarative service flow language. In: Proceedings of the Third international conference on Web Services and Formal Methods. WS-FM'06. Springer-Verlag, Berlin, Heidelberg, pp. 1–23.
URL http://dx.doi.org/10.1007/11841197_1

Van Riemsdijk, M. B., Wirsing, M., 2007. Using goals for flexible service orchestration: a first step. In: Proceedings of the 2007 AAMAS international workshop and SOCASE 2007 conference on Service-oriented computing: agents, semantics, and engineering. AAMAS'07/SOCASE'07. Springer-Verlag, Berlin, Heidelberg, pp. 31–48.
URL http://dl.acm.org/citation.cfm?id=1768363.1768366

van Wissen, B., Palmer, N., Kemp, R., Kielmann, T., Bal, H., Nov. 2010. ContextDroid: an expression-based context framework for Android. In: Proceedings of PhoneSense 2010.
URL http://sensorlab.cs.dartmouth.edu/phonesense/papers/Wissen-ContextDroid.pdf

Wasserman, A. I., 2010. Software engineering issues for mobile application development. In: Proceedings of the FSE/SDP workshop on Future of software engineering research. FoSER '10. ACM, New York, NY, USA, pp. 397–400.
URL http://doi.acm.org/10.1145/1882362.1882443

White, S. A., 2008. Business Process Modeling Notation, V1.1.
URL http://www.bpmn.org/Documents/BPMN_1-1_Specification.pdf

Wu, D., Parsia, B., Sirin, E., Hendler, J., Nau, D., 2003. Automating daml-s web services composition using shop2. In: Fensel, D., Sycara, K., Mylopoulos, J. (Eds.), The Semantic Web - ISWC 2003. Vol. 2870 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 195–210, 10.1007/978-3-540-39718-2_13.
URL http://dx.doi.org/10.1007/978-3-540-39718-2_13

Zeng, L., Benatallah, B., Ngu, A. H., Dumas, M., Kalagnanam, J., Chang, H., 2004. Qos-aware middleware for web services composition. IEEE Transactions on Software Engineering 30, 311–327.
URL http://doi.ieeecomputersociety.org/10.1109/TSE.2004.11