Politecnico di Milano
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

A Declarative Approach to Self-Adaptive Service Orchestrations

Doctoral Dissertation of:
Leandro Sales Holanda Pinto

Advisor:
Prof. Gianpaolo Cugola
Coadvisor:
Prof. Carlo Ghezzi
Tutor:
Prof. Barbara Pernici
Supervisor of the Doctoral Program:
Prof. Carlo Fiorini

2012 – XXV

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32   I-20133 — Milano

TO LENINA

Acknowledgements

First of all, I would like to thank, with all my heart, my wife, Lenina, for supporting me and sharing with me those three years full of new experiences. I am sure we will remember them with affection. *"It's something unpredictable but in the end it's right, I hope you've had the time of your life."*

Then I want to thank my family back home in Brazil. Although the distance they were always present supporting and cheering for us.

I want also to thank my Advisor, Prof. Gianpaolo Cugola for his patience and support during those three years, his office's door was always open, no matter what. My sincere thanks also go to Prof. Carlo Ghezzi, he was the one who gave me the opportunity of being here in the first place. I really appreciate that. I hope to see you guiding bright minds still for a long time. Your passion and enthusiasm are remarkable. I have probably said this before, but the title of PhD is the less important thing I will carry on with me. How much I have learned with both of you is not something one can measure with a title.

A special thanks to my closest friends, Alessandro Margara, Andrea Ciancone, Giordano Tamburrelli, Antonio Filieri, Marinho Sangiorgio and Paulo Nardi. Somehow, you made me feel like home.

I cannot forget all my colleagues and officemates, Nicolò Calcavecchia, Liliana Pasquale, Marco Funaro, Andrea Mocci, Alessandro Sivieri, Marcello Bersani, Daniel Dubois, Santo Lombardo, Amir Shariﬂoo and Luca Cavallaro. I spent the most significant moments of this journey with them.

I am also grateful to the friends I made while visiting Georgia Tech. First big thanks to Prof. Alex Orso.[1] Your dedication and commitment to research (and to your students) are inspiring. Then to Paola Spoletini, Shauvik Roy Choudhary and Mattia Fazzini.

Finally, I would like to thank my friends (first and more frequent guests) Nabor Mendonça and Andréia Formico for the advices and the support from the very beginning.

Agradeço a todos do fundo do coração!!

[1] *"You are a funny guy"*

Abstract

Most modern software systems have a decentralized, modular, distributed, and dynamic structure. They are often composed of heterogeneous components and operate on several different infrastructures. They are increasingly built by composing *services*; that is, components owned (designed, deployed, maintained, and run) by remote and independent stakeholders. The quality of service perceived by the clients of such a composite application depends directly on the individual services that are integrated in it, but also on the way they are composed. At the same time, the world in which such applications are situated (in particular, the remote services upon which they can rely) change continuously. These requirements ask for an ability of applications to self-adapt to dynamic changes, especially when they need to run for a long time without interruption. This, in turn, has an impact on the way service compositions are implemented using ad-hoc process languages defined to support compositions.

Indeed, the principles behind *Service Oriented Computing* (SOC) has brought a simplification in the way distributed applications are developed. We claim, however, that the fully potential of SOC is still to be reached and it is our belief that this is caused by the same tools used to build such service compositions. Indeed, mainstream approaches failed to support dynamic, self-managed compositions, characteristics they must have due to the changeable world they are embedded in, where it is essential to be able to adapt to changes that may happen at run-time. Unfortunately, mainstream SOC languages make it quite hard to develop such kind of self-adapting compositions. It is specially hard because they too closely resemble traditional languages, with their imperative style of programming: it requires service architects to take care of an intricate control flow, in which one tries to capture all possible ways things can go wrong and react properly to exceptional conditions. A great effort is required to program the application to continue to meet its requirements in the presence of anticipated or unanticipated changes.

In this thesis we present DSOL - *Declarative Service Orchestration Language*, a novel approach in which we abandon the imperative style adopted by currently available languages in favor of a strongly declarative alternative. DSOL allows an orchestration to be modeled by giving

a high-level description of the elementary activities, a loosely coupled implementation layer and the overall *Goal* to be met by the orchestration.

DSOL models are then executed by an ad-hoc service orchestration engine, which leverages automatic planning techniques to elaborate, at run-time, the best sequence of activities to achieve the goal. Whenever a change happens in the external environment, which prevents execution to be completed, DSOL engine behaves in a self-healing manner. Through dynamic re-planning and advanced re-binding mechanisms, it finds an alternative path toward the goal and continues executing it, until the goal of the orchestration is reached.

DSOL provides several advantages w.r.t. traditional approaches, ranging from a better support for the implementation of self-adaptive orchestrations to a more decoupled architecture in which they may smoothly evolve to cope with exceptional situations or to requirement changes. DSOL models also promotes a more readable and maintainable code through a clear separation of concerns, in which the orchestration logic is isolated from the logic to used to handle and adapt to exceptional situations.

Contents

List of Figures

List of Tables

1 Introduction

Software is the driving engine of modern society. Most human activities are either software enabled or entirely managed by software. Examples range from health-care and transportation to commerce and manufacturing to entertainment and education. We are today in a stage where existing Web technology allows information (*i.e.,* data) to be accessed from everywhere, being it computers or mobile devices, through the network. From this situation, we are moving to a stage where also functionalities (*i.e.,* services) may be openly accessed and integrated to provide new functionality and serve different users.

Although the terms *(software) service* and *service-oriented computing (SOC)* are becoming widely used, they should be made more precise to better understand the nature of the problems we are currently facing. A service is a software component that provides some functionality of possible general use. Functionalities can be of different levels of complexity, generality, and granularity. Services and SOC differ with respect to *(software) components* and *component-based computing* in that services are owned (developed, deployed, run, and maintained) by independent stakeholders, who make them available for external use by multiple potential clients. Accordingly, services are designed to support interoperable software-to-software (potentially written in different programming languages) interaction over the network.

When first proposed, the principles behind *Service Oriented Architectures (SOAs)* and the *Service Oriented Computing (SOC)* paradigm held the promise to solve the complexity behind programming large-scale, distributed applications. By *orchestrating* [1] existing services through easy to use languages, even non-technical users were promised to be empowered with the ability of creating their own added-value offerings. Unfortunately, after some years of research, technological developments, and experience, we are still far from reaching these goals: "service orchestration" is still a difficult and error-prone art, which requires sophisticated skills.

The main source of complexity hampering a wider adoption of SOC has to do with the fact that service orchestrations live in a very unstable world, in which changes occur continuously and unpredictably [2]. Indeed, if not adequately managed, these changes inevitably lead to failures. As an example, orchestrations may fail because the target services they invoke have been discontinued by their providers, because they became unreachable through the currently used interconnection network, or because new versions have been deployed, which are incompatible with the previous ones. It is therefore fundamental that the orchestration could respond and adapt to such kinds of changes, finding an alternative strategy to achieve its goal.

Defining and managing service orchestrations in an open and evolving environment, however, is hard. It is specially hard using traditional "programming" approaches adopted by mainstream orchestration languages, like BPEL [3] and BPMN [4]. Indeed, we strongly argue that the main reason behind this complexity is related to the very nature of such languages which too closely resemble traditional programming languages and their *imperative style*: this requires service architects to take care of an intricate control flow if they want to (1) capture all possible ways things can go wrong and (2) adapt to exceptional conditions, letting the application meet its requirements even in the presence of changes.

In particular, the traditional way to achieve such kind of adaptation is by explicitly programming the orchestration *workflow*, hard-coding all the possible alternatives to meet the requirements, and by heavily using exception handling techniques to manage failures when they occur. This is quite hard per-se and cannot be done by inexperienced users. In addition, this approach forces the alternative ways to achieve the orchestration's goal to be mixed together with the exception handling code. This brings further complexity and results in orchestration models that are hard to read and maintain.

This severe limitation of the state-of-practice in service orchestration has been recognized by part of the research community, which is proposing *Automated Service Composition* (ASC) as an alternative approach. Unfortunately, we find that this alternative also has limitations. Indeed, the number of different approaches that fall under the ASC umbrella can be roughly classified in two main groups [5, 6]. The first includes approaches that aim at reaching a fully automatic construction of the orchestration from a large (potentially universal) set of semantically-rich service descriptions, to be interpreted, selected, combined, and executed by the orchestration engine. This should happen without the intervention of the service architect, whose role is fully subsidized by the engine itself. The second group of proposals is less ambitious, leaving to service architects the goal of defining an abstract model of the orchestration, which the engine interprets and makes concrete by selecting and invoking the actual services to accomplish each task. We claim that none of the two approaches completely solve the problem above. The former works in specific, restricted domains, but can hardly be applied in general scenarios, since it requires all services to be semantically described with enough details to allow the engine to choose and combine them in the right way to satisfy the users' goals. The latter is too restrictive, as it relies on the service architect to provide an abstract yet detailed enough model of the orchestration, often using languages like BPEL and BPMN, whose structure is the ultimate source of the problems.[1]

As we will better argument in the following chapters, existing approaches can hardly provide the required features to simplify the development of *adaptive service orchestrations*, which could actually cope with the inherent complexity of the open world they live in. To overcome these limitations, this thesis introduces DSOL,[2] an innovative approach for service orchestrations, explicitly designed to combine an easy-to-use language and a powerful runtime system to support the development and execution of self-adaptive compositions.

In DSOL we follow the mainstream path that suggests human intervention to model service orchestrations through an ad-hoc language, but we abandon the imperative style of currently available languages in favor of a strongly declarative alternative. DSOL allows an orchestration to be modeled by describing: (i) a set of *Abstract Actions*, which provide a high-level description of the elementary activities that are typical of a given domain, (ii) a set of *Concrete Actions*, which map the abstract actions to the actual steps to be performed to obtain the expected behavior (typically, invoking an external service), (iii) a Quality-of-Service (QoS) profile for each concrete action that models its non-functional characteristics (*e.g.,* response time and reliability), and (iv) the overall *Goal* to be met by the orchestration, including also the global non-functional requirements to be satisfied.

DSOL models are then executed by DEng - *the DSOL Engine* which leverages automatic planning techniques to elaborate, at run-time, the best sequence of activities (*i.e.,* service invocations) to achieve the goal. Whenever a change happens in the external environment, which prevents execution to be completed, DEng behaves in a self-healing manner. Through dynamic re-planning and advanced re-binding mechanisms, it finds an alternative path toward the orchestration goal and continues executing it. To evaluate the effectiveness of this approach we fully implemented DEng and made it available for downloading.[3]

In general, DSOL provides several advantages w.r.t. traditional approaches, ranging from a better support for the implementation of self-

[1]We will come back to this issues in Chapter 7, while discussing related work.
[2]DSOL stands for Declarative Service Orchestration Language
[3]DEng is available at http://www.dsol-lang.net

adaptive orchestrations to a more decoupled architecture in which service orchestrations may smoothly evolve to cope with changes in the external environment. DSOL also promotes a more readable and maintainable code through a clear separation of concerns, in which the orchestration logic is isolated from the logic to handle exceptions during execution.

Besides the proposal of the DSOL approach, other contributions of this thesis include:

- A new approach for Quality-of-Service (QoS) management in service orchestrations in which we don't limit the choice of the most appropriate services to the beginning of the execution, but we apply a technique we called *Adaptive re-binding* to proactively change service binding using the information collected during execution. Such technique leverages the DSOL adaptive capabilities to maximize the QoS perceived by end users.

- SelfMotion, a successor of DSOL for the development of adaptive service-oriented *mobile applications*. In SelfMotion, we propose a declarative approach to easily build phone and tablet applications by composing not only web services but also third-party apps, and platform-dependent components to access device-specific hardware (*e.g.,* camera, GPS, etc.). SelfMotion was also fully implemented for Android devices [7].

1.1 Organization of the Thesis

The reminder of this thesis is organized as follows:

- Chapter 2 describes the problem we are trying to tackle. To do so, it presents the deficiencies of current available mainstream language and why we need a paradigm shift like the one we propose. A motivating example guides this description.

- Chapter 3 starts by giving an overview of DSOL and its main advantages. Afterwards, it presents all the elements that compose a DSOL orchestration model. Furthermore, it describes how DSOL runtime system works and how it was designed to overcome exceptional situations.

- Chapter 4 presents an extensive comparison between DSOL and other state-of-the-art approaches, through several running examples. Furthermore, it presents an exhaustive performance evaluation of the DSOL runtime system and the overhead it imposes to the running orchestrations.

- Chapter 5 presents an extension to DSOL developed to include the support for built-in Quality-of-Service (QoS) attributes. It also presents a performance assessment of the additional overhead perceived when such QoS attributes are used to select the orchestrated services.

- Chapter 6 describes how we ported the DSOL language and its runtime system to another context: that of mobile application development.

- Chapter 7 discusses related works, starting from a historical perspective, describing those systems that are at the roots of DSOL. Then, it presents the most relevant alternative approaches and how they differ from DSOL, including the QoS part. Finally, it shows some related work in the area of mobile application development.

- Finally, Chapter 8 draws some conclusion and depicts directions for future works.

1.2 Publications

The research work behind this PhD thesis has lead to several publications, listed in this section in order of appearance in the thesis.

1. G. Cugola, C. Ghezzi and L. Sales Pinto. DSOL: A Declarative Approach To Self-Adaptive Service Orchestrations. In *Computing*, Volume 94, Issue 7, pages 579–617. Springer, 2012.

2. G. Cugola, C. Ghezzi and L. Sales Pinto. Process Programming in the Service Age: Old Problems and New Challenges. In *Engineering of Software, The Continuing Contributions of Leon J. Osterweil*, pages 163–177. Springer, 2011.

3. L. Sales Pinto, G. Cugola and C. Ghezzi. Writing Dynamic Service Orchestrations with DSOL In *ICSE 2012: Proceedings of the 34th International Conference on Software Engineering, Formal Demonstration*, pages 1383–1386. IEEE, 2012.

4. L. Sales Pinto, G. Cugola and C. Ghezzi. Dealing with Changes in Service Orchestrations. In *SAC 2012: Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1961–1967. ACM, 2012.

5. L. Sales Pinto. A Declarative Approach to Enable Flexible and Dynamic Service Compositions In *ICSE 2011: Proceedings of the 33th International Conference on Software Engineering, Doctoral Symposium*, pages 1130–1131. ACM, 2011.

   These 5 papers are at the basis of Chapter 3 and Chapter 4

6. G. Cugola, L. Sales Pinto and G. Tamburrelli. QoS-Aware Adaptive Service Orchestrations. In *ICWS 2012: Proceedings of the 19th International Conference on Web Services*, pages 440–447. IEEE, 2012

   This paper is at the basis of Chapter 5.

7. G. Cugola, C. Ghezzi, L. Sales Pinto and Giordano Tamburrelli. SelfMotion: A Declarative Approach for Adaptive Service-Oriented Mobile Applications. Submitted to *Journal of Systems and Software, Special Issue on Middleware for Mobile Data Management.* Elsevier.

8. G. Cugola, C. Ghezzi, L. Sales Pinto and G. Tamburrelli. Adaptive Service-Oriented Mobile Applications: A Declarative Approach. In *ICSOC 2012: Proceedings of the 10th International Conference on Service Oriented Computing*, pages 607–614. Springer, 2012.

9. G. Cugola, C. Ghezzi, L. Sales Pinto and G. Tamburrelli. SelfMotion: A Declarative Language for Adaptive Service-Oriented Mobile Apps. In *FSE 2012: Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Tool Demonstrations*, Article No. 7. ACM, 2012.

   These 3 papers are at the basis of Chapter 6.

2 Problem Statement

As introduced in the previous chapter, the *Service Oriented Computing (SOC)* paradigm has brought a simplification in the way distributed applications are developed, giving the possibility to easily compose existing services to provide new added-value functionalities for end users. We claim, however, that the fully potential of SOC is still to be reached and it is our belief that this is caused by the same tools used to build such service compositions. Indeed, mainstream approaches failed to support dynamic, self-managed compositions, characteristics they must have due to the changeable world they are embedded in, where it is essential to be able to adapt to changes that may happen at run-time.

This chapter presents a deeper analysis of the deficiencies of mainstream service orchestration languages in supporting self-adaptation, which were also the main motivations to the approach proposed by this thesis.

2.1 Limitations of Currently Available Orchestration Languages

Throughout the last two decades, different approaches were taken towards the so-called *process programming* using workflow languages. Several programming and modeling languages were defined in an attempt to best define and automate different kinds of processes. More recently, the advent of SOC has attracted much research into the area of business processes, to provide foundations for formalization, automation, and support to business-to-business integration, where services provided by different organizations are combined, *i.e.,* orchestrated, to provide new added-value services that can be made available to end users.

Two languages emerged as the de-facto standards for modeling service orchestrations: BPEL [3] and BPMN [4]. Although the two have some differences [8], they share a number of commonalities that result in the same limitations in modeling complex processes. In particular, both adopt an imperative style, in which service orchestrations are modeled as monolithic programs that must capture the entire flow of execution. This requires service architects to address every detail in the flow among services—they must explicitly program all the sequences of activities and take care of all dependencies among them, consider all the different alternatives to accomplish the orchestration goal, and forecast and manage in advance every possible exception that may occur at run-time.

To be more precise about this issue and better illustrate the limitations of existing SOC approaches, let us introduce an example scenario called *The Event Planning Service.* Such example will be further used to motivate the need for a paradigm shift like the one we propose and to help us better describe our approach in details in Chapter 3.

2.2 The Event Planning Example

The Event Planning Service is an application designed to support the user to organize a trip to participate on an event of her choice. The service is supposed to help the user to buy tickets for the (night) event and to arrange the transportation from the city where the user lives to the city where the event is scheduled, plus the related accommodation. Such a service is built as an orchestration of existing external services, through which one can buy the ticket, book the transportation, and book the accommodation. In particular, we consider the following requirements:

1. The system shall initially ask the user to provide her relevant data, the city where she lives, the event she wants to attend, the credit card data to pay for the ticket, and the desired transportation and accommodation types.

2. Purchase of the ticket shall precede other actions, since the purpose of the trip is to participate in the event.

3. Transport purchase shall precede accommodation reservation. Transportation can be arranged either by plane, train, or bus. In the case the participant does not express a preference for a specific transportation type, the system shall automatically proceed by trying plane, train, and finally bus (in this order), to book the transportation.

4. Choice of accommodation shall have the following options: hotel and hostel, in this order of preference, unless explicitly chosen by the user.

5. The transportation and the accommodation must be booked for the same period. The preferred option is to book the transportation in such a way that the participant arrives at the event's location the day before the event and departs the day after, taking her

time to visit the place. The other choice is to book the outbound transportation for the same day of the event and returning the day after, thus requiring accommodation for a single night.

6. In the case the booking of transportation or the accommodation could not be performed, the ticket purchase must be canceled. The transportation booking must also be canceled if the accommodation could not be booked.

Looking at the requirements of the Event Planning Service it is clear that its overall goal can be accomplished in several ways, although there is some preferred (partial) ordering among the different actions that build the orchestration. In particular, while some paths represent different alternatives to accomplish the same task, others have to be done in sequence. For example, if the participant has no preference for transportation, booking a train is only required if there are no flights available, but the action of booking the transportation has to come before the action of booking the accommodation.

Beyond that, several things can go wrong at run-time that must also be managed. For example, service invocations may fail or take more time than expected. This means that, at design time, architects can make certain assumptions, but these may be invalidated at run time. To cope with those kind of uncertainty and unexpected situations, orchestrations can be designed as *adaptive* [9, 10], in order to easily adapt to changes occurred at runtime.

Implementing this kind of adaptive orchestrations using BPEL, considered the de-facto standard language for implementing service orchestrations, however, is not an easy task. We believe that the main reason behind this problem is related to the imperative programming style it adopts, which forces service architects to explicitly code all possible action flows, and to forecast all possible exceptions. Indeed, in practice, architects typically design adaptive behaviors by explicitly programming alternative paths and by heavily using exception handling techniques to adapt the execution to detected changes in the environment (*i.e.,* an unavailable service). Using this approach, however, such alternative paths cannot be kept separated from each other, from the exception handling code, and from eventual compensation logic used to undo already executed actions. As a result, the architecture and the code are hard to understand, maintain and evolve.

To illustrate this fact, Listing 2.1 shows a code snippet that expresses the different alternatives for booking the transportation in the Event Planning example. The code is divided in four distinct *if*-blocks. The first (lines 4-17), following the order of preference for transportation, contains the code responsible for booking the flight ticket. The condition to enter the first *if*-block checks either if the preferred transportation (`preferredTrans` variable) is set to `airplane` or set to `null`, what means the user has not expressed any preference. The second *if*-block (lines 18-31) is responsible for booking a train. Note that, besides checking the value bound to the `preferredTrans` variable, it also checks if the flight was already booked. This is part of the application's adaptation logic, which, in the case the user does not express any preference, tries all the available possibilities for booking the transportation. To implement such adaptation mechanism, the control flow relies on the fact that the external web services invoked to perform each single operation return `true` if they are successfully executed, and `false` otherwise. The return value is then bound to a variable. Afterwards, the value of this variable is tested, and if it is `false` the next alternative is executed; otherwise, the other alternatives are skipped. For instance, the condition of the second *if*-block (line 18) also includes the variable `flightBooked`, which guarantees that the execution will only try to book the train if the service used to book the flight has previously failed. The same is done subsequently, while trying to buy the bus ticket (third *if*-block), where such operation is only executed if the `flightBooked` and the `trainBooked` variables are false. This means that both operations could not be executed successfully. In the end, if no transportation is booked, an exception is thrown. Note that, together with all this adaptation logic it is also included compensation handlers to undo the effects of booking the transportation in case the accommodation for the same period could not be booked.

Although this is just a small fragment of the orchestration we have considered, 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 turns out to be. Things become even more complex when run-time exceptions, like an error in invoking an external service, enter the picture and we have to add the code to effectively manage them, *e.g.,* by invoking alternative services.

As previously stated, we argue that the main reasons behind these problems is that the most used orchestration languages too closely resemble traditional imperative programming languages with their need to explicitly program the flow of execution. Furthermore, this programming style forces the application code to be mixed with the code for fault and compensation handling, further reducing the overall readability of the resulting code.

Another important point to highlight is that, this approach makes it impossible to introduce alternative paths of actions when something unexpected happens during execution and none of the initially provided options are able to accomplish the orchestration's goal. Every possible problem has to be anticipated and managed at design time.

```
...
<scope name='EventPlanning'>
 <scope name='BookTransportation'>
  <if>
   <condition>
    <!-- preferredTrans equals airplane or
         preferredTrans is null -->
   </condition>
   <scope name='BookFlight'>
    <compensationHandler>
     <!-- Cancel flight reservation -->
    </compensationHandler>
    <invoke operation='bookFlight'
            inputVariable='transportationDetails'
            outputVariable='flightBooked' .../>
   </scope>
  </if>
  <if>
   <condition>
    <!-- preferredTrans equals train or
        (preferredTrans is null and not flightBooked) -->
   </condition>
   <scope name='BookTrain'>
    <compensationHandler>
     <!-- Cancel train reservation -->
    </compensationHandler>
    <invoke operation='bookTrain'
            inputVariable='transportationDetails'
            outputVariable='trainBooked' ... />
   </scope>
  </if>
  <if>
   <condition>
    <!-- preferredTrans equals bus or
         (preferredTrans is null and not flightBooked and
          not trainBooked) -->
   </condition>
   <scope name='BookBus'>
    <compensationHandler>
     <!-- Cancel bus reservation -->
    </compensationHandler>
    <invoke operation='bookBus'
            inputVariable='transportationDetails'
            outputVariable='busBooked' ... />
   </scope>
  </if>
  <if>
   <condition>
    <!-- not (trainBooked or flightBooked or busBooked) -->
   </condition>
   <throw faultName='TransportationNotBooked' />
  </if>
 </scope>
</scope>
...
```

Listing 2.1: Booking transportation in BPEL

3 DSOL: A Declarative approach for Service Orchestrations

Our main motivation in defining a new language for service orchestrations was the fact that none of the currently available languages designed for this purpose were able to cope efficiently with unforeseen exceptions, a feature we consider fundamental for a system that has to operate in an open, dynamic world.

After some initial research, we realized that to achieve this goal, we had to rethink the way in which service orchestrations are defined: we need a paradigm shift. As we already noted, the imperative programming style adopted by most process languages seems to be inappropriate to support flexible orchestrations for several reasons: (i) processes are modeled in a normative and rigid form, making runtime adaptations hard to achieve; (ii) they must capture completely different aspects within a single monolithic model, from control flow to exception and compensation handlers; (iii) they require sophisticated programming skills, precluding SOC from reaching a key goal: empowering even non-technical users to build their own service orchestrations.

To overcome such limitations, in this chapter we present DSOL[1] and its innovative runtime system which adopt a radically different, *declarative* approach to support self-adaptive service orchestrations. With DSOL, we aim to achieve two different goals: (i) simplify the definition of complex service orchestrations, in order to empower even non-technical users, such as domain experts; and (ii) increase the possibility of runtime adaptations by letting orchestrations evolve when unforeseen situations happen, or when the orchestration's requirements change.

A service orchestration modeled in DSOL includes different aspects, which are defined separately using different idioms, possibly by different stakeholders, each bringing their own competences. In particular, as shown in Figure 3.1, a service orchestration in DSOL includes the following elements:

- the definition of the *orchestration interface*, *i.e.,* the signature of the service that represents the entry point to the orchestration;

[1]DSOL stands for Declarative Service Orchestration Language

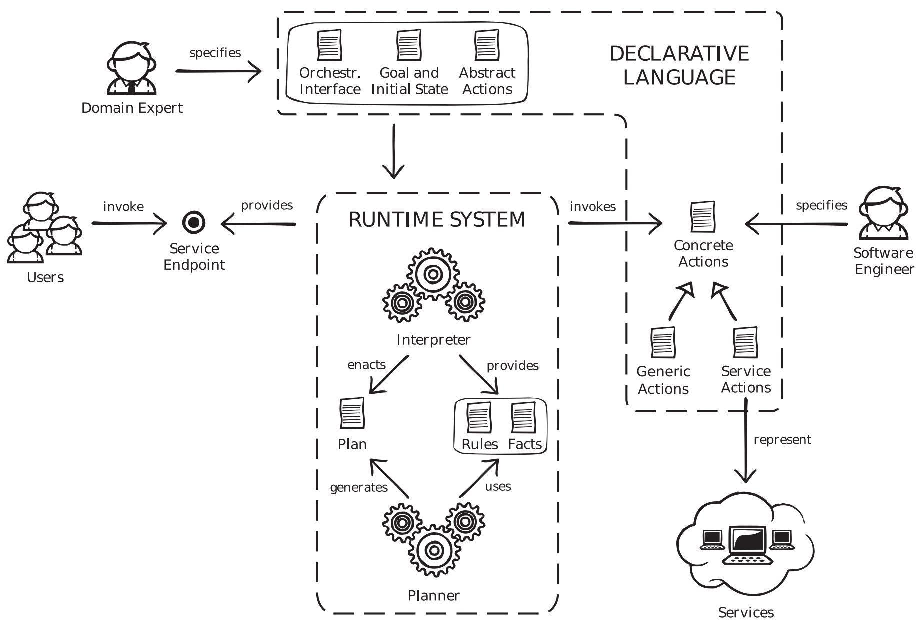


Figure 3.1: The DSOL approach to service orchestration

- the *goal* of the orchestration, declaratively expressed by a domain expert, not necessarily competent in software development, as a set of facts that are required to be true at the end of the orchestration;
- the *initial state*, which models the set of facts one can assume to be true at orchestration invocation time. This is described by the same domain expert who formulates the goal;
- a set of *abstract actions*, which model the primitive operations that can be invoked to achieve a certain goal and are typical of a given domain. They are described using a simple, logic-like language that can be mastered even by non-technical domain experts;
- a set of *concrete actions*, one or more for each abstract action, which maps each abstract action into the concrete steps required to implement the operation modeled by them, *e.g.,* by invoking an external service or executing some code. Clearly, defining concrete actions require programming skills, so they are written by a software engineer.

When the service exposed by the orchestration is invoked, the *Interpreter* translates the goal, the initial state, and the abstract actions into a set of *rules* and *facts* used by an internal *Planner* to build an abstract plan of execution, which lists the logical steps through which the desired goal may be reached. This plan is taken back by the Interpreter, which enacts it by associating each step (*i.e.,* each *abstract action*) with a *concrete action* that is executed, possibly invoking external services. If something goes wrong (*i.e.,* it returns an exception), the Interpreter first tries a different concrete action for the abstract action that failed, otherwise it invokes the Planner again to try a different course of action. In the extreme case, the service architect may intervene to add new abstract and concrete actions to be used to solve very complex situations.

This brief description shows the main advantages of our approach with respect to traditional ones:

1. We achieve a clear separation among the different aspects of an orchestration: from the more abstract ones, captured by goals, initial state, and abstract actions, to those closer to the implementation domain, captured by concrete actions.

2. We meet one of the original goals of SOC; *i.e.,* we involve users who are not expert in software development into the cycle.

3. By focusing on the primitive actions available and letting the actual flow of execution to be automatically built at run-time through the Planner, we allow orchestration designers to focus on the general aspects that are typical of a certain domain and remain stable over time, ignoring the peculiarities of a specific orchestration, which may change when requirements change. This last aspect also holds the promise to increase reusability, since the same abstract and concrete actions can be reused for different orchestrations within the same domain.

4. By separating abstract and concrete actions, with several concrete actions possibly mapped to a single abstract action, we allow the DSOL Interpreter to find the best implementation for each orchestration step and to try different routes if something goes wrong at run-time, in a fully automated way.

5. Because abstract actions only capture the general rules governing the ordering among primitive actions, the Interpreter, through a careful re-planning mechanism, can automatically overcome potentially disruptive and unexpected situations happening at run-time.

6. The modularity and dynamism inherent in the DSOL approach allow the orchestration model to be easily changed at run-time, by adding new abstract/concrete actions when those available do not allow to reach the orchestration's goal.

```
1 action buyTicket(Event,PD)
2 pre:  event(Event), paymentDetails(PD)
3 post: ticketBought
4
5 action bookFlight (From,To,Arrival,Departure)
6 pre: city(From), city(To), date(Arrival), date(Departure),
7      ticketBought, preferredTrans(airplane)
8 post: transportationBooked (From, To, Arrival, Departure)
9
10 action bookTrain (From,To,Arrival,Departure)
11 pre:  city(From), city(To), date(Arrival), date(Departure),
12       ticketBought, preferredTrans(train)
13 post: transportationBooked (From, To, Arrival, Departure)
14
15 action bookBus (From,To,Arrival,Departure)
16 pre:  city(From), city(To), date(Arrival), date(Departure),
17       ticketBought, preferredTrans(bus)
18 post: transportationBooked (From, To, Arrival, Departure)
19
20 action bookHotel (City,CheckIn,CheckOut)
21 pre:  city(City), date(CheckIn), date(CheckOut),
22       at(City,CheckIn,CheckOut), preferredAccomm(hotel)
23 post: accommodationBooked(City, CheckIn, CheckOut)
24
25 action bookHostel (City,CheckIn,CheckOut)
26 pre:  city(City), date(CheckIn), date(CheckOut),
27       at(City,CheckIn,CheckOut), preferredAccomm(hostel)
28 post: accommodationBooked(City, CheckIn, CheckOut)
```

Listing 3.1: The abstract actions for the Event Planning example

3.1 The Language

In this section, we provide a detailed description of the language, including details of each one of the elements used to specify a service orchestration in DSOL. To better describe such elements, we leverage the *Event Planning Service* presented in Chapter 2.

3.1.1 Abstract Actions

Abstract actions are high-level descriptions of the primitive actions available in a given domain, which we use as the building blocks of orchestration plans. They are modeled in an easy-to-use, logic-like language,in terms of their *signature*, *precondition*, and *postcondition.*

Listing 3.1 illustrates the abstract actions involved in modeling our Event Planning reference scenario. To clarify the structure of actions, let us take the `bookFlight` (line 5) case as an example:

1. The action *signature* includes its *name*, that must be unique in the orchestration domain, and a list of *arguments.* In the example,

   `bookFlight` is the name and `From, To, Arrival, Departure` is the list of arguments;

2. The *precondition* is expressed as a list of *facts* that are expected to be *true* in the current state for the action to be enabled. In our example, the expressions `city(From)`, `city(To)`, `date(Arrival)`, and `date(Departure)` are used to constrain the values of the arguments of the action, while the facts `preferredTrans(airplane)` and `ticketBought` express the fact that the `bookFlight` action must be invoked only if the user has chosen the airplane as its preferred transportation and that the ticket must have been already bought.

3. The *postconditions* models, instead, the effects of the action on the current state of execution by listing the facts to be added to the state, *i.e.,* facts that become `true`, and those to be removed, *i.e.,* become `false`. In our example, when `bookFlight` is executed the fact `transportationBooked(From, To, Arrival, Departure)` is added to the state while no facts are removed (removed facts, when present, are designed using the "$\sim$" symbol).

Facts in our language are expressed as *propositions*, characterized by a name and a set of arguments. The latter represent the relevant objects of the domain. More specifically, arguments that start with an uppercase letter are considered as *unbound* objects and must be replaced by actual instances, *i.e.,* those which start with a lowercase letter, to generate an execution plan. For instance, if at any point of plan generation the fact `city(event.city)` is added to the state, the object `event.city` becomes available to be bound either to the `From` or `To` generic arguments in the `bookFlight` action.

In some cases, it is necessary to relate different states of a domain,*e.g.,* to specify that when a certain situation arises new facts could be deduced. To model these situations we introduce *seam actions*. Unlike standard abstract actions, seam actions do not have a concrete counterpart, as they do not model an actual step of the orchestration, but rather a logical relation among facts in the domain. Listing 3.2 shows the seam actions used in the example scenario. Consider the `onTransportationBooked` case as an example. This action models the fact that after transportation is booked we may assume that the user will be at destination from the date of arrival till the date of departure. Similarly, the seam action `setTransportationPreference` models the fact that if the user does not express any preference about the transportation mode then all the three means (airplane, train, and bus) are equally possible.

```
seam action extractEventInformation
pre: event(event)
post: city(event.city), date(event.date), date(event.dayAfter),
      date(event.dayBefore)

seam action setTransportationPreference
pre:  preferredTrans(empty)
post: preferredTrans(airplane), preferredTrans(train),
      preferredTrans(bus)

seam action setAccommodationPreference
pre:  accommodation(empty)
post: accommodation(hotel), accommodation(hostel)

seam action onTransportationBooked(From,To,Arrival,Departure)
pre:  transportationBooked(From, To, Arrival, Departure)
post: at(To,Arrival,Departure)
```

Listing 3.2: The seam actions for the Event Planning example

```
action bookFlight (From,To,Arrival,Departure)
pre: city(From), city(To), date(Arrival), date(Departure),
     bookFlightAllowed
post: transportationBooked (From, To, Arrival, Departure)
```

Listing 3.3: The abstract action `bookFlight` from a different model

```
action enableBookFlight
pre: ticketBought, preferredTrans(airplane)
post: bookFlightAllowed
```

Listing 3.4: The seam action used to move the `bookFlight` abstract action to the Event Planning model

Another important use of seam actions is to increase reusability of abstract actions coming from different models when they use different terms to express the same concepts. Seam actions may therefore help abstract actions to move from one ontology to another. Let us consider, for example, that the `bookFlight` was initially taken from a different model and was defined as in Listing 3.3. Note that, now instead of having `preferredTrans(airplane)` and `ticketBought` as preconditions, the `bookFlight` abstract action has `bookFlightAllowed`. In order to reuse such abstract action, we could use a seam action, as illustrated in Listing 3.4 to correlate the facts from the two models.

3.1.2 Orchestration Goal and Initial State

Besides abstract actions, the *initial state* and *goal* are also needed to produce an orchestration plan. The former models the state from which the orchestration starts, while the latter represents the desired state to reach after executing the orchestration. The goal may actually include a *set of states*, which reflect all the alternatives to accomplish the goal of the orchestration, listed in order of preference. The Planner will try to build a plan that satisfies the first goal; if it does not succeed, it will try to satisfy the second goal, and so on.

The initial state and goal of the Event Planning scenario are illustrated in Listing 3.5. In this case the initial state is empty, while the goal models the acceptable results of our example, in which the orchestration will first try to book the transportation and accommodation for the night before the event until the day after, and then, if it can not be accomplished, it tries to book from the same day of the event until the next day.

```
start true

goal
  ticketBought,
  transportationBooked(participantCity, event.city, event.dayBefore,
                       event.dayAfter),
  accommodationBooked(event.city, event.dayBefore,
                      event.dayAfter)
or
  ticketBought,
  transportationBooked(participantCity, event.city,
                       event.date, event.dayAfter),
  accommodationBooked(event.city, event.date, event.dayAfter)
```

Listing 3.5: Initial state and goal for the Event Planning example

3.1.3 Orchestration Interface

To formalize how the orchestration is exposed as a web service, DSOL uses a Java interface properly annotated with JAX-WS [11] verbs to let the Interpreter automatically build the WSDL of the service.

The same annotations, in particular `@WebParam`, are also used by the Interpreter to create a set of additional facts to be passed to the Planner. As an example, from the orchestration interface of our reference scenario, shown in Listing 3.6, the Interpreter builds the facts `city(participantCity)`, `event(event)`, and `paymentDetails(pd)`, which introduce new objects to be used in generating the plan. The other two arguments of the interface are also transformed into facts for the Planner,

```
@WebService
public interface EventPlanning {
 public void plan(
  @WebParam(name="city")
  String participantCity,
  @WebParam(name="event")
  Event event,
  @WebParam(name="paymentDetails")
  PaymentDetails pd,
  @WebParam(name="preferredTrans") @Concrete
  String transportationMode,
  @WebParam(name="preferredAccomm") @Concrete
  String accommodationType
}
```

Listing 3.6: The Event Planning orchestration interface

```
buyTicket(event,pd)
bookFlight(participantCity, event.city, event.dayBefore,event.dayAfter)
bookHotel(event.city,event.dayBefore,event.dayAfter)
```

Listing 3.7: A possible plan for the Event Planning example

but in a different manner. As they are annotated as `@Concrete`, their actual value (as passed by the client and transformed into a string using the `toString` Java method) is used, not the formal parameter name. Hence, if the client invokes the service with a value `airplane` for the `transportationMode` parameter, the fact `preferredTrans(airplane)` is added to the set of facts passed to the Planner. Similarly, if a `null` value is used, the fact `preferredTrans(empty)` is used. The same will happen for the `accommodationType` argument.

Using these facts, plus the goals and initial state presented in the previous section, together with the abstract and seam actions, the Planner is able to build a plan, like the one presented in Listing 6.7 for our reference example.[2] It includes a list of abstract actions that can lead from the initial state to a state that satisfies an orchestration goal (the first one in our case). Notice that: (i) when several sequences of actions could satisfy the preferred orchestration goal, the Planner chooses one, non deterministically; (ii) although the plan is described as a sequence of actions, the Interpreter is free to execute them in parallel, by invoking each of them as soon as their precondition is satisfied.

[2]We omit the seam actions from the plan as they do not represent steps to be actually performed (*e.g.,* invoking external services) at run-time.

3.1.4 Concrete Actions

Concrete actions are the executable counterpart of abstract actions. They are intended to be specified by a different actor, *i.e.,* a technical person with programming skills, once the abstract actions have been identified and specified by the domain expert. In the reference implementation of the DSOL engine, concrete actions are implemented through Java methods using the ad-hoc annotation `@Action` to refer to the abstract actions they implement. In general, several concrete actions may be bound to the same abstract action. This way, if the currently bound concrete action fails, *i.e.,* it returns an exception, the DSOL Interpreter haves other options to accomplish the orchestration step specified by the failed abstract action.

Among concrete actions, we distinguish between *service actions* and *generic actions*. Service actions are abstract methods directly mapped to external services. An special attribute `service` of the `@Action` annotation specifies the external service to invoke, while a hot-pluggable module of the Interpreter, the *Service Selector*, is responsible for taking this information and finding the specified service to be invoked. This way, service actions may represent different kinds of services, *e.g.,* SOAP or HTTP, while the Interpreter can be easily extended to support other SOA technologies. Furthermore, having just a mnemonic label to reference a service allows us to have a loosely coupled model that could be easily modified, even at runtime. In fact, the actual information about the service (*e.g.,* URI, operation) is specified externally.

As an example of service actions, see Listing 3.8. Note that the `bookFlight` abstract action is bound to two different concrete actions, `bookFlightUsingExpedia` and `bookFlightUsingKayak`, which reference two different services, *expedia.com* and *kayak.com*, respectively.

```
@Action(name="bookFlight",service="expedia.com")
@ReturnValue("transportationDetails")
public abstract TransportationDetails bookFlightUsingExpedia(
        String from, String to, Date arrival, Date departure);

@Action(name="bookFlight",service="kayak.com")
@ReturnValue("transportationDetails")
public abstract TransportationDetails bookFlightUsingKayak(
        String from, String to, Date arrival, Date departure);
```

Listing 3.8: The `bookFlight` service action

Unlike service actions, generic actions are ordinary Java methods to be used as utilities every time an abstract action cannot be implemented by simply invoking an external service. For example, when the parameters of the abstract action to implement have to be pre-processed before being passed to an external service or when the action requires that some information must be saved to an internal database. Listing 3.9 shows two generic actions (code omitted for simplicity) to implement the `buyTicket` abstract action. Notice the use of the `@When` annotation to guide the Interpreter in choosing among different concrete actions for the same abstract action, based on the actual state of the orchestration.

```
@Action("buyTicket")
@When("pd.method.equals('CREDIT_CARD')")
public void buyTicketCreditCard(Event evt,
                                PaymentDetails pd){
  // Buy ticket and pay with credit card
  buyTicketUsingCreditCard(evt.getId(), pd.getCardNumber(),
                           pd.getCardHolder());
}

@Action("buyTicket")
@When("pd.method.equals('BANK_TRANSFER')")
public void buyTicketBankTransfer(Event evt,
                                  PaymentDetails pd){
  // Buy ticket and pay with bank transfer
  buyTicketUsingBankTransfer(evt.getId(), pd.getAccountNumber());
}
```

Listing 3.9: The `buyTicket` generic action

To execute concrete actions and, consequently execute the orchestration plan, the Interpreter needs to be able to determine which objects should be used as parameters in the methods invocations. Indeed, the actual state of the orchestration 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, which maps each abstract object used by the Planner and referenced inside the plan with a corresponding concrete object. When the orchestration is invoked, the values passed by the client are associated with the corresponding abstract objects and they are used to start populating the Instance Session. When the Interpreter must invoke a concrete action to execute the next step of the plan, it uses the Instance Session to retrieve the Java objects to pass to the action, while the value returned by the action, if any, is kept into the Instance Session, mapped to the abstract object whose name is given through the `@ReturnValue` annotation (see Listing 3.8 for an example). This way the abstract plan produced by the Planner is concretely executed by the Interpreter, step by step.

To illustrate such mechanism, let us consider an invocation to the

Event Planning service, as illustrated in Listing 3.10. As soon as the DSOL Engine receives such request, it initializes the Instance Session as illustrated in Table 3.1. Note that, the formal parameter names of the orchestration interface are used as keys in the Instance Session and mapped to the received objects. Subsequently, as previous explained, the Interpreter generates the initial state, also based in the orchestration interface, and invokes the Planner, which returns the plan illustrated in Listing 6.7. The plan is then enacted as described hereafter.

To execute the first step of the plan, `buyTicket(event, pd)`, the Interpreter first takes the name of the abstract action, `buyTicket` in this case, and looks for all concrete actions bound to it, which are those illustrated in Listing 3.9. The Interpreter then chooses between the two concrete actions by evaluating the expression specified in the `@When` annotation. Let us first take the expression defined in the method `buyTicketCreditCard`, *i.e.,* `pd.method.equals(‘‘CREDIT_CARD’’)`. To evaluate such expression, the Interpreter uses the objects in the Instance Session, what means it will take the object mapped to `pd`. In such case, as the value of field `method` of object `pd` is `‘‘CREDIT_CARD’’`, the expression evaluates to `true`, and this concrete action is eligible to be executed. The other one, as its expression evaluates to `false`, is discarded. Having chosen which concrete action to be executed, the Interpreter needs to figure out which parameter values should be used in order to invoke it. To do so, it take the abstract objects used by the Planner in the current step of the plan, *i.e.,* `event` and `pd`, and uses them as keys to retrieve the objects from the Instance Session. After the objects are retrieved from the Instance Session they are used as parameters to invoke the selected concrete action. Note that, as the `buyTicketCreditCard` concrete action does not return any value, the state of the Instance Session remains the same.

To execute the second step of the plan the Interpreter initially follows the same process. It takes the name of the abstract action, *i.e.,* `bookFlight`, and find the concrete actions bound to it, *i.e.,* `bookFlight-UsingExpedia` and `bookFlightUsingKayak`. As both implementations are not annotated with a `@When` condition, both are eligible for execution. In this case the Interpreter chooses one of them non-deterministically.[3] Note that, in this step, to determine the parameters to be used in the concrete action invocation, the Interpreter uses a slightly different mechanism. Indeed, for some of the objects, *i.e.,* `event.city, event.dayBefore, event.dayAfter`, the “dot notation” is used to call

[3]Chapter 5 provides details on how non-functional requirements can be used to select the most suitable concrete action

```
...
String participantCity = "Fortaleza";

String eventName = "Beach Boys Concert";
Date eventDate = new Date("2012-11-12");
String eventCity = "Milan";
Event desiredEvent = new Event(eventName, eventCity, eventDate);

String creditCardNumer="0987654321";
String creditCardHolderName="Leandro Sales H Pinto";
PaymentDetails pd  = new PaymentDetails("CREDIT_CARD", creditCardNumer,
                                        creditCardHolderName);

String transportationMode = "airplane";
String accommodationType = "hote";

EventPlanning eventPlanning = ...; //Create client side proxy
eventPlanning.plan(participantCity, desiredEvent,
                   pd, transportationMode,
                   accommodationType);
...
```

Listing 3.10: Sample request to the Event Planning Service

their respective *getter* methods in the object extracted from the Instance Session. Thus, the following values are used to invoke the selected concrete action: “Fortaleza”, “2012-11-11”, “2012-11-13”. Finally, the value returned by the concrete action is added to the Instance Session under the `transportationDetails` key (the value specified in the `@ReturnValue` annotation). The third step is executed using the same mechanisms. Finally, if the orchestration interface contains a `@ReturnValue` annotation, its value will be used to extract from the Instance Session the object to be returned to the client.

KEY	VALUE
participantCity	"Fortaleza"
event	Event Object [name="Beach Boys concert", date="2012-11-12", city="Milan"]
pd	PaymentDetails Object [method="CREDIT_CARD", cardNumber="0987654321", holder="Leandro Sales H Pinto"]
transportationMode	"airplane"
accommodationType	"hotel"

Table 3.1: Instance Session before starting to execute the plan

3.2 Failures and Compensation Actions

The ability to tolerate—both expected and unexpected—exceptions to the standard flow of actions is fundamental for a system that has to operate in an open, dynamic world. Indeed, to pursue this goal, we provide both specific language constructs and ad-hoc run-time facilities.

Among language constructs, we have already mentioned the ability of associating different concrete actions to the same abstract action. This gives the Interpreter the ability to try different options to realize each step of a plan. Indeed, when an abstract action A has to be executed, the Interpreter tries the first concrete action CA_1 implementing A. If CA_1 fails, *e.g.,* it throws an exception, the Interpreter tries the second one, and so on. As an example, consider the two concrete actions mapped to the same abstract action `bookFlight` in our reference orchestration model, which invokes two different services. Now suppose that the *expedia.com* service is temporarily unavailable. When the Interpreter executes the `bookFlight` abstract action, the concrete action it is bound to may fail, but this does not stop the orchestration execution. In fact, the Interpreter automatically captures the exception and tries the second concrete action, which invokes a different external service, which hopefully is available and executes correctly.

If, however, none of the available concrete actions is able to execute correctly, a second mechanism is available, which involves the ability of building an *alternative plan* when something bad happens at run-time. That is, if the Interpreter is unable to realize a step (*i.e.,* an abstract action invoked with specific parameters) of the current plan, it invokes the Planner again forcing it to avoid the failed step. This way a new plan is computed that does not include the step that was impossible to realize. Furthermore, by comparing the old and the new plan, and considering the current state of execution, the Interpreter is able to calculate the set of actions that need to be *compensated* (*i.e.,* undone), as they have already been executed but are not part of the new plan. Figure 3.2 illustrated this process. As an example, consider the case where the outbound flight has been booked for the day before the event (and the inbound flight for the day after) according to the plan in Listing 6.7, but neither a hotel nor a hostel are available the day before the event. In such a situation, the Interpreter invokes the Planner again, which produces a new plan that reaches the second goal in Listing 3.5. This requires the `bookFlight` action to be compensated because the new plan requires the flight to be booked for the same day of the event.

Since the design of *compensating actions* usually requires application level knowledge, DSOL allows service architects to explicitly define them

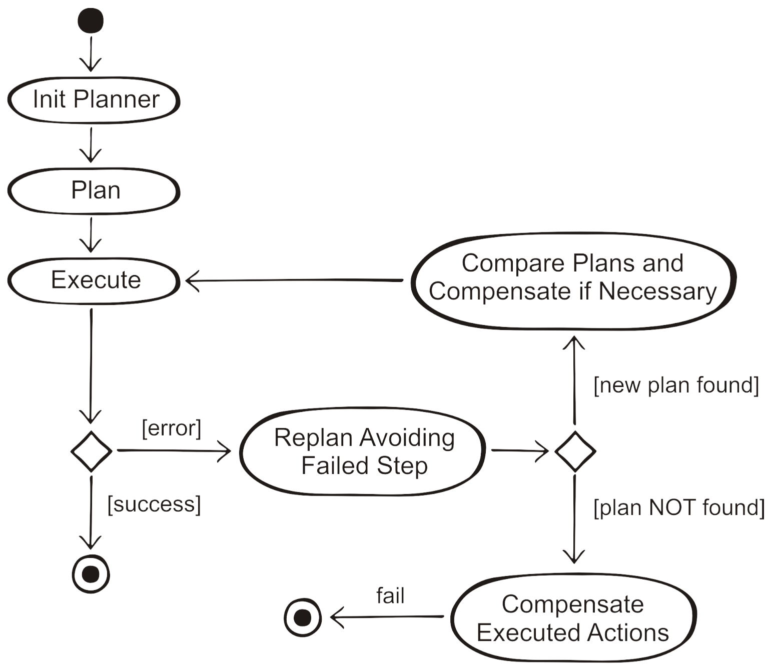

Init Planner
Plan
Execute
Compare Plans and Compensate if Necessary
[new plan found]
[error]
Replan Avoiding Failed Step
[success]
[plan NOT found]
fail
Compensate Executed Actions

Figure 3.2: Process followed by the Interpreter to execute a service orchestration

for each defined concrete action. Listing 3.11 shows how to compensate the `bookFlight` action. We notice how compensation actions use the same syntax of concrete actions, with the following differences: (i) it includes the special `compensation=true` attribute to indicate that this method is used for compensation purposes, and (ii) the `name` attribute refers to the concrete action, instead of an abstract action, which effects will be compensated when this method is invoked. By default, compensation actions are invoked using the same parameters of the action to compensate. For those cases where it is necessary to use different parameters, they can be taken from the Instance Session using the `@ObjectName` annotation, as shown in Listing 3.11, which invokes the compensation action `cancelExpediaFlightReservation` with the `TransportatioDetails`, previously returned by the `bookFlightUsing-Expedia` it undoes.

Notice how in this example we used a further mechanism provided by DSOL to increase robustness of the orchestration, namely the ability to specify multiple goals for each orchestration (see Section 3.1.2). This opportunity has been leveraged by the Planner at run-time, to build a new plan that automatically bypasses the failed step.

```
@Action(name="bookFlightUsingExpedia",service="cancelExpedia",
        compensation=true)
public abstract void cancelExpediaFlightReservation(
 @ObjectName("transportationDetails")
 TransportationDetails flightDetails
);
```

Listing 3.11: Action used to compensate `bookFlightUsingExpedia`

In summary, by combining the ability to specify different implementations (*i.e.,* concrete actions) for each step of a plan, with the ability to rebuild failed plans in search of alternative courses of actions, possibly achieving different, still acceptable goals, our language and run-time system allow robust orchestrations to be built in a natural and easy way. Indeed, by combining these mechanisms, DSOL orchestrations are able to automatically get around failures and any other form of unexpected situation, by self-adapting to changes in the external environment.

This goal is also achieved thanks to the DSOL approach to modeling orchestrations, which focuses on the primitive actions more than on the specific flow of a single orchestration. This approach maximizes the chance that when something bad happens, even if not explicitly anticipated at modeling time, the actions that may overcome the situation have been modeled and are available to the Planner and Interpreter.

Furthermore, DSOL achieves a clear separation among the different aspects of the orchestration: 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, domain experts may focus on the functionalities the orchestration requires, ignoring how they will be implemented. This choice is delayed to the time when concrete actions are actually defined. Indeed, such approach, that decouples system design from its implementation, is typical of mature engineering domains but it is not currently supported by mainstream orchestration languages. DSOL's loosely coupled architecture promotes also a clean modularization of the orchestration's functionalities, avoiding convoluted code which uses cascaded if-else blocks and exception handling constructs, improving readability and maintainability of the code. Finally, by encapsulating the different features of an orchestration into independent actions and by letting the actual flow of execution to be automatically built at run-time by the Interpreter, DSOL increases reusability, since the same action can be easily reused across different models, and also the possibility of evolving the orchestration model at runtime, as described hereafter.

3.3 Dealing with Changes at Runtime

Service orchestrations live in a very unstable world in which changes occur continuously and unpredictably. External services invoked by the orchestration may be discontinued by their providers, they may fail, or they may become unreachable or incompatible with the original versions. Furthermore, the orchestration requirements may evolve due to business needs. It is therefore fundamental that orchestration languages and their run-time systems provide ways to support dynamic evolution, allowing orchestration models or even currently running instances to be modified to cope with unforeseen situations and changes in requirements.

To better illustrate these needs, we introduce another example scenario, which we will use throughout this section to present how DSOL model can be changed at runtime.

3.3.1 The Bookstore Example

The Bookstore Service is a service orchestration used by a new bookstore to perform its sales. Initially, this service composes the following operations: an internal service that checks if the desired book is available in stock, an external service that handles the payments, and another external service used to contact the business partner that handles deliveries. In particular, let us consider the following requirements:

- At service invocation, the client provides the relevant information about the books she wants to buy, the delivery address and the details about payment;

- First action to perform is checking the availability of books. If they are in stock the order is saved with status *open*. Otherwise, an exception is thrown;

- After checking availability and saving the order, the process continues to the payment stage. For payment, two alternatives are available: PayPal or credit card. The preferred option is to use PayPal, because in such way the bookstore does not need to receive or keep any information about payments, *e.g.,* credit card numbers, limiting its responsibilities. PayPal itself also offers much more alternatives of payments to the client. If, for any reason, the system is not able to invoke the PayPal services, it enables the use of credit card with two alternatives. The first is to contact the card operator directly, using its services API. In case this route fails, the last alternative is to save the credit card information so that the payment can be done manually. In such case, the status of the order becomes *payment pendent*, (otherwise it is *payment authorized*). If the payment is not authorized, an exception is thrown;

- The delivery can only be scheduled after the payment. Depending on the payment status, the delivery is scheduled as *immediate* or *wait for confirmation*.

As a new bookstore that wants to enter the market gradually, due to contractual costs reason, the managers decided to initially sell only to national destinations. However, during this initial period the bookstore receives a huge, international order request from a foreign university. Although it is still not worth to start selling worldwide, the manager sees this situation as an interesting business opportunity and wants to fulfill this request. The problem is that neither the current system is able to handle it, because international orders require an extra insurance in case of miscarriage, nor the current deliver company is able to handle international deliveries. In order to complete this order, the system needs to deviate [12] from the current process, including the mandatory insurance, and also contacting a delivery company able to deliver at the required destination.

Another natural evolution would be to accept international orders and ship worldwide. The main difference of this case w.r.t. the previous one is that the former is a deviation that applies only to a specific running instance of the process, while the latter is an evolution of the whole process, which will affect all further executions.

The DSOL Orchestration Model

To recall the concepts we have previously introduced in this chapter, hereafter we describe how the Bookstore Service(BS) is implemented in DSOL.

Orchestration Interface

Listing 3.12 shows how the orchestration interface is defined for the BS orchestration. Note that, reflecting the requirements, it receives as arguments the books to be ordered, the delivery address and payment details.

Orchestration Goal and Initial State

Listing 3.13 shows the initial state and the goal for the bookstore scenario. In particular, two alternative goals are listed, the preferred one

```
@WebService
public interface Bookstore {
 @ReturnValue(name='order')
 public OrderInfo order(
  @WebParam(name='books_list')
  List<Book> books,
  @WebParam(name='deliveryAddress')
  Address deliveryAddress,
  @WebParam(name='paymentDetails')
  PaymentDetails pd);
}
```

Listing 3.12: The bookstore orchestration interface

```
start true

goal
  (booksAvailableInStock and orderSaved  and
   paymentDoneByPayPal    and deliveryScheduled)
or
  (booksAvailableInStock    and orderSaved  and
   paymentDoneByCreditCard and deliveryScheduled)
```

Listing 3.13: Initial state and goal

that describes the situation when payment has been done through PayPal, and the alternative to follow if paying through PayPal is impossible. As for the initial state, to model the bookstore scenario we do not need to assert any special fact, so the initial state becomes `true`. Notice however that the DSOL runtime system automatically populates the initial state with facts that reflect the orchestration arguments: `books_list(books)`, `address(deliveryAddress)`, and `paymentDetails(pd)`.

Abstract Actions

Listing 3.14 illustrates the abstract actions that model the bookstore scenario. On one hand, we notice the fact that the payment by credit card can be implemented in two different ways (automatically or manually) is not visible at this level. Similarly, we notice that there is no reference to the order of execution, which will be automatically chosen by the Interpreter (see Figure 3.2) at runtime in order to satisfy the goal.

Using the specified abstract actions, the initial state and the orchestration goal the DSOL runtime system is able to build the plan shown in Listing 3.15. Indeed, it includes a list of abstract actions that can lead from the initial state to a state that satisfies the first goal of the bookstore service.

```
action  checkStock(Books)
pre:    books_list(Books)
post:   booksAvailableInStock

action  saveOrder(Books)
pre:    books_list(Books), booksAvailableInStock
post:   orderSaved, order_info(order)

action  payByPayPal(Order,PD)
pre:    order_info(Order), paymentDetails(PD)
post:   paymentDoneByPayPal

action  payByCreditCard(Order,PD)
pre:    order_info(Order),paymentDetails(PD)
post:   paymentDoneByCreditCard

action  scheduleDelivery(Order, Address)
pre:    order_info(Order), deliveryAddress(Address)
post:   deliveryScheduled
```

Listing 3.14: Abstract actions

```
buyTicket(event,pd)
bookFlight(participantCity, event.city, event.dayBefore,event.dayAfter)
bookHotel(event.city,event.dayBefore,event.dayAfter)
```

Listing 3.15: A possible plan for the bookstore example

Concrete Actions

Listing 3.16 illustrates some of the concrete actions present in the BS orchestration. In particular, it includes one implementation of the `payByPayPal` abstract action and two alternatives to accomplish the abstract action *payByCreditCard*.

Changing the Bookstore model

While the declarative nature of DSOL allows easily modeling of flexible orchestrations, the modularity and dynamism inherent in the DSOL runtime system approach provide a perfect substrate where ad-hoc mechanisms can be added to change the orchestration at runtime. Indeed, as the plan of execution, *i.e.,* the actual sequence of activities to perform, is built at runtime, changing the orchestration is much simpler in DSOL compared to the complex mechanisms that other, more traditional systems must put in place to obtain the same result.

In particular, as we explain in detail in the remainder of this section, changing the orchestration at runtime requires the plan of a running orchestration to be re-built from the current state of execution: some-

```
@Action(name="payByPayPal",service="paypal")
public abstract void payByPayPal(OrderInfo order,
                                 PaymentDetails pd);

@Action(name="payByCreditCard",service="visa")
public abstract void payByCreditCard(OrderInfo order,
                                     PaymentDetails pd);

@Action(name="payByCreditCard")
public void savePaymentInfo(OrderInfo order,
                            PaymentDetails pd){
        order.setStatus(PAYMENT_PENDENT);
        order.setPaymentDetails(pd);
        order.save();
}
```

Listing 3.16: Example of concrete actions for the bookstore service

thing very similar to what the Interpreter already does to bypass a faulty situation that blocks the current plan.

In general we support full changes to the orchestration model. The service architect may add new abstract or concrete actions, remove or modify them, change the goal of the orchestration, and even change its interface. Moreover, we allow changes that impact the orchestration at various levels. Indeed, when the architect submits a new model for an existing orchestration it has to specify if it has to affect future executions, current ones, or both. This way, we cover different levels of updates: from small changes, applied to single running instances, to changes to be applied to future calls only, to major changes that have to affect current and future executions.

To better characterize this aspect, we define the concept of *orchestration instance* as the running orchestration that is created each time a request is made to the interface service of the orchestration. Such instance is represented internally by an *instance descriptor*, which we extended to include a copy of the orchestration model as it was defined at the time when the orchestration was invoked.

Given this premise, we notice that the case of a change that must affect only new instances does not creates special problems. In such case,we let current instances proceed using their own copy of the model, while the main copy, used for future calls, is overwritten with the new one.

The situation is more complex when the new model has to be applied to running instances. Indeed, this requires modifying the way the Interpreter operates. In particular (see Figure 3.3), as soon as the new model is submitted, the Interpreter stops executing the current plan and invokes the Planner to build a new plan in line with the new model. At

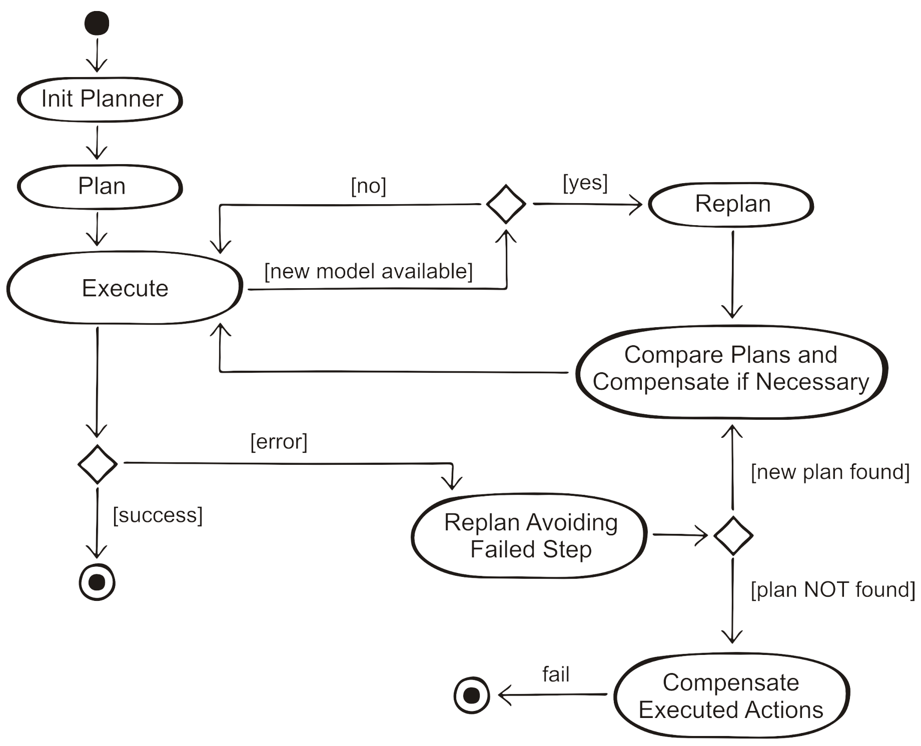


Figure 3.3: Process followed by Interpreter to execute an orchestration supporting changes at runtime

this point, as it happens for standard re-planning, the new plan is compared with the old one to decide where to start executing it and which action to undo, if any.

Notice that in applying a new model to a running instance we do not consider changes to the orchestration interface, which are taken into consideration only for future calls. Also notice that the declarative nature of DSOL and the modularity and dynamism inherent in its runtime system eliminate most of the typical problems about possible mismatches between the state of current executions and the changed model. In particular, during the re-planning phase that follows the submission of a new model, the Planner does not start from a generic "`true`" state, but it takes into consideration the current state of execution, *i.e.,* the facts already asserted during the execution of the old plan. This guarantees that the new plan, if found, is coherent with the current state.

To put in evidence the potential of these mechanisms, let us consider our case study. When the order request from the foreign university arrives we may decide to accept it by adding to the original model the

```
action buyOrderInsurance(Order)
pre:   order_info(order)
post:  orderInsuranceDone

action scheduleInternationalDelivery(Order, Address)
pre:   orderInsuranceDone, order_info(Order), deliveryAddress(Address)
post:  deliveryScheduled
```

Listing 3.17: New abstract actions for small deviation

abstract actions described in Listing 3.17 with the related concrete actions (omitted for brevity). As we do not want this new behavior to be shared by the whole system, those changes will be applied only to the running instance that received the request. This instance, instead of failing because the *scheduleDelivery* action could not be performed toward an international delivery, would detect those changes and trigger the re-plan phase. The new plan would include the *scheduleInternationalDelivery* action, that replace the *scheduleDelivery* actions to accomplish part of the goal, and the *buyOrderInsurance* that satisfies one of the pre-conditions of *scheduleInternationalDelivery*. The new plan executes successfully and allows to manage this exceptional situation.

On the other hand, imagine that at some time the managers of our bookstore had the chance to find two good partners to handle the international deliveries. The first and preferred one provides also the insurance of the order. The second one, although it deliveries to a broaden number of destinations, does not include the insurance.

To include the first partner into the job and open the bookstore to international clients, changing the concrete actions that implement the *scheduleDelivery* is enough. This is shown by Listing 3.18, which shows the concrete actions to include the first partner. Notice that we used the `@When` annotation to guide the Interpreter in choosing between the two methods. This change could be submitted as a global one, affecting current and future instances of the orchestration when the manager decides to extend their business worldwide. Similarly, the second partner could be included by changing the model as made to manage the special order from the international university, but this time applying the change globally.

```
@Action(name="scheduleDelivery")
@When("deliveryAddress.isNational()")
public void
scheduleNationalDelivery(OrderInfo order, Address deliveryAddress)
{...}

@Action(name="scheduleDelivery")
@When("deliveryAddress.isInternational()")
public void
scheduleInternationalDelivery(OrderInfo order,
                              Address deliveryAddress)
{...}
```

Listing 3.18: New concrete actions for process evolution

3.4 The DSOL Execution Engine

This section presents some relevant implementation details about the DSOL execution engine, called DEng. As described so far, DEng is organized into three main components: the Interpreter, the Planner, and the Service Selector.

The Interpreter is the core of the tool, being responsible for the execution of the orchestration. Essentially, it replies to requests coming from the clients by interpreting the orchestration modeled in DSOL, with the help of the Planner and Service Selector.

The Interpreter is built on top of Apache CXF [13], an open-source service framework. Apache CXF supports all the service-related parts inside DEng. It is responsible for building the service model (*i.e.,* the WSDL of the orchestration) from the orchestration interface specified in DSOL. Furthermore, it handles all the communication between external clients and the orchestration, including support for several transport protocols, *e.g.,* HTTP, Servlet, and JMS, and a variety of web service specifications including WS-Addressing, WS-Policy, WS-ReliableMessaging, and WS-Security. Apache CXF is also used to invoke the SOAP-based services that implement DSOL's service actions, as it provides a dynamic way to invoke such services, without the need for creating stub and data classes in advance. Conversely, to invoke HTTP services the Interpreter leverages the Apache Http Components [14], which support the different HTTP methods.

When a client submits a request to a DSOL orchestration, the Interpreter forwards the request to a class that implements the orchestration interface. This generic implementation is automatically built by the Interpreter at run-time, using the Code Generation Library (CGLIB) [15]. Its role is to read the meta-data of the called method and the actual values of the parameters and, based on this information, build the initial state of the orchestration and initiate the Instance Session with the key-value pairs that correspond to the actual parameters. Then, it retrieves the abstract actions and goals associated with the invoked method and passes them to the Planner.

The Planner is a hot-pluggable component of DEng, which is accessed by the Interpreter through a generic interface. The current DEng prototype uses an ad-hoc planner, built as an extension of JavaGP [16, 17], an open-source implementation of the Graphplan [18] planner. The JavaGP planner was extended 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 in the DSOL model are mapped to concrete actions whose invocation failed.

Using the initial state and the abstract actions provided by the Interpreter, the Planner builds a plan to reach the specified goal and returns it to the Interpreter, which enacts it by linking each step of the plan to the concrete actions that implement it. Classes that contain concrete actions (methods annotated with `@Action`) are parsed once for all at orchestration start-up, and they are stored in memory using a Hash Map binding the name of the each abstract action with the list of concrete actions (*i.e.,* Java methods) that implement it. In such a way, when a step of the plan needs to be executed, all available implementations can be retrieved in constant time.

To actually invoke concrete actions, the Interpreter parses the next step of the plan extracting the name of the abstract action involved and the names of the objects to pass as parameters. The former is used to retrieve the concrete actions to invoke from the Map mentioned above, while the latter are used to retrieve, from the Instance Session, the actual object instances to be used as parameters. Notice that concrete actions are invoked one by one until one is found that completes successfully. The return value of such concrete action is stored into the Instance Session, making it available to the following step of the orchestration.

As explained in Section 3.2, if none of the concrete actions completes successfully, the Interpreter starts a new interaction with the Planner in order to find a new plan. In this interaction, the Interpreter first informs the Planner of the steps that cannot be used any more (those corresponding to abstract actions for which no concrete actions are available) and then it requests a new plan. When the new plan is built, the Interpreter compares it with the old one to figure out if they have steps in common that have already been executed and if there are actions that have to be compensated. The corresponding compensation actions are executed before proceeding with the new plan.

Notice that some of the concrete actions to be invoked when enacting a plan may be "service actions" (see Section 3.1.4) modeled as properly annotated abstract Java methods. For such methods an implementation is created at run-time using the previously mentioned CGLIB. In particular, the Interpreter uses the meta-data of the called method to request the Service Selector for services that match the given identifier. Once the services are selected, they are invoked using the dynamic clients front-end provided by Apache CXF in case of SOAP-based services, and using Http Components in case of HTTP services. The parameters used to invoke the service are the same passed to the concrete action (which were retrieved from the Instance Session as explained above). The object returned by the service, if any, is used as a return value of the concrete action and it is stored into the Instance Session for later use.

As it happens for the Planner, the Service Selector was also designed to be a hot-pluggable component that can be replaced according to specific needs of the users. Currently, the Service Selector is implemented as a database of service descriptions, which can be managed at run-time through a specific API that allows new services to be added or existing ones to be removed. For each SOAP-based service known to the engine, this database holds information such as the service endpoint (WSDL), the port and the operation that must be invoked. For HTTP services, it holds the service endpoint (URL), the HTTP method to be used (GET, POST, PUT, DELETE) and the media type (our current prototype supports both XML and JSON) used to exchange messages.

Finally, service orchestrations written in DSOL can be deployed in two different ways. First they can be packaged as a Web application to be deployed in any Servlet container, such as Tomcat or Jetty. The second option is to use an embedded server, running as a standalone application. The first option has to be preferred when the orchestration is part of a Web application that includes its own HTML GUI as a front-end. The second option is easier to use as it relies on our DEng prototype only, not requiring any external component.

3.5 Conclusions

In this chapter we presented the DSOL approach for self-adaptive service orchestrations. In particular, we presented its language constructs and features to support runtime adaptation, including how exceptional situations can be easily overcome and managed. Furthermore, additional implementation details about DEng, the DSOL execution engine, were also presented.

The next chapter presents an exhaustive evaluation of DSOL, in which it is compared with other state-of-the art approaches through several additional case studies. Furthermore, it also presents an extensive evaluation of DEng's performance and scalability.

4 Case Studies and Evaluation

This chapter is divided in three parts. The first takes various examples of orchestrations from the literature and compares their implementation in DSOL and BPEL. The second does the same but taking AO4BPEL [19] and JOpera [20] as a reference. The goal is to show the advantages of DSOL with respect to two mature research proposals that address similar limitations of BPEL. The last part of the chapter focuses on performance, showing how the use of a planner to automatically decide the actual flow of the orchestration at run-time in practice is not a threat to scalability.

4.1 Comparison with BPEL

To reinforce the benefits provided by DSOL we take several examples from the literature and study how they can be implemented in DSOL and the differences between DSOL and a state-of-the-practice language such as BPEL.

4.1.1 The DoodleMap Example

DoodleMap [21] is a poll service used to make choices about places, with a map view of the selected locations. DoodleMap is built by composing the services provided by Yahoo! Local Search [22], Doodle poll service [23], and Google Static Maps [24]. The Yahoo! Local Search service is used to find places in a given location based on some criteria; for example, find available restaurants in the center of Milan. The results are used as choices for the Doodle poll service, which will create the poll, and as markers for the Google Static Maps, which will create and return an image map including the selected places.

At first, one could say that DoodleMap is a simple example but we argue that this is true only because it was not designed to be fault tolerant. For example, the service provided by Yahoo! Local Search only works if the queried location is inside the United States, taking the whole service to fail when the user searches for a location abroad. To overcome this limitation, one could replace this service by the one provided by Google Places [25] that works for more countries. However, there are situations in which even Google Places could fail, for example if it is temporarily unavailable or if it returns an empty list of places. In general, one way to implement a fault tolerant orchestration is to provide alternatives for the involved services, possibly in a straightforward manner. Accordingly, in our example we will include both Yahoo! Local Search and Google Places.

Unfortunately, the APIs provided by these two services are not compatible, so this situation cannot be addressed as a late binding problem. Yahoo! Local Search can be invoked directly with the search criteria and the location as provided by the user, while Google Places does not accept a textual location but needs geographic coordinates as the reference for the search. This is a common situation: including an alternative to a service also implies changing the workflow to introduce new accessory services, which were not originally required but are now needed to correctly execute the replacing service. In our case, before invoking Google Places we have to invoke a geocoding service to translate the location provided in human-readable form by the user in geographic coordinates, *e.g.,* Yahoo! PlaceFinder [26] or Google Geocoding [27].

```
<scope>
 <variables>
  <variable name='invokeGooglePlaces' type='xsd:boolean'/>
 </variables>
 <scope name='Yahoo'>
  <faultHandlers>
   <catchAll>
    <assign>
     <copy>
      <from>true()</from>
      <to variable='invokeGooglePlaces' />
     </copy>
    </assign>
   </catchAll>
  </faultHandlers>
  <!-- invoke Yahoo! Local Search -->
 </scope>
 <if name='GooglePlaces'>
  <condition>$invokeGooglePlaces</condition>
  <sequence>
   <!-- invoke Google Geocoding -->
   <!-- invoke Google Places -->
  </sequence>
 </if>
<scope>
```

Listing 4.1: BPEL implementation for the the DoodleMap alternatives

Listing 4.1 illustrates how the scenario presented above would be written in BPEL, applying the traditional approach that achieves fault tolerance through exception handling mechanisms. A boolean variable is declared (line 3) to determine whether Google Places (together with Google Geocoding) should be used or not. Initially the value of this variable is set to `false` and it is changed to `true` as soon as the fault handler (lines 6-15) attached to the scope `Yahoo` (lines 5-17) is enabled. This will happen when the Yahoo! Local Search service fails (line 16). Notice how the code, despite the simplicity of the example, is hard to read and how it would increase in complexity as we start adding other alternatives to this and the other services that are part of the overall orchestration. Furthermore, the order in which the alternatives are invoked, *i.e.,* the fact that Yahoo! Local Search has to be preferred to Google Places, is hard-coded into the orchestration and it is relatively difficult to change.

```
action findAvailablePlacesByLocation(Location, Query)
pre: searchLocation(Location), searchQuery(Query)
post: listofplaces(availablePlaces), bylocation

action getCoordinate(Location)
pre: searchLocation(Location)
post: searchCoordinate(Coordinate)

action findAvailablePlacesByCoordinate(Coordinate,Query)
pre: searchCoordinate(Coordinate), searchQuery(Query)
post: listofplaces(availablePlaces), bycoordinate
```

Listing 4.2: DSOL implementation for the the DoodleMap alternatives

Listing 4.2 illustrates how the same sub-scenario would be written in DSOL with a major focus on the abstract actions involved. The action `findAvailablePlacesByLocation` models services in which places are searched by location, as Yahoo! Local Search, while the action `find-AvailablePlacesByCoordinates` models the case in which nearby places are located based on geographic coordinates, *e.g.,* Google Places. Finally, the action `getCoordinates` models services that take a human-readable location and transform it in geographic coordinates. If the goal of this part of the orchestration is to find the list of available places, which we could model with the DSOL fact `listofplaces(availablePlaces)`, the Planner may satisfy it in two ways: searching places directly by location or first transforming the location into coordinates and then searching places using these coordinates. DEng will try the former route first and, if it fails, it would try the second one. All this happens at run-time and it is fully automatic: the domain expert focused on the available alternatives without the need for explicitly programming the exception handling code.

```
goal listofplaces(availablePlaces), bylocation
     or
     listofplaces(availablePlaces), bycoordinate
```

Listing 4.3: Using multiple goals to define ordering of actions

As for the order in which the alternatives are tested, it can be left unspecified or it can be explicitly modeled, through an accurate use of the goal. As an example, one could use the facts `bylocation` and `bycoordinates` provided by the abstract actions `findAvailablePlaces-ByLocation` and `findAvailablePlacesByCoordinate`, respectively, to write a goal (see Listing 4.3) that lists, as two different subgoals, the two alternatives, thus making the preferred order among them explicit: first try to find places by location, then by coordinates.

Another important point to highlight is how easily the DSOL code can be reused. Imagine a variant of the original DoodleMap example in which the user is equipped with a GPS device and the location to use is the current one. To address this case we have to change the orchestration interface and the overall workflow. The former receives the location as geographic coordinates instead of using a string, while the latter can now invoke the Google Places directly but it needs a reverse-geocoding service before invoking Yahoo! Local Search. In DSOL, this variant of the original orchestration can be modeled by fully reusing the actions part of the original model and adding the new abstract action shown in Listing 4.4. It is the Planner that chooses which actions to invoke and in which order to satisfy the goal of the original orchestration or the goal of the new one.

```
action getLocation(Coordinate)
pre: searchCoordinate(Coordinate)
post: searchLocation(location)
```

Listing 4.4: Reverse geocoding abstract action

Continuing with the DoodleMap example, we must now use the returned list of places to invoke the poll and the map service. As they are not related to each other, they can be invoked in parallel. However, before invoking those services, we need to transform the list of places into a list of choices compatible with the poll service and a list of markers compatible with the map service. Besides this, as the poll service is a stateful service it needs to be undone (*i.e.,* compensated) if for any reason the whole orchestration fails.

```
 1<scope>
 2 <faultHandlers>
 3  <catchAll>
 4   <compensateScope target="Doodle" />
 5  </catchAll>
 6 </faultHandlers>
 7 <flow>
 8  <scope name="Map">
 9   <sequence>
10    <!-- convert places into markers -->
11    <!-- invoke map service -->
12   </sequence>
13  </scope>
14  <scope name="Doodle">
15   <compensationHandler>
16    <!-- delete created poll -->
17   </compensationHandler>
18   <sequence>
19    <!-- convert places into poll choices -->
20    <!-- invoke poll service -->
21   </sequence>
22  </scope>
23 </flow>
24</scope>
```

Listing 4.5: Control flow mixed with compensation activities

In BPEL, as shown by Listing 4.5, all these issues have to be addressed together. The `<flow>` statement (lines 7-23) is used to specify that activities related to the map and poll services must be invoked in parallel, while the `<sequence>` statement (lines 9-12 and 18-21) indicates that the inner activities must be executed in a sequence. Lines 15-17 define the compensation handler for the `Doodle` scope, which is activated by the fault handler (lines 2-6) attached to the outer scope, which, in turn, catches all the faults that might occur as the orchestration is executed. The result is a rather convoluted code that would become really unmanageable if we had included the different service alternatives available.

The DSOL approach instead separates the various aspects. The fact that actions can be executed in parallel or have to be executed in sequence is deduced by the Interpreter based on the pre- and postconditions of the various actions part of the plan. While executing the plan, the Interpreter invokes actions as soon as all of their preconditions are true, *i.e.,* all the actions of a plan whose precondition is true are invoked in parallel. Instead if an action A_1 requires a fact f to be true before starting and f is asserted as a postcondition of an action A_2 then A_2 and A_1 will be executed in this order. The service architect does not need to worry about ordering and parallelism, which are deduced by the Interpreter looking at the model itself.

```
action createMarkers(Places)
pre: listofplaces(Places)
post: listofmarkers(markers)

action createMapWithMarkers(Markers)
pre: listofmarkers(Markers)
post: map(mapWithMarkedPlaces)

action createOptions(Places)
pre: listofplaces(Places)
post: listofoptions(options)

action createPoll(InitiatorName, PollTitle, Options)
pre: initiatorName(InitiatorName), title(PollTitle),
     listofoptions(Options)
post: poll(poll)
```

Listing 4.6: New abstract actions for the DoodleMap orchestration

Similarly, alternative ways to execute the same action can be coded by listing different concrete actions for the same abstract action, or different abstract actions with the same pre- and postconditions. Finally, compensation actions are written separately, as a special type of a concrete action. Ad-hoc annotations (see below) are used to identify them as compensation actions and to specify which action they compensate. It is the Interpreter's responsibility to decide, at run-time, if and when compensation actions have to be invoked.

If we look at our DoodleMap example, Listing 4.6 illustrates the abstract actions that need to be included to cover the new part of the orchestration. We notice that pre- and postconditions of `createMarkers` and `createMapWithMarkers` are put in relation by the predicate `listOf-Markers(...)`, denoting the need of running the two actions in sequence, if they appear in the same plan, while the precondition of `createMarkers` and `createOptions` are the same, making these two actions eligible for parallel execution, if they are part of the same plan.

Similarly, Listing 4.7 illustrates some of the concrete actions that implement the aforementioned abstract actions. Lines 3-8 represent a *generic action* that implements the abstract action `createOptions` and transforms the available places into options to the poll. Lines 14-16 represent a *service action* that implements the abstract action `createPoll` and is executed by invoking the service labeled with the key `poll` as defined in the `@Action` annotation. The returned object could be referenced in the future through the key `pollId`.

Listing 4.8 also illustrates the compensation action for `createPoll`. It can be identified as a compensation action because the `compensation`

```
@Action
@ReturnValue('options')
public List<Option> createOptions(List<Place> places) {
 List<Option> options = new List<Option>();
 for (Place place : places) {
  Option option = new Option(place.getName());
  options.add(option);
 }
 return options;
}

@Action(service = 'poll')
@ReturnValue('pollId')
public abstract String createPoll(String initiatorName,
                                  String pollTitle,
                                  List<Option> options);
```

Listing 4.7: Some concrete actions for DoodleMap scenario

```
@Action(name = 'createPoll', service = 'deletePoll',
        compensation = true)
public abstract void deletePoll(@ObjectName('pollId')
                                String poll);
```

Listing 4.8: Compensation action for the `createPoll` action

attribute in the `@Action` annotation is set to `true`. It receives the poll's id returned when the poll was created as a parameter and is executed by invoking the `deletePoll` service.

The last case we consider is that of an alternative that depends on an user's choice. In DoodleMap we can imagine that the map could be created using Google Maps or Bing Maps [28] based on the preferences of the user. The traditional approach to solve this case, and the one used in BPEL, is based on nested `if` statements or `switches`. This can be another source of complexity and another factor that limits reusability of the orchestration code, as its control flow is hardwired in the code. DSOL addresses the same case by using the actual value of the parameters passed to the orchestration as facts that become part of the Initial State (see Section 3.1.3) and can be used into the precondition of the various actions to decide which one has to be used. Listings 4.9 and 4.10 show the DSOL and BPEL code for this case, respectively.

As a final remark, Table 4.1 shows a comparison between DSOL and BPEL in terms of *lines of code* (LOC). The interpretation of LOC as a quality index is rather controversial and a meaningful statistical sample should be examined in order to support any conclusion. However, on the one side, we can say that the value of LOC for different languages to

```
action createMapUsingGoogle(Places)
pre: listofplaces(Places), mapProvider(google)
post: map(mapWithMarkedPlaces)

action createMapUsingBing(Places)
pre: listofplaces(Places), mapProvider(bing)
post: map(mapWithMarkedPlaces)
```

Listing 4.9: Alternatives based on the actual parameter values as part of the Initial State

```
...
<if name='MapProvider'>
 <condition>
  $doodleMapRequest.mapProvider = 'google'
 </condition>
 <!-- use Google Maps -->
 <else>
  <if>
   <condition>
    $doodleMapRequest.mapProvider = 'bing'
   </condition>
   <!-- use Bing Maps -->
  </if>
 </else>
</if>
...
```

Listing 4.10: Alternatives based on a service parameter using BPEL

implement the same functionality is an indication of the level of abstraction of the language. Table 4.1 indicates that DSOL provides a higher abstraction level. On the other side, LOC can be seen as an indicator of the effort needed to develop a piece of software (see [29]).The example suggests that DSOL might indeed help simplify development and reduce the required effort.

4.1.2 Other Examples

We repeated our exercise of comparing DSOL with BPEL by implementing four more scenarios: the Event Planning service, the Loan Approval service, the ATM service, and the Trip Reservation Service. The first is the example we presented in Chapter 2, which we implemented both in DSOL and BPEL. The last three were taken from the JBoss jBPM-BPEL v1.1.1 documentation [30]. In these examples, we took the BPEL implementations from the JBoss distribution and re-implemented them in DSOL.

We will not examine all of them in details as we did for the DoodleMap

LANGUAGE	CODE	#LOC
DSOL	Concrete Actions	42
	Orchestration Interface	12
	Initial State, Goals and Abstract Actions	22
	TOTAL	**76**
BPEL	Process	214
	Orchestration Interface (WSDL)	47
	TOTAL	**261**

Table 4.1: A comparison (in term of LOC) of DSOL and BPEL when used to model the DoodleMap orchestration

example, since the general considerations would be very similar. Rather, here we focus on the size of the resulting model. Tables 4.2 to 4.5 report the results we obtained. In all scenarios, the size of the BPEL code is between 2.2 and 4 times bigger than the code required by DSOL. The saving in size is bigger when the examples become more complex, with various alternatives in the execution flow. These results confirm what we argued before: that DSOL simplifies the specification of flexible and self-adaptive service orchestrations.

LANGUAGE	CODE	#LOC
DSOL	Concrete Actions	54
	Orchestration Interface	16
	Initial State, Goals and Abstract Actions	38
	TOTAL	**108**
BPEL	Process	381
	Orchestration Interface (WSDL)	53
	TOTAL	**433**

Table 4.2: A comparison (in term of LOC) of DSOL and BPEL when used to model the Event Planning orchestration

4.2 Comparison with Alternative Approaches

In the last years, several alternatives to BPEL were proposed by researchers to address some of the issues we emphasized in this thesis. In this section we describe two of them: namely AO4BPEL and JOpera,

LANGUAGE	CODE	#LOC
DSOL	Concrete Actions	36
	Orchestration Interface	19
	Initial State, Goals and Abstract Actions	12
	TOTAL	**75**
BPEL	Process	140
	Orchestration Interface (WSDL)	44
	TOTAL	**184**

Table 4.3: A comparison (in term of LOC) of DSOL and BPEL when used to model the Loan Approval orchestration

LANGUAGE	CODE	#LOC
DSOL	Concrete Actions	81
	Orchestration Interface	46
	Initial State, Goals and Abstract Actions	40
	TOTAL	**167**
BPEL	Process	369
	Orchestration Interface (WSDL)	116
	TOTAL	**485**

Table 4.4: A comparison (in term of LOC) of DSOL and BPEL when used to model the ATM orchestration

which we chose as mature representatives of the state of the art in the area. Other systems will be reviewed in Chapter 7.

4.2.1 Aspect-Oriented Extensions to BPEL

To increase modularity of orchestration models and to better support their run-time adaptation, some researchers proposed to use Aspect-Oriented Programming (AOP) techniques. BPEL'n'Aspects [31] and AO4BPEL [19] are two notable representatives of this class of systems, which are built around an aspect-oriented extension to BPEL. In this section we focus on the latter as it addresses both modularity and run-time adaptability as we do with DSOL.

In particular, in [19] the authors introduce various examples to illustrate how BPEL lacks tools to properly modularize *crosscutting concerns* among several processes. Here we take them and we show how DSOL behaves in those cases.

LANGUAGE	CODE	#LOC
DSOL	Concrete Actions	56
	Orchestration Interface	22
	Initial State, Goals and Abstract Actions	42
	TOTAL	**120**
BPEL	Process	201
	Orchestration Interface (WSDL)	66
	TOTAL	**267**

Table 4.5: A comparison (in term of LOC) of DSOL and BPEL when used to model the Trip Reservation orchestration

The first example presented in [19] (*data collection for billing*) assumes that a service provider starts charging a fee for using its Web Service S_1. The client, who will receive a bill from the provider, wants to check whether the bill is accurate. This requires counting how many times S_1 has been called from any deployed orchestration. In BPEL one would need to examine all the deployed orchestrations, finding out where the service is invoked, and manually including there the code to invoke a counting Web Service. AO4BPEL solves this problem more elegantly by declaring a single aspect to be executed after S_1, which invokes the counting Web Service.

Looking at this example from the DSOL perspective, however, we notice that what was hard to modularize in BPEL (the crosscutting concern) can be easily integrated into a single module in DSOL. Indeed, in DSOL the various orchestrations invoking the original Web Service S_1 would share a single abstract action, similar to action `invokeS1` in Listing 4.11. To count how many times such action has been invoked one could introduce a new action `countS1Invocations` (whose pre-condition is the post-condition of the `invokeS1`), changing the goals of the involved orchestrations to include the post-condition of the new action as shown in Listing 4.11[1]. This way the counting code is inserted only once and it is the Planner which guarantees that it is included into all plans that included the original action.

The second example presented in [19] (*data persistence*) moves from the consideration that in BPEL all data elaborated during an orchestration is lost as soon as the orchestration ends. In several scenarios, discarding the orchestration data is not an acceptable behavior. For in-

[1]The concrete action associated with action `countS1Invocations`, which actually invokes the counting service, is straightforward and has been omitted.

```
//Actions
action invokeS1(..)
pre: ..
post: S1Invoked

action countS1Invocations
pre: S1Invoked
post: counterForS1Incremented

//Goal
goal(.. and S1Invoked and counterForS1Incremented)
```

Listing 4.11: New abstract action and goal for the "data collection for billing" example

stance, in the Event Planning scenario, the payment confirmation code, the booking confirmation for the hotel, and the flight details should be stored. In BPEL, the solution for such problem would be similar to the solution presented for the *data collection for billing* example. The code to keep the desired data persistent would not be modularized in one place but replicated in different parts of different orchestrations. Again, AO4BPEL addresses such a situation by modularizing the persistence code into a single aspect that intercepts the calls for a given activity and stores the desired data for later use.

As in the first example, the peculiar approach to modeling orchestrations taken by DSOL makes the data persistence aspect a well modularized one. Indeed, DSOL distinguishes between the abstract, high-level model of an orchestration (abstract actions and goals) and its implementation (the concrete actions), leaving the actual flow of execution to be decided at run-time. The data persistence policy can be considered an implementation aspect to being modeled by introducing ad-hoc concrete actions as in Listing 4.12, which shows the original `bookFlight` and `bookTrain` concrete actions of the Event Planning examples, and their persistent counterparts that use an external data access object (DAO) [32] to persist data after invoking the original actions.

The third use case for AOP presented in [19] is about *business rules*. The authors do not make specific example, but claim that in general business rules are hard to modularize in BPEL and are amenable to be modeled as an aspect that intercepts some activities and encodes the business rule in a single place. As in the previous examples, DSOL does not suffer from this problem. Its rule-based nature allows to encode most business rules easily as part of the various abstract/concrete actions, while the fact that the orchestration flow is derived at run-time by the Planner starting from the available actions and the goal, guarantees that

```
@ReturnValue("transportationDetails")
@Action(name="bookFlight")
public TransportationDetails PersistentBookFlight(
        String from, String to, Date arrival, Date departure){
  TransportationDetails flightDetails = bookFlight(from,to,arrival,departure);
  DAO.saveData(flightDetails);
}

@ReturnValue("transportationDetails")
@Action(name="bookTrain")
public TransportationDetails PersistentBookTrain(
        String from, String to, Date arrival, Date departure){
  TransportationDetails trainDetails = bookTrain(from,to,arrival,departure);
  DAO.saveData(trainDetails);
}

@Action(service="flight")
public abstract TransportationDetails bookFlight(
        String from, String to, Date arrival, Date departure);

@Action(service="train")
public abstract TransportationDetails bookTrain(
        String from, String to, Date arrival, Date departure);
```

Listing 4.12: The modified `bookFlight` and `bookTrain` concrete actions including data persistence

the appropriate actions (i.e., the appropriate business rules) are included into every plan when they are required. As a further justification of this claim we notice that the same authors of AO4BPEL, in their paper [33] focus explicitly on the issue of appropriately modeling business rules and sketch two solutions considered equivalent: one based on AOP, the other based on a rule-engine that operates in a way similar to our Planner.

The final example presented in [19] concerns the *measurement of activity execution time and logging*. We acknowledge that this is the example that least fits DSOL and is also the one that mostly benefits from AOP, in our opinion. While we do not have a general solution for this case, we observe that DSOL and DEng are ultimately based on Java, so it is not hard to integrate them with one among the many available aspect-oriented extensions of Java, such as AspectJ [34]. For example, Listing 4.13 illustrates an AspectJ pointcut that intercepts all calls for a method annotated with `@Action` and calculates its execution time.

Besides modularity, AO4BPEL also claims that aspects can be used to solve the problem of dynamic changes and process evolution at runtime. To do so, aspects could be used to add, at runtime, new activities in specific points of the process. If business rules changes, the only thing to do is to activate or deactivate the appropriate aspect. We claim, however,

```
@Around("call(@org.dsol.annotation.Action * *(..))")
public Object executionTimeMonitor(ProceedingJoinPoint thisJoinPoint) {
  long start = System.currentTimeMillis();
  Object returnObject= thisJoinPoint.proceed();
  long end = System.currentTimeMillis();
  long executionTime = end - start;
  ...
  return returnObject;
}
```

Listing 4.13: Example on how AspectJ could be integrated with DSOL

that such an approach is not adequate for changing an orchestration. First, the changes are limited to the declared aspects, ignoring the fact that it is often necessary to change or remove also the activities that were initially declared into the orchestration. Furthermore, using aspects to evolve the orchestration means that changes, instead of being incorporated as a natural evolution of the model are realized more as a sort of patches, which could even complicate the overall understandability and maintainability of the orchestration.

In DSOL, changes are handled in a complete different way, since the modularity and dynamism inherent in the DSOL approach provide support for ad-hoc mechanisms to change the orchestration at runtime. Indeed, as the plan of execution, i.e., the actual sequence of activities to be performed, is built at runtime, changing the orchestration is much simpler in DSOL compared to the complex mechanisms that other, more traditional systems, must put in place to obtain the same result. In general we support full changes to the orchestration model. The service architect may add new abstract or concrete actions, remove or modify them, change the goal of the orchestration, and even change its interface. Moreover, we allow changes that impact the orchestration at various levels. When a new model for an existing orchestration is submitted, one can specify if it has to affect future executions, the current ones, or both. This way, we cover different levels of updates: from small changes applied to single running instances, to changes to be applied to future calls only, to major changes that have to affect current and future executions.

4.2.2 JOPERA

JOpera [20] is a mature research product that offers a visual language and a fully functional execution platform for building distributed application composed of reusable services, which includes, but is not limited to Web Services. The graphical and visual approach offered by JOpera simplifies modeling complex orchestrations when compared with BPEL. Moreover, the fact that the control flow and the data flow of the composition are described separately, allows one to build orchestrations that are easier to understand and maintain. On the other hand, the overall style of the language is still largely procedural and consequently it suffers from most of the flexibility problems we have previously highlighted.

Figures 4.1 and 4.2 illustrate, respectively, the control flow and the data flow diagrams for the DoodleMap scenario described in Section 4.1.1. These descriptions are inherently rather complex. As in BPEL, service orchestrations in JOpera must be modeled in all details, forcing the architect to decide and forecast at design time the best alternatives and the order in which such alternatives must be tried. Fault tolerance must be explicitly programmed by heavily using exception handling mechanisms. For example, the arrow with a red dot end that connects activity `YahooLocal` with activity `GoogleGeocode` means that `GoogleGeocode` must be executed after `YahooLocal` if the latter fails. Similarly, the question mark that annotates some activities denotes that they are guarded by some condition. For example, `CreateGoogleMap` and `CreateBingMap` depend on the user's choice. In DSOL all these features are handled automatically by the Planner, facilitating the job of the architect and allowing the orchestration to evolve more easily. Moreover, although the JOpera user does not need to specify some details, like, for example, which activities have to be performed in order and which ones can be done in parallel (a detail that is deduced by the execution engine from the data flow diagram), the process structure remains quite complex to define and rigid to evolve.

Although the graphical formalism provided by JOpera and the textual one provided by DSOL are not easy to compare, we contend that the fine-grain, imperative modeling imposed by JOpera leads to readability and maintainability issues that do not apply to the DSOL solution, which benefits from a declarative approach that focuses on the relevant details of the orchestration, while other aspects, including the actual flow of execution, are decided at run-time. As a comparison, DSOL requires 76 lines of code and 7 abstract actions in total to encode the same example reported in Figures 4.1 and 4.2.

4.3 Empirical Assessment

In the previous sections we focused on the expressiveness and usability of our modeling language DSOL. Here instead we are interested in testing if and how using a planner to decide the actual flow of the orchestration at run-time may negatively impact performance. To do so, we developed an

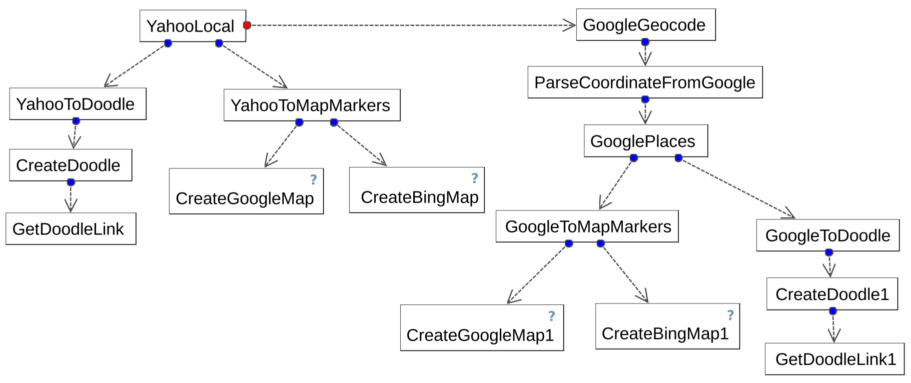


Figure 4.1: DoodleMap control flow modeled in JOpera

application that automatically generates different sets of related abstract actions and goals and we used the Planner to extract a plan from these data. To test the performance of the Planner under different situations, we varied the number of abstract actions that are part of the model and their structure, i.e., the number of parameters they have and the complexity of their pre- and postconditions. For preconditions, we also distinguished between the predicates that include some of the action's parameters (e.g., predicate `listofplaces(Places)` in the precondition of abstract action `createMapUsingGoogle` from Listing 4.9), from those that are pure (fully bound) facts that must be true for the action to be called (e.g., predicate `mapProvider(google)` in the same abstract action). The same distinction was made for postconditions, distinguishing between facts that involve some of the action's parameters (e.g., fact `listofplaces(availablePlaces)` in the postcondition of abstract action `findAvailablePlacesByLocation`), from those that are fully bound (e.g., fact `byLocation` in the same abstract action).

We considered a base scenario characterized by the following parameters: 50 abstract actions, 4 predicates in each precondition and postcondition (2 of one type and 2 of the other). As for the goal, it was chosen in a way that plans with different sizes would be generated[2]: 5, 10, and 15. We repeated each experiment 30 times and plotted the average result we measured and the 95% confidence interval [35].

Our evaluation was carried out for a server deployed in the Amazon cloud, configured as a small instance [36] and running Ubuntu Linux 10.10. This environment was set up to emulate a typical configuration

[2]The size of the plan differs from the actual number of abstract actions that compose it, as some of the actions can be executed in parallel. For plans of size 5, 10, and 15 the mean number of abstract actions in the plan is 8, 16, and 26, respectively.

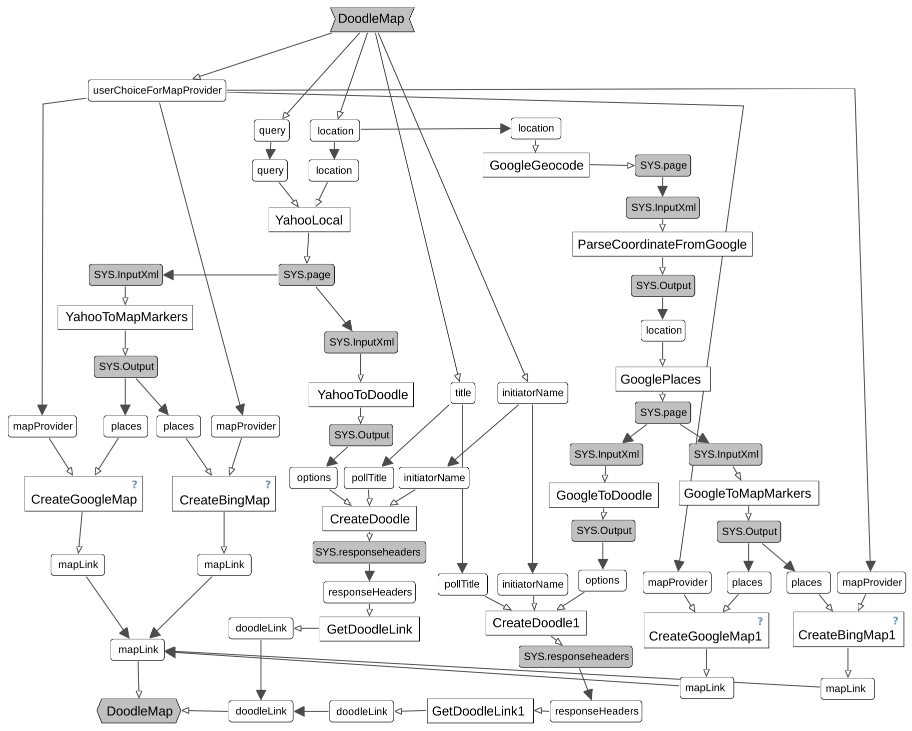


Figure 4.2: DoodleMap data flow modeled in JOpera

used to deploy service orchestrations in real scenarios.

Figure 4.3 shows the results we measured in the base scenario. They show the feasibility of our approach as the time to create the plan is very reasonable. For instance, if we consider the case of plans of size 10 (i.e., involving 16 abstract actions, on average) it only takes 250 ms to the Planner to build such plans. If only one third of the 16 actions are calls of external services, the overhead caused by external interactions will dominate the overhead imposed by the Planner. Even in the case of building a rather complex plan including 26 abstract actions on average, with size of 15, requires only 550 ms of the Planner.

Figures 4.4 and 4.5 show how the complexity of preconditions impacts on the time required to build the plan. In Figure 4.4, we consider the base scenario but we change the number of predicates in preconditions from 2 to 6. In Figure 4.5, we keep the number of predicates in preconditions fixed(4) and change the ratio of the two types of predicates involved. At one extreme all the predicates reference one of the action's parameters (they are unbound predicates), while at the other extreme they are all bound. While a growing number of predicates in precon-

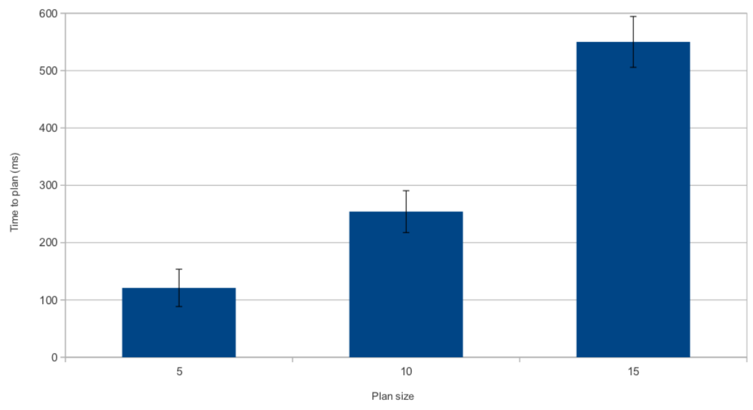


Figure 4.3: Time required to build plans of different sizes for the base case

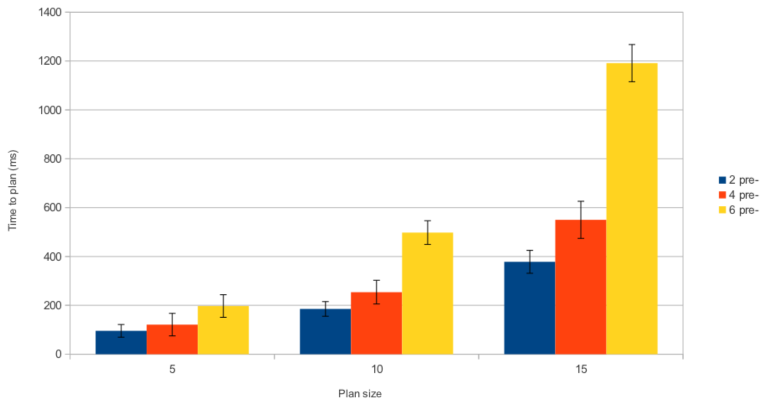


Figure 4.4: Time required to build plans of different sizes varying the number of predicates in preconditions

ditions and particularly a growing number of the unbound ones has a negative impact on the plan building time, this remains acceptable, with a max value of 3.5 sec and an average value just above 1 sec. Again, we expect these values to be dominated by the time required to invoke the external services that build such complex orchestrations.

Figures 4.6 and 4.7 make a similar analysis on postconditions. Notice that in Figure 4.7, the case in which none of the facts in postconditions refers to the actions' parameters is not possible because it would lead to an empty plan. The results are similar to the previous case, with a worst case of 4 sec and an average of 1.5.

Figure 4.8 shows the time required to build a plan in our base scenario

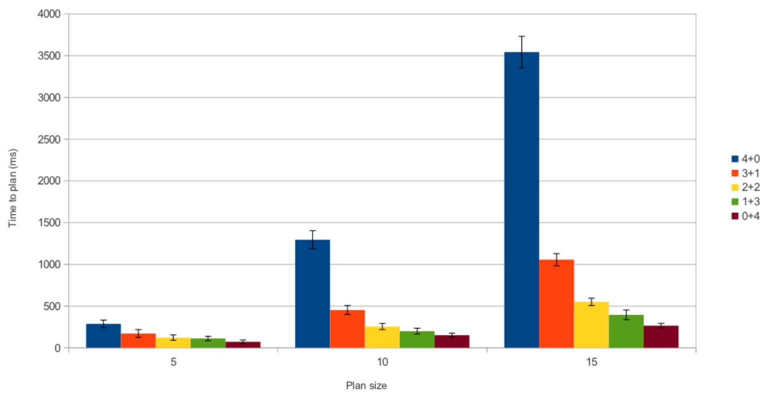


Figure 4.5: Time required to build plans of different sizes varying the proportion between the types of predicates in preconditions

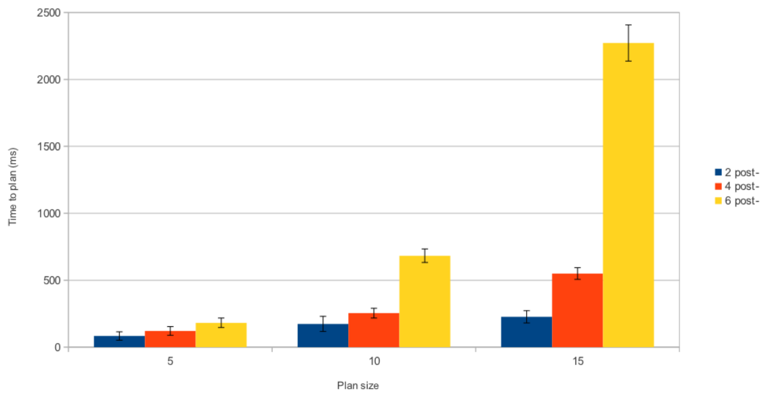


Figure 4.6: Time required to build plans of different sizes varying the number of facts in postconditions

but changing the number of available abstract actions from 30 to 100 (they were 50 in the base scenario). Our Planner scales very smoothly here, with a time that is almost constant and always below 1 sec. Starting from this positive result, we decided to investigate what happens in the worst case in which there is no plan to reach the goal. Figure 4.9 shows our base scenario, with a growing number of available abstract actions and with specially chosen goals that cannot be satisfied. Here we see that the number of available actions impacts on the number of combinations to test before concluding that no plan can be built. In the worst case (100 abstract actions) the Planner takes more than 15 sec to decide that no plan may reach the goal. This is a negative result but it has been obtained in a fairly tough scenario: 100 actions and no plan. In a more

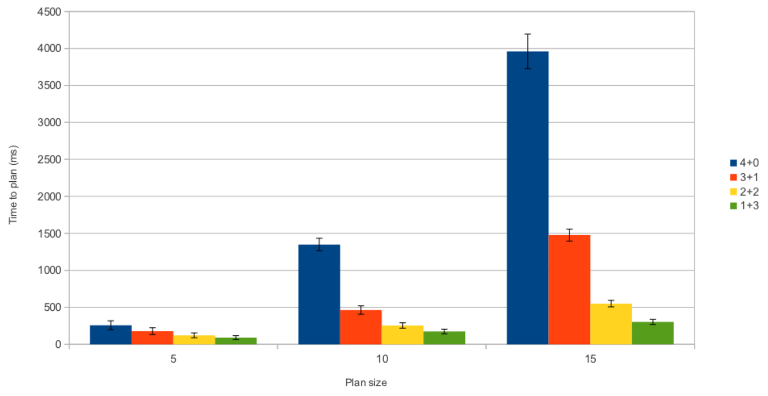


Figure 4.7: Time required to build plans of different sizes varying the proportion between the two types of facts in postconditions

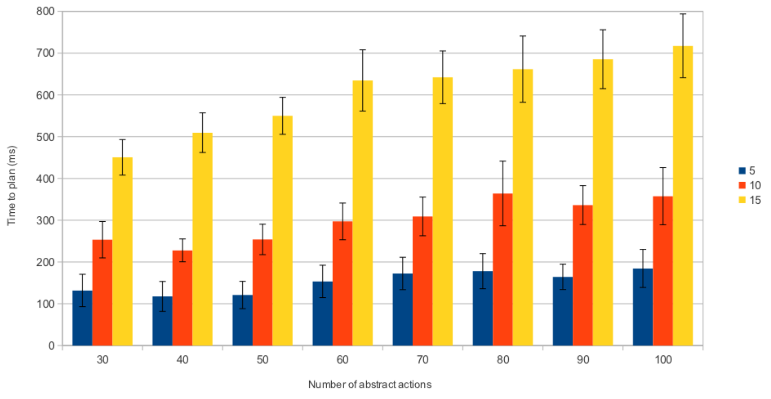


Figure 4.8: Time required to build plans of different sizes varying the number of available abstract actions

reasonably sized scenario like our default one (50 actions), the worst case of no plan requires only 2 sec to be solved.

In general, from the above assessment we may conclude that our approach is feasible and the use of a Planner introduces an acceptable (often negligible) overhead in the execution time of the overall orchestration. To further confirm this statement, we compared the overall performance of our DEng run-time, from planning to actual execution of the orchestration, with one of the most widely adopted BPEL engines: ActiveBPEL [37]. In particular, we compared the time required to complete the whole DoodleMap orchestration (see Section 4.1.1) including the time to invoke the various web services involved, observing the system from a client's perspective. We invoked the orchestration

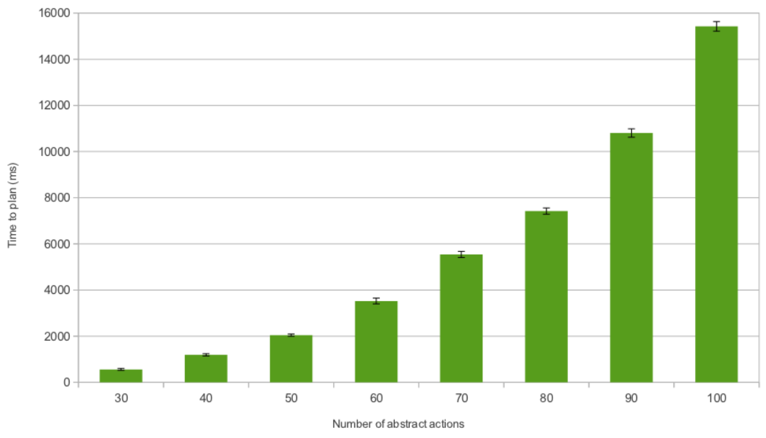


Figure 4.9: Time required to discover that a goal is not satisfiable in scenarios characterized by a growing number of abstract actions

with different inputs to test different paths of execution (including those that require DEng to initially build a plan that will fail, with the need to re-plan at run-time), and repeated our tests multiple times to account for the variations that may come from invoking the external services.

Engine	Time (ms)
DEng	1571
ActiveBPEL	1543

Table 4.6: Performance comparison between DEng and ActiveBPEL

Table 4.6 shows the results we obtained when the orchestration is invoked to build a DoodleMap for pizza restaurants in New York. Yahoo! Local Search completes successfully in this scenario and the orchestration is executed without faults. Both engines require approximately the same time to complete the orchestration, with DEng being slightly slower (by less than 30 ms).

Engine	Time (ms)
DEng	2579
ActiveBPEL	2130

Table 4.7: Performance comparison between DEng and ActiveBPEL in presence of faults that require re-planning

Table 4.7 shows the results obtained when the orchestration is invoked with pizza restaurants in Milan as the key parameter. The Yahoo! Local

Search fails as it can only handle requests for locations in the USA. In BPEL this failure triggers the fault handler that enables the orchestration to use the Google Places service to find the set of available locations. In DSOL, this will force the Interpreter to invoke the Planner once again to build the new plan that includes the Google Places related action. Again, the response time is similar, with DEng being slower by a greater but still acceptable margin of 450 ms.

Engine	Time (ms)
DEng	2075
ActiveBPEL	1873

Table 4.8: Performance comparison between DEng and ActiveBPEL

Finally, Table 4.8 shows the average time considering both alternatives together. In this scenario, the impact of our planning-based approach, plus the overhead introduced by the other parts of our engine, which make many run-time decisions using late-binding at each step, from the choice of the actual services to invoke, to the choice of the concrete actions to use, to the choice of the flow of execution itself, affect performance for 200 ms, i.e., 10% of the total time. We argue that this is an acceptable price to pay for the flexibility that it brings.

5 Extending DSOL with Quality of Service

So far, we have pointed out the advantages of implementing service orchestrations using DSOL w.r.t. traditional approaches (*e.g.,* [3, 4]) which force service architects to explicitly code all possible action flows, and to forecast both expected and unexpected situations. Unexpected situations in the service orchestrations, however, are not limited to the functional behavior of the composed services but also extend to the *Quality of service* (QoS) profile they offer, which includes, for example, their expected response time, reliability, and accuracy. Indeed, changes in the non-functional behavior of services may affect the orchestration's ability to satisfy its own QoS requirements.

To prevent those kind of violations and maximize the probability of satisfying also non-functional requirements, in this chapter, we present some extensions we have included to DSOL to consider also QoS profiles in order to select the services that best meet the overall non-functional requirements of the orchestration.

In particular, DSOL models the QoS attributes of external services as part of the available actions, while the QoS requirements of the whole orchestration are modeled as part of its goals. At orchestration invocation, DEng uses linear optimization techniques to search for an optimal service binding that could satisfy the orchestration goals, even in presence of conflicting non-functional requirements. In addition, DSOL supports two forms of run-time adaptation directly related with QoS. First, it is able of modifying the initial service binding to achieve further optimization given the knowledge acquired during execution (*e.g.,* the fact that a service, originally considered non-fully reliable, executed correctly). Second, it leverages the DSOL re-planning techniques to optimize the orchestration QoS in presence of faults, maximizing reliability. These forms of optimization and adaptability applies both to pre-defined QoS metrics (execution time and reliability), but also to user-defined metrics, which allow service architects to express domain specific, non-functional requirements.

5.1 Programmable Dinner Example

ProgrammableDinner (PD) is the service orchestration we use to illustrate the QoS-related functionalities of DSOL and its capabilities. PD orchestrates external services to organize a social event by choosing a restaurant, a movie, and inviting a group of friends. To implement PD, we considered the following requirements:

$RQ1$: The system shall initially ask the user to provide the relevant data, *i.e.,* the list of friends she wants to invite, the desired location for the social event and a movie title.

$RQ2$: Based on the indicated location, the system shall provide the weather forecast for the following days, plus a list of restaurants (including a set of reviews about each of them) and a map showing their position. Based on the movie title, the system shall suggest a list of movies similar to the one indicated, including a set of reviews about each of them.

$RQ3$: Given the data presented to the user, she must select a movie, a restaurant and the best day for the social event. Based on such information, the system shall provide the list of theaters that play the movie at the requested date.

$RQ4$: Using the location of the selected restaurant and theater, the system shall present the needed directions.

$RQ5$: The indicated list of participants shall receive a message containing all the details concerning the organized event.

$RQ6$: The overall system response time, not including the user think time, shall be less than 1.5s and the overall system reliability should be greater than 0.99.

$RQ7$: The system should perform as fast as possible.

$RQ1$-5 represent the desired functional behavior of the system, while $RQ6$ and $RQ7$ are non-functional requirements. Note that, while the satisfaction of the functional requirements depends directly on the choice of the available actions, the satisfaction of non-functional requirements depends on the choice of the specific service providers adopted which may be functionally identical but differ significantly for their quality profile.

As for the previously presented orchestrations, the overall goal—*i.e.,* the union of all functional and non-functional requirements—may be

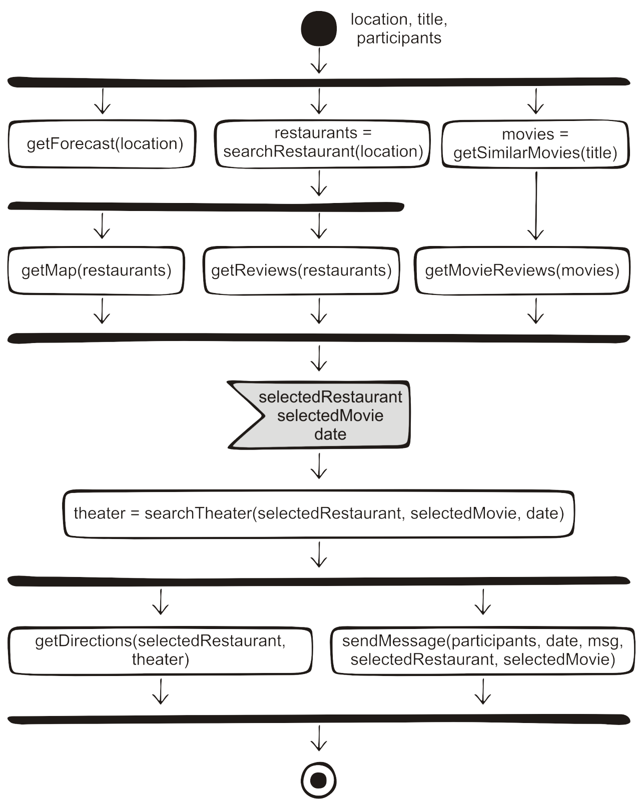

location, title, participants
getForecast(location)
restaurants = searchRestaurant(location)
movies = getSimilarMovies(title)
getMap(restaurants)
getReviews(restaurants)
getMovieReviews(movies)
selectedRestaurant selectedMovie date
theater = searchTheater(selectedRestaurant, selectedMovie, date)
getDirections(selectedRestaurant, theater)
sendMessage(participants, date, msg, selectedRestaurant, selectedMovie)

Figure 5.1: ProgrammableDinner Orchestration Schema

accomplished in several ways, although there is some preferred ordering among the different actions that build the orchestration, *e.g.,* the choice of the movie has to precede the choice of the theater. The UML Activity Diagram in Figure 5.1 (input activities in gray), models these precedence relationships.

To implement PD we may exploit the set of existing, publicly available Web services listed in Table 5.1. In most cases there are different alternatives for the same required feature that are functionally equivalent, while in one case (the `searchRestaurant` action) we have two alternatives that differ in their parameters (*i.e.,* coordinates or location name). Each service in the table may be associated to QoS data like response time (T) and reliability (R). These data may come from SLAs provided by service providers (as for the Azure Service Bus which reports in its SLA a reliability equal to 99.95%)[1] or they may come from direct measurements and estimates (*e.g.,* [38, 39]).

[1]http://www.windowsazure.com/en-us/support/sla/

Action	**Return Value**	**Possible Providers**
getMap	A map of a location	Google, Bing, MapQuest
getDirections	Directions to a destination	Google, MapQuest, Bing
getForecasts	Weather forecast	Wunderground, World Weather
searchRestaurant	List of restaurants given a location	Yahoo!, Google, CityGridMedia
searchTheater	List of theaters given a location and a movie	Google, Yahoo!, CityGridMedia
getReviews	Restaurant reviews	Yelp, CityGridMedia
getSimilarMovies	Similar Movies from a movie title	Rotten Tomatoes, TasteKid
getMovieReviews	Reviews of a list of movies	Rotten Tomatoes, NY Times Movie Reviews API
sendMessage	None	Nexmo, Hoiio

Table 5.1: Programmable Dinner Composed Services

```
@WebService
public interface ProgrammableDinner {
 @ReturnValue('recommendation')
 public Recommendation start(
  @WebParam(name='desiredLocation')
  String location,
  @WebParam(name='movie')
  String similar_movie,
  @WebParam(name='participants')
  List<Participant> friends);
}
```

Listing 5.1: The ProgrammableDinner orchestration interface

The ProgrammableDinnner Orchestration Model

Before introducing the new features included to support QoS attributes, let us describe how PD is implemented in DSOL.

Orchestration Interface

Listing 5.1 shows how the orchestration interface is defined for the PD orchestration. As for the requirements, it receives as arguments the desired location for the event, the movie to be used as reference to suggest another movies, and the list of participants.

Orchestration Goal and Initial State

Listing 5.2 shows the initial state, which already includes those facts automatically generated by DENG, and the goal for the first part of PD (before asking the user to select the restaurant, the movie and the date).

```
start  desiredLocation(location), movie(similar_movie),
        participants(friends)

goal  forecast(forecast_info, location) and
      list_of_restaurants(nearby_restaurants) and
      reviewsIncludedTo(nearby_restaurants) and
      mapWithMarkers(restaurants) and
      list_of_movies(similar_movies) and
      reviewsIncludedTo(similar_movies)
```

Listing 5.2: Initial state and goal

Abstract Actions

Listing 5.3 illustrates some abstract actions present in the PD scenario. Notice how they closely reflect the activities described in Figure 5.1, leaving out all the implementation details, *i.e.,* which service of Table 5.1 to be invoked, and also the actual execution sequence. Notice also the use of the seam action `getAddress` to help the planner to deduce the fact that the desired location is also an address, so it can be "geocoded".

Using the specified abstract actions, the initial state and the orchestration goal the DSOL runtime system is able to build two distinct plans as shown in Listing 5.4 and Listing 5.5. Indeed, both plans include a list of abstract actions that satisfies the goal in Listing 5.2. Note that, although their are functional equivalent, their structure is slightly different. The first plan search restaurants using the location as informed by the user, while the second had to geocode it and search restaurant using the coordinate of the specified location.

Concrete Actions

Listing 5.6 illustrates the concrete actions from the PD model that implements the `getForecast` abstract action. Note how service and generic actions are mixed together to guarantee that both concrete actions return compatible objects.

5.2 Adding QoS to DSOL

QoS-aware orchestration infrastructures must consider the following non-functional aspects: (i) choosing the best services to be orchestrated, and (ii) designing the orchestration structure based on the available services.

To achieve these objectives, we had to include new features to both DSOL language and run-time system. In particular, the language now includes the following constructs:

```
action getForecast(Location)
pre: desiredLocation(Location)
post: forecast(forecast_info, Location)

action searchRestaurants(Location)
pre: searchLocation(Location)
post: list_of_restaurants(nearby_restaurants)

action searchRestaurantsByCoordinate(Coordinate)
pre: coordinate(Coordinate)
post: list_of_restaurants(nearby_restaurants)

action getRestaurantReviews(Places)
pre: list_of_restaurants(Places)
post: reviewsIncludedTo(Places)

action getSimilarMovies(MovieTitle)
pre: movie(MovieTitle)
post: list_of_movies(similar_movies)

action getMovieReviews(Movies)
pre: list_of_movies(Movies)
post: reviewsIncludedTo(Movies)

action createMapWithMarkers(Places)
pre: list_of(Places)
post: mapWithMarkers(Places)

seam action getAddress(Location)
pre: desiredLocation(Location)
post: address(Location)

action geoCode(Address)
pre: address(Address)
post: coordinate(address_coordinate)
```

Listing 5.3: ProgrammableDinner Abstract actions

- The `@QoSProfile` annotation allows concrete actions to be augmented with their QoS profiles. It requires the orchestration designers to specify, for each annotated concrete action, the quality metrics it contains and also its correlated expected value. As previously mentioned, these data are obtained by direct measurements or declared by the service provider as part of a service contract (SLAs). Listing 5.7 shows the first concrete action of Listing 5.4 with information about its expected reliability, *e.g.,* 99.5%, and average response time, 300ms.

- The goal definition is now extended to include also the non-functional requirements of the orchestration. Listing 5.8 illustrates the goal definition corresponding to requirements $R6$ and $R7$. Notice that to cope with conflicting QoS requirements, we may specify

```
getForecast (location)
searchRestaurants (location)
getSimilarMovies(similar_movie)
addReviewsTo(nearby_restaurants)
getMovieReviews(similar_movies)
createMapWithMarkers(nearby_restaurants)
```

Listing 5.4: A possible plan for the ProgrammableDinner example

```
getForecast (location)
geoCode (location)
searchRestaurantsByCoordinate (address_coordinate)
searchRestaurants (location)
getSimilarMovies(similar_movie)
addReviewsTo(nearby_restaurants)
getMovieReviews(similar_movies)
createMapWithMarkers(nearby_restaurants)
```

Listing 5.5: Another possible plan for the ProgrammableDinner example

both desired bounds and preferred optimizations. Depending on the QoS metric, the desired bounds represent the lower (*e.g.,* for reliability) or upper (*e.g.,* for response time) bounds that the orchestration must meet. In our example, according to $R6$, we want a reliability of at least 0.99 and a response time of at most $1.5s$. Since different configurations could meet these bounds, we may also ask to optimize against a specific metric by using the `min` and `max` keywords. In our example, according to $R7$, we ask to minimize the response time.

These enhancements in the modeling language reflect to changes in the run-time system. Indeed, the original DSOL Planner (as introduced in Chapter 3.1) generates all possible plans that satisfy the functional requirements of the orchestration, while the Interpreter chooses non-deterministically one of them and executes it, binding each abstract action to one among the available concrete actions. At this point, DSOL has to change this behavior to consider QoS, since (1) the structure of plan to be executed and (2) how the chosen abstract actions are bound to concrete actions (*i.e.,* services) plays an important role on QoS management. In particular, DSOL now formalizes the problem of finding the plan to run and the concrete actions to bind as an *optimization problem*, and it exploits *linear programming techniques* [40] to solve it.

```
@Action(name = "getForecast")
@ReturnValue("forecast_info")
public List<Forecast> WundergroundServiceWrapper(String location) {
 WundergroundResults results = getForecastWithWunderground(location);
 List<ForecastDay> result = results.getForecast()
                              .getSimpleforecast().getForecastday();
 List<Forecast> forecasts = new ArrayList<Forecast>();
 for (ForecastDay forecast : result){
   forecasts.add(new Forecast(forecast.getConditions()));
 }
 return forecasts;
}
@Action(name = "getForecast")
@ReturnValue("forecast_info")
public List<Forecast> WorldWeatherOnlineServiceWrapper(String location) {
 WorldWeatherOnlineResults results =
                            getForecastWithWorldWeatherOnline(location);
 List<Weather> result = results.getData().getWeather();
 List<Forecast> forecasts = new ArrayList<Forecast>();
 for (Weather weather : result);
   String value = weather.getWeatherDesc().get(0).getValue();
   forecasts.add(new Forecast(value));
 }
 return forecasts;
}

@Action(service = "worldweatheronline")
public abstract WorldWeatherOnlineResults
getForecastWithWorldWeatherOnline(String location);

@Action(service = "wunderground")
public abstract WundergroundResults
getForecastWithWunderground(String location);
```

Listing 5.6: Concrete actions

```
@Action(service = "wunderground")
@QoSProfile(metrics={"reliability", "response_time"}, values={0.995,300})
public abstract WundergroundResults
getWeatherForecastWithWunderground(String location);
```

Listing 5.7: Concrete actions with QoS Profile

The optimization problem includes:

- A set of abstract actions $A = \{a_1, \dots, a_n\}$ where $n \geq 1$.
- For each abstract action a_i, a set of concrete actions $C_i = \{c_{i,1}, \dots, c_{i,m_i}\}$ where $m_i \geq 1$. Each concrete action $c_{i,j}$ is characterized by a response time $t_{i,j} > 0$ and a reliability $r_{i,j} \in [0, 1]$.
- A set of plans $P = \{P_1, \dots, P_l\}$ where $l \geq 1$, each modeled as a set of abstract actions.

```
goal RQ1 and RQ2...RQ5
and desired(reliability, 0.99) and desired(response_time, 1500)
and min(response_time)
```

Listing 5.8: Goal Definition Including QoS Requirements

We define a *binding variable* $s_{i,j,x} \in \{0,1\}$ (where $i \in [1,n]$, $j \in [1,m_i]$, and $x \in [1,l]$) to indicate if the concrete action $c_{i,j}$ is bound to the abstract action a_i in plan P_x (*i.e.,* $s_{i,j,x} = 1$) or vice versa (*i.e.,* $s_{i,j,x} = 0$). This assignment must meet the following constraints:

$$(\forall x, i | a_i \in P_x) \sum_{0<j\leq m_i} s_{i,j,x} \leq 1 \tag{5.1}$$

$$(\forall x, i | a_i \notin P_x) \sum_{0<j\leq m_i} s_{i,j,x} = 0 \tag{5.2}$$

$$(\forall x, i, j | s_{i,j,x} \neq 0) \rightarrow (\forall x', i', j' | x' \neq x)\ s_{i',j',x'} = 0 \tag{5.3}$$

Equation 5.1 and 5.2 together indicate that we bind at most one concrete action to every abstract action part of a plan, while we leave unbound those abstract actions that are not part of a plan. Equation 5.3 indicates that if we bind a concrete action in a plan x, we cannot bind any action in plans different from x. In other words we bind one and only one plan. Accordingly, a valid assignment to binding variables $s_{i,j,x}$ returns a single plan bound to a set of concrete actions.

Given these definitions, our optimization problem boils down to find the *optimal assignment* to $s_{i,j,x}$, *i.e.,* the assignment corresponding to a bound plan that satisfies QoS requirements. This can be formalized by introducing two aggregation functions f_R and f_T, for reliability and response time, respectively.[2] In particular, f_T is the sum of all $t_{i,j}$ of each concrete action such that $s_{i,j,x}$ is set to one:[3]

$$f_T = \sum_{\forall i,j,x} s_{i,j,x} \times t_{i,j}$$

Similarly, f_R aggregates reliability by multiplying $r_{i,j}$ to the power of $s_{i,j,x}$:

$$f_R = \prod_{\forall i,j,x} r_{i,j}^{s_{i,j,x}}$$

[2]In presence of other, user defined metrics, we add similar functions.

[3]For parallel actions, DSOL considers only the slowest one.

Given these two aggregation functions and indicating with G the set of all possible assignments to binding variables $s_{i,j,x}$, we may define the optimization problem looking at the goal definition. For example, recalling Listing 5.8, we have that the optimal assignment is defined as follows:

$$\begin{aligned} \underset{\forall g \in G}{\text{minimize:}} \quad & f_T, \\ \text{subject to:} \quad & f_T < 1500ms \\ & f_R > 0.99 \end{aligned}$$

Since f_R is non linear with respect to the optimization variables $s_{i,j,x}$, we linearize it using a typical technique [40], which applies the logarithm to both sides of the equation: $\log(f_R) > \log(0.99)$.

The optimization problem above is solved by the DSOL Interpreter, which internally relies on AMPL [41] to find the optimal assignment to $s_{i,j,x}$, *i.e.,* to choose the plan to execute and how to bind concrete actions to abstract ones.

At run-time, DSOL applies two additional mechanisms: (1) *Adaptive Re-Binding* and (2) *Adaptive Re-Planning*, to further optimize the QoS perceived by the end user based on the actual situations encountered. Next paragraphs illustrate these two mechanisms referring to the PD example.

5.3 Maximizing Performance via Adaptive Re-binding

In the case the orchestration goals specify that the response time must be minimized, the DSOL engine does not limit to blindly execute the plan returned by the Optimizer, but it applies an *Adaptive Re-Binding* strategy. With this term we indicate the fact that it evaluates, at every step of execution, alternative bindings to abstract actions that could decrease the expected response time of the orchestration. If a better alternative (*i.e.,* a faster concrete action) is found, the engine re-binds the current action automatically.

Notice that better alternative actions may be actually found since, even if the condition of optimality concerning the bindings produced by the Optimizer holds before starting the execution of the plan, it may not hold anymore during its execution. Indeed, during execution, the QoS values associated to concrete actions can be updated. In particular, the reliability of actions already performed can be set to one.[4] Increas-

[4]If a concrete action appears more than once in a plan, we set reliability equal to one only for the invocations already occurred.

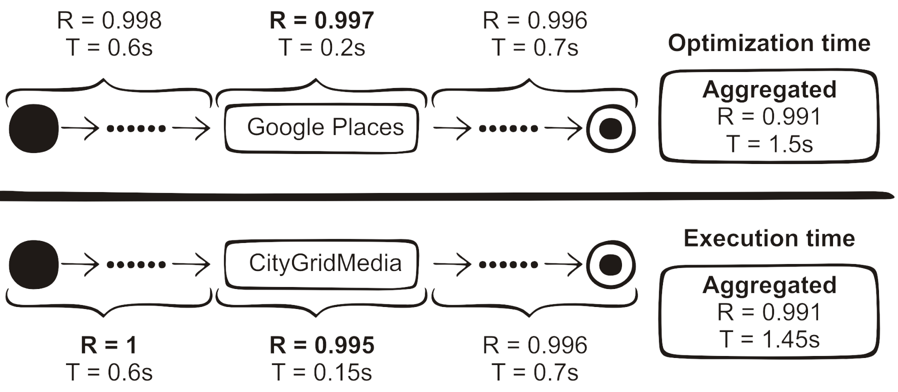


Figure 5.2: Adaptive Re-Binding Example

ing the reliability of certain actions implies that other concrete actions, initially discarded by the Optimizer because too unreliable to reach the specified bound, may now become eligible for execution, and they could be actually chosen if lead to a lower response time. In other words, their high probability of failure was compensated by the already successfully executed actions, and they can now be taken into consideration.

Let us illustrates this scenario by recalling our PD example. Let us imagine that the DSOL engine has already executed part of the plan: the next action to execute is the `searchTheater` operation. As reported in Table 5.1 we have three alternatives to implement this step and let us assume Google was the concrete action chosen by the Optimizer. We can decompose the overall reliability of the plan in three contributes. The first regards actions already performed, the second is the reliability of the concrete action currently bound to `searchTheater` (*i.e.,* Google), and the third is the expected reliability of concrete actions to be executed after `searchTheater`. If the first contribute was equal to 0.998 at optimization time (see Figure 5.2), at run-time, as it was already successfully executed, we may assume it as being 1. This enables the CityMediaGrid concrete action: it has a lower reliability than Google, but now that we updated the reliability of the preceding actions it is reliable enough to reach *RQ*6. Being faster than Google it becomes the best choice to minimize the response time, *i.e.,* to satisfy *RQ*7.

This kind of reasoning is performed by DSOL at every step of the plan. Indeed, this is a fast search (we proceed incrementally, focusing only on the next abstract action to execute) that may considerably improve the overall response time of the orchestration, especially in presence of very efficient but unreliable services.

5.4 Maximizing Reliability in Presence of Failures

If we focus on reliability, we may observe that in defining our optimization problem we did not take into consideration the DSOL ability to adapt the orchestration at run-time, re-binding faulty actions to alternative services and, if this was not enough, re-building the entire plan to circumvent multiple failures. This choice is motivated by the impossibility to correctly estimate and account how this re-binding and re-planning mechanisms (to improve reliability) may impact response time.[5] This means that the optimal plan initially found by the Optimizer, including the mapping to concrete actions, is expected to provide the desired reliability by itself. In our PD example, the optimal plan coming from the Optimizer will succeed 99% of the times ($RQ6$).

On the other hand, the adaptive features of DSOL are there, and they can be used to improve the reliability perceived by the end-user. In practice, in presence of a faulty service, DSOL re-binds the corresponding abstract action to an alternative concrete action. This way, an invocation that should fail can be saved and terminate correctly, contributing to increase the overall reliability.

Moreover, if all alternatives fails, DSOL builds an alternative plan that skips the failed action (eventually compensating already performed actions that are not part of the new plan), thus enabling new opportunities to end the orchestration correctly. In this re-planning case, the Optimizer is invoked again to choose the new optimal plan among the alternative ones.

It is important to notice that in all these cases response time requirements are not guaranteed anymore. The engine is doing its best to overcome a failure and the completion of the orchestration is the current priority. On the other hand, we do not ignore response time. At re-binding time DSOL chooses alternative actions ordered by response time, while at re-planning time, it runs the Optimizer to choose the best plan also considering response time.

Finally, it is important to notice that DSOL updates automatically the QoS attributes of concrete actions based on their observed behavior. This implies that two subsequent invocations to the same orchestration may be served by different services or different plans if the conditions change between the first and the second invocation. For example, a variation on the response time or reliability of the concrete actions used in the plan in Listing 5.4 may affect its overall non-functional profile what is automatically perceived by the Optimizer, and in the case this

[5]Indeed, re-planning and re-binding may require compensating one or more actions, which further impact execution time.

plan was previously the one with higher probability to satisfy the non-functional goal, it may not be anymore, what means that now the first choice is the plan in Listing 5.5.

5.5 Domain Specific QoS Metrics

So far we only considered reliability and response time as QoS metrics. However, orchestration designers may need to consider other QoS aspects, such as costs or availability, to model domain specific requirements. To cover these scenarios, DSOL allows domain specific QoS metrics to be easily defined.

Domain Specific QoS Metrics (DSQM) are characterized by their name (*e.g., "cost"* or *"availability"*) and by two *aggregation operators*: $\langle s, p \rangle$, which are used by DSOL to calculate the DSQM value for an entire plan starting from the DSQM value of each action. In particular, both s and p can be algebraic operators $(+, -, \times, /)$ or simple functions (*min*, *max*, *abs*, *avg*). Operator s indicates to DSOL how to aggregate the value of actions executed in sequence, while operator p indicates how to aggregate parallel actions. As an example, $\langle +, max \rangle$ are the operators for the predefined response time metric, while $\langle \times, \times \rangle$ are those for reliability. The definition of DSQM is characterized also by the *metric limit* which may be *upper* or *lower*. The former indicates that the optimal plan should have *at least* the desired value expressed by the goal definition, viceversa the latter indicates that the desired value will be considered as a maximum limit for the DSQM.

The introduction of DSQM affects both DSOL language and run-time system. Let us start from the language with an example that extends the PD orchestration adding the following requirement:

- *RQ*8: The total cost for executing the orchestration should not exceed 3$.

Imagine that both the `sendMessage` and `getForecast` services charge a small fee for each invocation, the exact amount depending from the service provider. To satisfy *RQ*8 we define the DSQM *"cost"* (C) as reported in Listing 5.9. It uses the sum as aggregate operator both for sequential and parallel actions.

```
define(cost, aggregation<+, +>, limit<upper>)
goal RQ1 and RQ2...RQ5
and desired(reliability, 0.999) and desired(response_time, 1500)
and desired(cost, 3) and min(response_time)
```

Listing 5.9: DSQM Definition

Given a DSQM, at run-time DSOL behaves as explained in the previous sections except for the optimization problem, which now includes additional aggregation functions, similar to f_R and f_T and defined using the respective aggregation operators. For the cost example, this means adding the aggregation function f_C, defined using the sum to aggregate both sequential and parallel actions. We use it, together with the definition of the metric limit, to define an additional constraint $f_C < 3$ starting from the goal in Listing 5.9.

5.6 Performance Assessment

To validate the new features presented in this chapter we have extended DSOL evaluation in a two step approach. First, it evaluates the overhead imposed by the optimizer to find the optimal mapping between concrete and abstract actions.[6] Secondly, it investigates the potential speed-up it provides.

Our experiments were carried out in a local server configured to emulate a typical application server used to deploy service-based applications. Such server had the following configuration: Intel Core i5 processor, 4GB RAM and Ubuntu Linux (version 11.10) operating system.

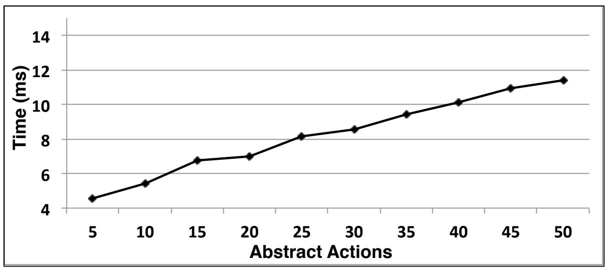
5.6.1 DSOL + QoS Overhead

Determining the optimal plan given the set of available actions and the orchestration goal, introduces an overhead at execution time, which we measured as follows. First of all we measured the overhead considering plans of variable dimensions in terms of the number of abstract actions considered. Figure 5.3(a) reports the optimization time for plans composed by an increasing number of abstract actions. In this experiment, each abstract action has three alternative concrete actions. The measured optimization time account for less than $12ms$ for a plan comprising fifty abstract actions, which is a perfectly acceptable overhead considering that the resulting orchestration would probably involve a number of service invocations close to fifty,[7] which needs seconds to execute.

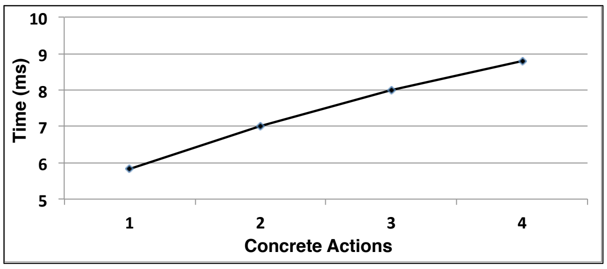
The second experiment we made kept constant the number of abstract actions (*i.e.,* the plan length) increasing the number of concrete alternatives associated to each abstract actions. Figure 5.3(b) reports the results obtained with a plan composed of twenty abstract actions. Even

[6]This overhead does not include the time required to find the possible plans, as such time is strongly related to the abstract actions domain. For further details we refer to Chapter 4.

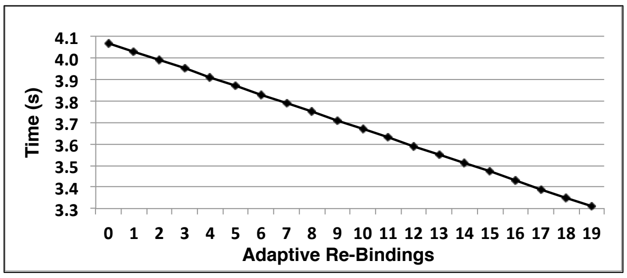
[7]Most DSOL actions results in invoking an external service.



(a) Optimization Time over Abstract Actions.



(b) Optimization Time over Concrete Actions



(c) Execution Time over Re-Bindings.

Figure 5.3: DSOL + QoS Validation

in this case the optimization time is negligible w.r.t. the plan execution time: less than $9ms$ in the worst case of four concrete alternatives for each abstract action. Note that, in practice, the number of alternative concrete actions that can be found for each abstract action is typically small, for example, in our PD orchestration we was not able to find more than three alternatives for each action.

5.6.2 DSOL + QoS Speed-Up

Once we verified that our general optimization strategy introduces a negligible overhead w.r.t. a non optimized solution, we were interested in measuring the impact on performance of our mechanisms.

We started measuring the speed-up gained through the adaptive re-binding mechanism described in Section 5.3. We considered a plan composed by twenty abstract actions with two concrete actions each. The first is selected at optimization time while the second becomes the more convenient option at execution time because it has a response time 20% faster and its lower reliability is compensated by the already executed actions. In absence of re-binding we measured an average execution time for the orchestration of $4.069s$. Then we activated the re-binding mechanism and let it run a growing number of times, from one to 19 (we have a total of 20 actions). The results we measured are reported in Figure 5.3(c). We notice that we gain a linear speedup which is maximum when we let DSOL re-bind every possible action. The speedup we measure in this case is 18.62%. This is very close to 20%, which is the maximum theoretical speedup we could obtain under this scenario. This demonstrates that the advantages of the re-binding mechanism come at a negligible cost: the difference between 18.62% and 20%, which is the overhead of finding the best alternative and re-binding it.

The last test we performed was to measure the overall speed-up of DSOL w.r.t. a non optimizing solution. To do so, we took our ProgrammableDinner example and measured the actual (average) execution time of all the alternative services that can be used to compose the orchestration. In absence of an optimizer like the one integrated in DSOL we could expect that a standard engine (like DSOL) takes a random plan, so we measured the average execution time of all possible plans: it takes $2s$ to complete. Then we let DSOL run the orchestration asking it to minimize the execution time. The result we measured was $1.2s$, which corresponds to a speed-up of 40%.

In general, those results demonstrate that the optimization problem solved by DSOL and the adaptive re-binding mechanism it implements, introduce a limited and negligible overhead with respect to the execution of the entire orchestration, while providing a significant potential speed-up in terms of response time (not to mention the advantages in terms of reliability and the fact that it provides a guaranteed QoS as specified by the user in the goal).

5.7 Conclusion

In this chapter we have presented how DSOL was extended to support quality requirements and adaptivity. We introduced an optimizer which generates an optimal solution in terms of execution plan and bindings. Furthermore, we provided a novel solution that further optimizes response time and reliability via adaptive re-binding and re-planning. We discussed all these features relying on a service orchestration built out of real publicly available services. Finally, we measured the overhead of our approach and the potential speed-up it may offer to orchestrations to demonstrate the pro and cons of the proposed extensions.

The next chapter presents the use of DSOL in the development of mobile applications, a different environment when compared to the one we addressed so far but which shares with the latter several commonalities like the unpredictability of the context in which applications are deployed, and how they combine different third party functionalities.

6 DSOL in Motion

The recent massive adoption of mobile devices—such as smartphones and tablet PCs—is coercing people to increasingly depend on them and on the services they provide even for the most ordinary daily activities. Looking from that perspective, 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, mainly because of all 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 operation systems, 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 developed 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, 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. This difference, albeit apparently minor, if left unmanaged may impact the application's ability to satisfy its requirements.

To cope with these peculiarities, apps also need to be *adaptive* ([9, 10]), both because of the heterogeneous environments they are deployed in and because of the external services and apps they rely upon. As for traditional service orchestrations, engineers develop such kind of adaptive mobile application by explicitly programming 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 all 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.

We address this issue by proposing an implementation of DSOL that targets the development of mobile applications which we refer to as SELFMOTION.[1] Using SELFMOTION, developers are forced to abandon the mainstream imperative path in favor of a strongly declarative alternative, which allows mobile 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*, 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).

[1]Self-Adaptive Mobile Application.

SelfMotion apps are executed by a client-side middleware that leverages an internal planner to elaborate, at run-time, the best sequence of abstract actions to achieve the goal, based also on the current deployment context, *e.g.,* available sensors and device's hardware. The mapping between abstract and concrete actions is done, then, in accordance with a previously specified QoS Policy. Furthermore, the SelfMotion middleware works similarly to the DSOL engine: whenever an unexpected situation is faced, *e.g.,* a third-party app is not installed in the device, which prevents successful completion of the elaborated plan of execution, the middleware automatically, and transparently to the developer and to the app's user, builds an alternative plan toward the goal and continues executing the app, which results in a nice and effective self-healing behavior.

In this chapter we describe such implementation approach in details, and we show, through a set of experiments, its effectiveness and its performance, showing how the approach based on a client-side embedded planner scales well even when the goal becomes complex and requires, to be satisfied, several activities to be called in the correct order.

In particular, for a clear and effective explanation of SelfMotion we rely on a realistic mobile app illustrated in Section 6.1 and used as a reference example throughout the chapter. The SelfMotion approach is described in detail in Section 6.2, while Section 6.3 discusses its advantages with respect to the state of the art. Finally, Section 6.4 evaluates the performance of SelfMotion in several scenarios of growing complexity while Section 6.5 draws some conclusions.

6.1 A Motivating Example: The ShopReview App

Let us now introduce *ShopReview* (SR), the mobile app we will use throughout this chapter. 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, a 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.

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

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 6.1: ShopReview functionality.

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 6.1 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 6.1 describes a possible implementation of the described adaptive behavior for the Android platform ([7]). 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 6.1 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.

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]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.

```
PackageManager mng = getPackageManager();
if(mng.hasSystemFeature(PackageManager.FEATURE_CAMERA_AUTOFOCUS)){
  //Run local barcode recognition
}else{
  //Invoke remote service with blurry decoder algorithm
}

Location location = null;
if(mng.hasSystemFeature(PackageManager.FEATURE_LOCATION_GPS)){
  LocationProvider provider = LocationManager.GPS_PROVIDER;
  LocationManager locManager =
        (LocationManager) getSystemService(Context.LOCATION_SERVICE);
  try{
    //Return null if the GPS signal is currently not available
    location = locManager.getLastKnownLocation(provider);
  }catch(Exception e){
    location = null;
  }
}

if(location==null){
  //Device whitout GPS or an excpetion was raised invoking it.
  //We show up a map to allow the user to indicate
  //its location manually
  showMap();
}
```

Listing 6.1: Adaptive Code Example.

6.2 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.

6.2.1 Introducing SelfMotion

To support the development of *adaptive* mobile applications SelfMotion follows the same declarative approach proposed by DSOL, including several steps both at design and run-time. At design-time, it supports the work 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 an execution environment, which uses planning techniques to reach the app's goals, adapting to the different, expected or unexpected, situations that may be encountered.

Although SelfMotion follows the main ideas of DSOL, it includes some particular concepts exclusive to mobile domains. More specifically, as shown in Figure 6.1, a SelfMotion application includes:

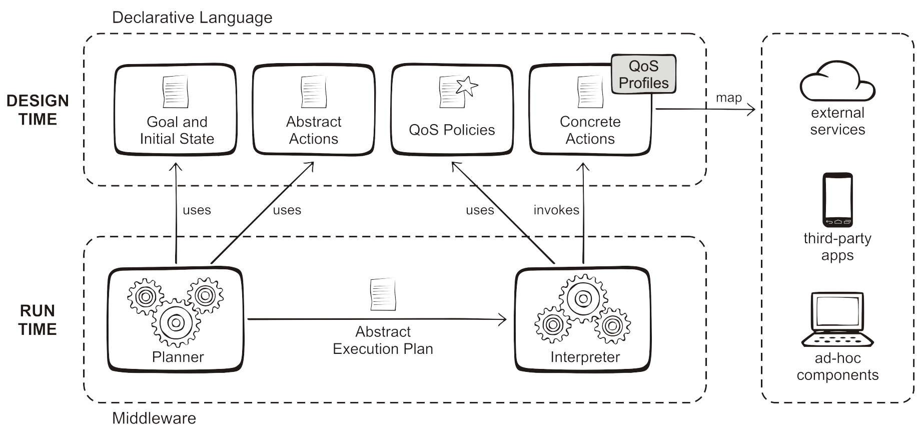


Figure 6.1: SELFMOTION Conceptual Architecture.

- the app's *Goals*, expressed as a set of facts that are required to be fulfilled by the app's execution;
- the *Initial State*, which models the set of facts one can assume to be true at app's invocation time. Moreover, 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;
- a set of *Abstract Actions*, which specify the primitive operations that can be executed to achieve the goal;
- 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 ad-hoc component or an installed third-party application);
- a *QoS Profile* for each concrete action, which models its non-functional characteristics (*e.g.,* energy and bandwidth consumption);
- 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.

6.2.2 The SelfMotion Declarative Language

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

Abstract Actions

Following the DSOL concepts, abstract actions are high-level descriptions of the primitive actions available to accomplish the app's goal. They represent the main building blocks of the app. Listing 6.2 illustrates the abstract actions for the SR reference example: they correspond to the high level functionalities listed in Table 6.1. 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.

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 6.3) 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.

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

action   enableGPS
pre  :   ~isGPSEnabled
post:    isGPSEnabled

action   getPositionManually
pre  :   true
post:    position(userDefinedPosition)

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

action   acquirePhoto
pre  :   hasCamera
post  :  image(barcodeImage)

action   readBarcode(Image)
pre  :   image(Image), hasAutoFocusCamera
post  :  barcode(productBarcode)

action   blurryReadBarcode(Image)
pre  :   image(Image), hasFixedFocusCamera
post:    barcode(productBarcode)

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

action   searchTheWeb(Name)
pre  :   productName(Name)
post  :  prices(onlinePrices)

action   searchTheNeighborhood(Barcode, Position)
pre  :   barcode(Barcode), position(Position)
post  :  prices(localPrices)

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

Listing 6.2: ShopReview Abstract Actions.

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 6.2 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 device does not have a GPS sensor.

```
prices(localPrices) and prices(onlinePrices) and
priceShared and position(gpsPosition)

or

prices(localPrices) and prices(onlinePrices) and
priceShared and position(userDefinedPosition)
```

Listing 6.3: ShopReview Goal.

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

Listing 6.4: ShopReview Initial State.

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 6.4.

Concrete Actions

As for DSOL, concrete actions are the executable counterparts of abstract actions. For example (see Table 6.3), 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), or cannot be executed (*e.g.,* an external required app is not installed in the device) by invoking an alternative concrete action.

As the current SelfMotion prototype was developed for the Android platform, concrete actions are also implemented through Java methods, supporting the same ad-hoc annotations we introduced previously. Recalling some concepts, we use the annotation `@Action` to refer to the implemented abstract action, as in Listing 6.5, which shows the three concrete actions that implement the `getProductName` abstract action.

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

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 6.3. 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 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 6.5. 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

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 (`searchupc.com`) Web service (`simpleupc.com`) Ad-hoc Component (textual input from the user)
searchTheWeb	Web service (`kelkoo.it`) Web service (`buscape.com`)
searchTheNeighborhood	Web service (`shopsavvy.mobi`)
sharePrice	Third-party app (`ubersocial.com`) Third-party app (`twicca.r246.jp`) Web service (`dev.twitter.com`)

Table 6.3: ShopReview Concrete Actions.

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

```
@Action(name="getProductName")
@ReturnValue("name")
@QoSProfile(metrics={"usability", "cost", "energy"},
            values={1,0,-1})
public String getProductNameViaSearchUPC(Barcode barcode){
  String barcodeValue = barcode.getValue();
  //Invoke http://searchupc.com/
  String productName = searchupc(barcodeValue);
  return productName;
}

@Action(name="getProductName")
@ReturnValue("name")
@QoSProfile(metrics={"usability", "cost", "energy"},
            values={1,1,-1})
public String getProductNameViaSimpleUPC(Barcode barcode){
  String barcodeValue = barcode.getValue();
  //Invoke http://simpleupc.com/
  String productName = simpleupc(barcodeValue);
  return productName;
}

@Action(name="getProductName")
@ReturnValue("name")
@QoSProfile(metrics={"usability", "cost", "energy"},
            values={-1,0,0})
public String getProductNameFromUser(Barcode barcode){
  String barcodeValue = barcode.getValue();
  //Ask the user for the product name
  String productName = ...;
  return productName;
}
```

Listing 6.5: `getProductName` Concrete Actions.

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.

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. Note that, as we are running in an environment with limited resources, we decided to follow a different strategy to QoS management when compared to the one adopted by DSOL execution engine which leverages an linear optimizer to find the best binding between abstract and concrete actions.

```
qos: default
pre: ~lowBattery
max: usability
min: cost
min: energy

qos: energySaver
pre: lowBattery
min: energy
max: usability
min: cost
```

Listing 6.6: QoS Policy Definitions.

Indeed, as explained hereafter, in SelfMotion we use QoS attributes to select the best concrete action looking only to the current step of the plan.

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 Figure 6.6, which reports two possible QoS policies for the SR example: `default` and `energySaver`. Imagine that the middleware, at start-up, set in the initial state the fact ∼`lowBattery`, indicating that the battery is well charged. In this case, the first policy with a valid pre-condition is `default` and, as a consequence, the SR application will be executed using this specific policy. In particular, `default` is composed by three ordered constraints: (1) `max: usability`, (2) `min: cost`, and (3) `min: energy`. The three constraints will be applied in order. Every time the Interpreter must execute an abstract action with many corresponding concrete actions, it will invoke the one with maximum usability. If this criterion does not result in the selection of an unique concrete action (*i.e.,* many actions have the same maximum usability value), the Interpreter applies the second constraint (*i.e.,* the minimum cost) to the set of actions with the maximum usability. If even this criterion is not able to identify an unique candidate, the Interpreter applies the third constraint (*i.e.,* minimum energy). If neither this is enough to find an 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.

Given these premises, if we consider, for example, the `getProductName` abstract action, with its concrete counterparts reported in Listing 6.5, 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.

Summing up, by specifying one or more QoS policies developers encode a hierarchical system of priorities among available concrete actions, which in turn allow an adaptive behavior of the resulting app, as discussed later on in Section 6.3.2.

6.2.3 The SelfMotion middleware

As previously introduced, the SELFMOTION middleware is in charge of executing the app. 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.

If something goes wrong during this process (*e.g.,* an ah-hoc component 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 charge of automatically determining the sequence of steps that satisfies the goal under the circumstances and the deployment context actually faced at run-time.

```
acquirePhoto
blurryReadBarcode(barcodeImage)
enableGPS
getPositionWithGPS
getProductName(productBarcode)
inputPrice(name)
searchTheWeb(name)
searchTheNeighborhood(productBarcode, gpsPosition)
sharePrice(name, price)
```

Listing 6.7: A Possible Abstract Execution Plan.

Listing 6.7 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 far as the implementation is concerned, the current SelfMotion prototype uses a porting of the same ad-hoc planner we used for the DSOL engine to the Android platform.

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 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 6.4.

6.3 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 also to the SR example.

6.3.1 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 pre-condition—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. Then, 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.

6.3.2 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 6.1 and compare it with its SelfMotion counterpart, which relies on several abstract actions with different preconditions (see Listing 6.2). 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 6.2 and 6.5). 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 *re-planning* 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 6.7 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. The Planner is then 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 6.6. 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.

6.3.3 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* 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 lifecycle, which is crucial in the mobile domain.

6.4 Validating the SelfMotion Approach

To validate the SelfMotion approach, we implemented a publicly available open-source tool.[5] Although our approach is general and applies with limited technological modifications to several existing mobile frameworks, we focused on the Android mobile platform [7] for our prototype.

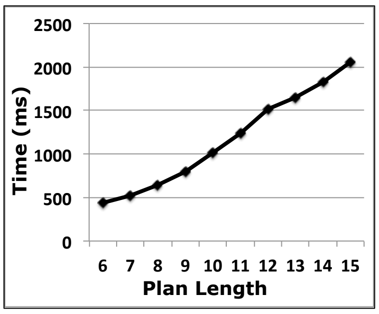
The initial validation we report in this section consists of a testing campaign we performed, exploiting the Android emulator as well as several real mobile devices, to measure the overhead introduced by SelfMotion w.r.t. conventional approaches. The experiments showed that this overhead exists but it 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: a *Samsung Galaxy SII*, which represents the typical Android-enabled device available today, and a netbook equipped with 1*GB* of RAM, an Atom processor, Debian Linux, and Sun Java Virtual Machine 1.5. 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 and the same amount of RAM. For the experiments involving a remote deployment, we installed the Planner on a remote server equipped with an *i7-2720qm* processor, 4*GB* of RAM, Debian Linux, and Sun Java Virtual Machine 1.5. 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.

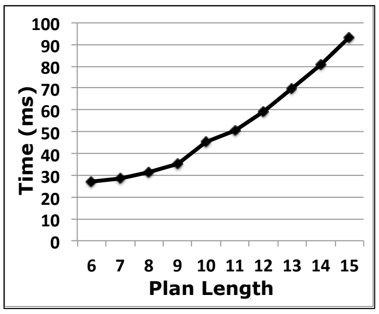
Moving from the consideration above, we started analyzing how the plan length impacts performance. In particular, we developed a scenario in which we had twenty abstract actions and a goal definition satisfiable through a plan composed of six of these actions. We measured the time needed to obtain the plan and we repeated the experiment changing the goal definition in order to obtain plans of increasing length—from

[5]`http://www.dsol-lang.net/self-motion.html` where the implementation the SR app can also be found.

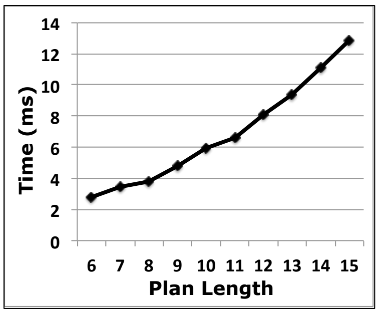
six to fifteen—recording the time needed to compute them, both with a local and with a remote deployment of the Planner. Figure 6.2(a) shows that, by running this testbed with a local Planner and with an initial plan including six actions, the Planner takes around 440*ms* to complete. The time needed to generate the plan gradually increases up to 2051*ms* for a plan that includes fifteen actions. Figure 6.2(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 6.2(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.



(a) Samsung Galaxy SII.



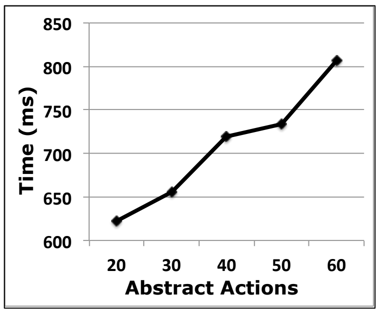
(b) Atom Platform.



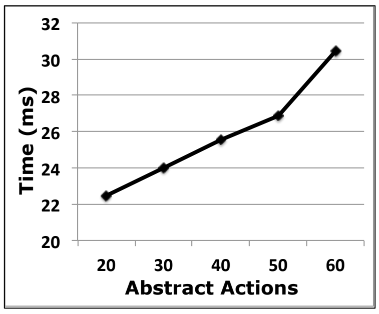
(c) Remote Evaluation.

Figure 6.2: Plan Generation Time over Plan Length.

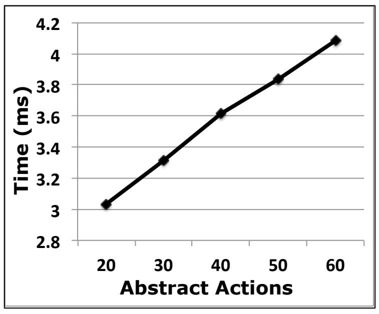
Our second test set focuses on the impact of the number of abstract actions on the plan generation time. For this we built a scenario in which there is an increasingly large set of abstract actions and a goal definition that generates a plan using eight of them. Figure 6.3(a) shows that, with ten abstract actions and a local deployment on the Samsung Galaxy SII, the SELFMOTION Planner takes about 622ms to complete. This time gradually increases up to 807ms when sixty abstract actions are available. As in the previous scenario, the Atom platform and the remote deployment provide further advantages, as reported in Figure 6.3(b) and 6.3(c).



(a) Samsung Galaxy SII.



(b) Atom Platform.



(c) Remote Evaluation.

Figure 6.3: Plan Generation Time over Abstract Actions.

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 a significant performance improvement may be achieved by introducing ad-hoc features, such as plan caching. Finally, we observe that our experiments considered plans of length up to fifteen and up to sixty 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 6.1. 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 $333ms$ (Samsung Galaxy SII), $55ms$ (Atom), and $6.6ms$ (remote execution).

Finally, we briefly report some considerations on local versus remote plan generation. The choice among them essentially depends on: (1) the number of abstract actions—which represent 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.

6.5 Conclusions

In this chapter, we presented SelfMotion, an incarnation of the DSOL declarative approach designed to support systematic development of mobile apps. In contrast to the approaches used by the most adopted mobile platforms, SelfMotion exploits automatic planning techniques to elaborate, at run-time, the best sequence of activities to achieve the app's goal.

SelfMotion contributes to the research in adaptive software systems and services by investigating a declarative approach for the effective and efficient development of adaptive apps conceived as hybrid compositions of services and components. Indeed, 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 example 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.

7 Related Work

In this chapter we provide a detailed description of some of the approaches available in the literature that are related to the work presented in this thesis. In the first part, we take a step back and introduce some works that have inspired us to propose DSOL. The second part discusses other alternative approaches which, similar to DSOL, also focus on tackling the complexity of defining service orchestrations, including quality-of-service management. Finally, we discuss some related work that focus on the effective and efficient development or mobile applications.

7.1 Looking to the Past to Take Inspiration for the Future

The problem of defining and managing service orchestrations in which architects have to deal with an intricate control flow trying to capture all possible ways things can go wrong and react to exceptional situations even in the presence of anticipated and unanticipated changes, *i.e.*, implement adaptive service orchestrations, has very strong similarities to what was discovered in the late 1980s and in the 1990s in the research area on *software processes*. This area was mostly boosted by Osterweil's seminal work [42]. Osterweil recognized the need to formalize the software development process so that it could be analyzed, improved, and automated. This area was sometimes referred to using the term *process programming*, although the low-level term "programming" does not do justice to the real essence of Osterweil's proposal. Rather, the idea was that software processes were important conceptual entities to understand, model, and possibly automate. Indeed, the same concept was later applied to other human-intensive domains besides software development, where the term *workflow* instead of *process* became more commonly used.

One of the important findings of the work on (software) processes was that because of the active and creative role of humans in the process, *deviations* [12] were important to handle [43, 44]. The software process, in fact, supports humans and manual activities as well as automated tools.

Unlike tools, humans cannot be seen as “subroutines" to invoke to get fully predictable results. Moreover, humans can tolerate inconsistencies, whereas tools seldom can. Finally, because processes are long-running entities, they need to evolve as the situation may change during the course of execution. Having recognized these distinctive feature, the process work in the 1990s sought ways to model flexible processes through sophisticated mechanisms and studied how to manage deviations and inconsistencies arising in the process enactment. This past work can be classified in three main directions:

7.1.1 Process programming with exceptions

A number of approaches investigated how to adapt the exception handling constructs that are supported by standard programming languages for inclusion in languages intended for process definition and automation. The emphasis here is on using a process language, as in Osterweil's original proposal, to program the process. Perhaps the most completely developed approach is the APPL/A language [43], which is based on an imperative paradigm. The idea of using exceptions has the obvious advantage that the *normal* process flows are clearly distinguishable from the *exceptional* flows in the process description. This allows for a certain degree of separation of concerns and supports a cleaner programming style than handling exceptional conditions through conventional *if–then–else* constructs. The main drawback of this approach is that it requires all possible exceptional conditions to be identified before writing the process code. This can be quite restrictive in highly dynamic contexts in which new and unanticipated cases may arise.

7.1.2 Reflective mechanisms

Through reflection, languages support reasoning about, and possibly modification of, programs. Reflective features are often available in conventional programming languages. They have been also proposed and experimented within process languages. As an example, on the SPADE environment [45], was developed a fully reflective process modeling language (SLANG) based on Petri nets, which allows meta-programming. That is, in SLANG one can develop a process whose objective is to modify an existing process or even an existing process instance. The potential advantage of such an approach over the previous one is clear: the process model does not need to anticipate all possible exceptional situations, since it can include the (formal) description of how the process model itself can be modified at execution-time to cope with unexpected situations. The main drawback of this approach is that it may bring further rigidity into the approach: not only the process must be modeled (or "programmed") in all detail, but so also must the meta-process, *i.e.,* the process of modifying the model itself.

7.1.3 Flexible approaches

Both previous cases are based on the assumption that a precise and enforceable process model is available and there is no way to violate the prescribed process. In other terms, there is no way to treat a deviation from the process within the formal system. Reflective languages support changes to the process, but all possible changes must follow a predefined change process, *i.e.,* again there is no way to "escape" from a fully defined, prescriptive model. The key idea to overcome this limitation was to abandon the ambitious but unrealistic goal of modeling every aspect of the process in advance, following an imperative, prescriptive style, to focus on certain constraints that should be preserved by the process, without explicitly forcing a pre-defined course of actions. Any process that satisfies the constraints would thus be acceptable. This brings a great flexibility in process enactment, avoiding micro-management of every specific issue while focusing on the important properties that should be maintained. Usually, these approaches are coupled with advanced runtime systems that support the users in finding their way through the actual situations toward the process goals, while remaining within the boundaries determined by the process model. An early example of this approach is described in [46].

This category is also the one we mainly took as inspiration to develop DSOL, abandoning the imperative style followed used by most of the service composition languages and adopting a strongly declarative and flexible approach. Such flexibility was leveraged by the runtime system to simplify the definition of exception-safe service orchestrations and to allow deviations and changes during process execution.

7.2 Alternative Approaches to Service Composition

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. Hereafter, we review those that are mainly related with our work.

As an alternative to BPEL and BPMN in the specification of service compositions, other languages, like JOpera [20] (see our specific comparison in Section 4.2.2), Jolie [47], and Orc [48], 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 problems that motivate our work.

To overcome these limitations, other researchers followed the idea of adopting a declarative approach. Among those proposals, DecSerFlow [49, 50] is the closest to our work. In DecSerFlow service choreographies 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 DecSerFlow and DSOL. First of all, DecSerFlow focuses on service choreographies and on modeling them to support verification and monitoring. Conversely, we focus on service orchestrations and 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 [49]. The DSOL approach to modeling offers less opportunities for verification but it can lead to an efficient enactment tool. Secondly, DSOL emphasizes re-planning at run-time as a mechanism to support self-adaptive service orchestrations that maximize reliability even in presence of unexpected failures and changes in the external services. This is an issue largely neglected by DecSerFlow, as it focuses on specification and verification and it does not offer specific mechanisms to manage failures at run-time.

GO-BPMN [51, 52, 53] 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 to 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 DSOL 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 [54] defines a goal-oriented service orchestration language inspired by agent programming languages, like AgentSpeak(L) [55]. 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.

As briefly introduced in Chapter 1, the complexity in defining Web service compositions is also being tackled through *Automated Service Composition* (ASC) approaches. While our research was motivated by the desire of overcoming the limitations of mainstream orchestration languages in terms of flexibility and adaptability to unexpected situations, ASC is grounded on the idea that the main problem behind service orchestration 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 [56], user requirements and Web services are both described in DAML-S [57], 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, [58] 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 DSOL. In these approaches the planning domain is composed by the semantically described services and goals are defined by end-users. For example, [59] uses the SHOP2 planner to build compositions of services described in DAML-S. Similarly, [60] 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 [61], 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. [62] and [63] use requirements specified by the final user through queries in order to discover and compose the final orchestration. Both approaches use an internal planner for choosing the services that best match the user request. While [62] uses an ad-hoc execution infrastructure, [63] augments an abstract BPEL process with the selected services to enact the composition.

Other ASC approaches start from an abstract “template process”, expressed either in BPEL, *e.g.,* [64, 65], or as a Statechart, *e.g.,* [66] 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 DSOL, 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 Chapter 2. Moreover, most of the ASC approaches proposed so far operate before the orchestration starts, while DSOL includes advanced mechanisms to automatically adapt the orchestration 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.

As a final notice, we observe that the three-layered architectural model for self-management described by [67] and [68] was also used as an inspiration for DSOL and its engine. In particular, the layers defined by this architecture are: the *goal management layer*, responsible for the generation of plans from high-level goals (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 DSOL 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/concrete actions, used to build and enact the plan).

7.3 Quality-of-Service Management

Many existing approaches have an explicit support for QoS with different levels of abstractions and leveraging on a plethora of different techniques. For example, Menascé [69] discusses QoS issues in the domain of services, introducing the response times, availability, security, and throughput as QoS parameters. His paper also discusses the need of SLAs without advocating any specific model to manage, aggregate and optimize QoS behaviors of service orchestrations.

In particular, concerning optimization, many approaches exploit linear programming to manage QoS for orchestrations. For example, Aggrawal et al. [70] view QoS-based composition as a constraint satisfaction/optimization problem and find an optimal solution by applying integer linear programming. Zeng et al. [71] present comprehensive research about QoS modeling and QoS-aware compositions. In particular, they use statecharts to model orchestrations in which services are selected from a pool of alternative services using linear programming techniques such that it optimizes a local as well as global QoS criteria. Alternatively to linear programming, in [72], the authors leverage on fuzzy distributed constraint satisfaction techniques for finding the optimal orchestration. All these approaches differs from our proposal in many aspects. First we do not consider only alternative bindings in finding the optimal orchestration but we also consider structural alternatives (*i.e.,* plans) in finding the optimal solution. Secondly, we support domain specific metrics and adaptivity in terms of adaptive re-binding and re-planning.

Concerning the aggregation functions for QoS metrics, other existing approaches propose similar techniques aimed at aggregating metrics (*e.g.,* [73, 74, 75]). For example, Cardoso et al. [75] compute aggregate QoS by applying a set of reduction rules to the workflow until one atomic task is obtained. In addition, other approaches support custom specific metrics such as [73, 76]. However, all these approaches may not guarantee the optimal solution with respect to QoS even if they may be suitable where optimality is not mandatory and execution efficiency is preferred. Among these works let us mention the approaches based on genetic programming such as [77, 78] or on heuristics (*e.g.,* [79]).

Finally, concerning specifically adaptivity and re-planning we may mention respectively [80, 81] and [82]. The first two approaches do not focus on QoS, conversely the third one provides an efficient re-planning technique that, however, do not guarantees the optimal solution.

Summing up, none of the existing approaches, at the best of our knowledge, mix together an optimal solution, custom specific metrics and adaptive capabilities as DSOL with the adaptivity techniques of re-binding and re-planning. In addition, this is the first approach that combines planning together with optimization which allow the easy of use of declarative languages with the guarantee of optimality. Furthermore, differently from most of the QoS approaches, DSOL consider quality-of-service as part of the language, including specific constructs and runtime mechanisms to deal with it. Indeed, DSOL does not require any external proxy or broker to select the best alternative to be invoked, further simplifying its application.

7.4 Software Engineering and mobile applications

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 [83] and [84]. Existing works span a wide range of approaches: from how to achieve context-aware behavior (*e.g.,* [85]) to how to apply agile methods in the mobile domain (*e.g.,* [86]).

Let us first consider context-aware frameworks. These approaches 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) ([87]). For example, Subjective-C ([85]) provides context-oriented abstractions on top of Objective-C, a mainstream language used for programming iOS applications. The EgoSpaces middleware ([88]) can be used to provide context information extracted from data-rich environments to applications. Another approach to mobile computing middleware is presented in [89], 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 [90, 91] 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.

Other existing related approaches (*e.g.,* [92]) provide solutions for multi-platform app development. Approaches like [93] and [94] 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 [95] 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.

8 Conclusions and Future Works

Service-oriented computing holds great promises. Through it, an open and dynamic world of services becomes accessible for humans, who can be empowered by useful application components that are developed by service providers and exposed for possible use. Service-oriented computing may also generate new business. For example, it supports service provision by brokers who can integrate third-party services and export new added-value services. Because services live in open platforms, the computational environment is continuously evolving. New services may be created, old services may be discontinued, and existing services may be evolved by their owners.

Service-oriented computing raises several important challenges that need to be addressed by research to become successful. In particular, how can the new systems we build by composing existing services be described? How can such descriptions accommodate the need for tolerating the continuous changes that occur in the computational environment?

Service compositions may be achieved through workflow languages. Workflows describe processes through which humans interact with software components and compose them to achieve their goals. The existing workflow languages that have been developed to support service compositions, unfortunately, are still very primitive. Indeed, they have limited support to deal with the unpredictability and changeability of the world they live in. Defining adaptive service orchestrations in a simple and straightforward manner is still a goal hard to be achieved.

In this thesis we presented an approach to overcome the limitations of currently available service orchestration languages, in particular when failures occur and the application needs to self-adapt to unexpected situations. This approach is based on a new language called DSOL, which models the orchestration declaratively, focusing on the set of available activities, without having to explicitly declare the control flow of the orchestration. The execution flow is just generated at run-time through an ad-hoc planner, part of the DSOL engine. This simplifies the task of modeling complex orchestrations, increases the level of reusability, and can easily achieve self-adaptation in the presence of failures.

Several mechanisms are part of DSOL to build self-adapting orchestrations. First, each activity is modeled through an abstract description coupled with several concrete implementations, to be tried in case of failures. If this is not enough, the Planner can be re-invoked during process execution to find an alternative way to accomplish the orchestration goal, by-passing those activities that cannot be successfully executed, and undoing already executed activities of the old plan, if necessary. At last, as no explicit workflow is pre-defined, new activities (*i.e.,* abstract and concrete actions) can be added to the model at run-time, without the need to redeploy the entire orchestration.

For the future, we envision different lines of work. First, we believe that there is still space to improve both the DSOL language and its runtime system. The former, for example, still does not provide support for loops in the abstract layer. Iterations need to be implemented in the concrete level. As for the DEng run-time system, while the current prototype is fully operational (and downloadable) [1], we think there is still space to further improve performance and also reliability and robustness. Another aspect that we want to improve is the support to monitoring running orchestrations. We currently catch faults as they happen and we start our counter-measures (invoking alternative concrete actions or re-building the plan), but we are not able to “anticipate” faults. More advanced monitoring mechanisms may try to anticipate faults, *e.g.,* by checking for the actual availability of external services in advance, before their unavailability impacts the running orchestration.

Moreover, we plan to build a complete (graphical) tool, possibly integrated in an IDE like Eclipse [96], to further simplify the definition of abstract actions, goals, and orchestration interfaces.

Finally, to fully validate our approach we want to test its feasibility in additional, real world systems.

To conclude, while research in Web services and related technologies is a relatively mature field and extensively used in practice, it is continuously evolving to satisfy new demands. We do not believe that the contributions of this thesis represent a final answer to most of the problems we identified. However, we are convinced that it provides a valid contribution to guide future research effort.

[1] DEng is available at http://www.dsol-lang.net

Bibliography

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

[2] Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward open-world software: Issue and challenges. *Computer*, 39:36–43, 2006.

[3] Alexandre Alves, Assaf Arkin, Sid Askary, Ben Bloch, Francisco Curbera, Yaron Goland, Neelakantan Kartha, Canyang Kevin Liu, Dieter Konig, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, Alex Yiu, and eds. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS, 2006. [online] `http://www.oasis-open.org/apps/org/workgroup/wsbpel/`.

[4] OMG. Business Process Modeling Notation (BPMN), Version 2.0, 2011. [online] `http://www.omg.org/spec/BPMN/2.0/`.

[5] R. Maigre. Survey of the tools for automating service composition. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 628 –629, 2010.

[6] Jinghai Rao and Xiaomeng Su. A Survey of Automated Web Service Composition Methods. In *LNCS*, volume 3387/2005, pages 43–54. Springer, 2005.

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

[8] Chun Ouyang, Marlon Dumas, Wil M. P. Van Der Aalst, Arthur H. M. Ter Hofstede, and Jan Mendling. From business process models to process-oriented software systems. *ACM Transactions on Software Engineering and Methodology*, 19(1):2:1–2:37, August 2009.

[9] Betty Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. In Betty Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science*, pages 1–26. Springer Berlin / Heidelberg, 2009.

[10] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, July 2004.

[11] Java API for XML-Based Web Services (JAX-WS) 2.0. [online] `http://jcp.org/en/jsr/detail?id=224`.

[12] Gianpaolo Cugola, Elisabetta Di Nitto, Alfonso Fuggetta, and Carlo Ghezzi. A Framework for Formalizing Inconsistencies and Deviations in Human-centered Systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(3):191–230, July 1996.

[13] Apache CXF: An Open-Source Services Framework. [online] `http://cxf.apache.org/`.

[14] Apache HttpComponents. [online] `http://hc.apache.org/`.

[15] Code Generation Library. [online] `http://cglib.sourceforge.net/`.

[16] Felipe Meneguzzi and Michael Luck. Declarative agent languages and technologies vi. chapter Leveraging New Plans in AgentSpeak(PL), pages 111–127. Springer-Verlag, Berlin, Heidelberg, 2009.

[17] JavaGP - Java GraphPlan. [online] `http://emplan.sourceforge.net`.

[18] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1):1636–1642, 1995.

[19] Anis Charfi and Mira Mezini. AO4BPEL: An Aspect-oriented Extension to BPEL. *World Wide Web*, 10:309–344, September 2007.

[20] Cesare Pautasso and Gustavo Alonso. Jopera: A toolkit for efficient visual composition of web services. *International Journal of Electronic Commerce*, 9(2):107–141, January 2005.

[21] Cesare Pautasso. Composing restful services with jopera. In *International Conference on Software Composition*, volume 5634, pages 142–159, Zurich, Switzerland, July 2009. Springer.

[22] Yahoo! Local Search Web Services. [online] `http://developer.yahoo.com/search/local/V3/localSearch.html`.

[23] Doodle APIs. [online] `http://doodle.com/about/APIs.html`.

[24] Google Static Maps API. [online] `http://code.google.com/apis/maps/documentation/staticmaps/`.

[25] The Google Places API. [online] `http://code.google.com/apis/maps/documentation/places/`.

[26] Yahoo! Place Finder. [online] `http://developer.yahoo.com/geo/placefinder/`.

[27] The Google Geocoding API. [online] `http://code.google.com/apis/maps/documentation/geocoding/`.

[28] Bing Maps APIs. [online] `http://msdn.microsoft.com/en-us/library/dd877180.aspx`.

[29] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamental of Software Engineering, 2nd Ed.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[30] JBOSS jBPM. [online] `http://www.jboss.org/jbpm`.

[31] D. Karastoyanova and F. Leymann. Bpel'n'aspects: Adapting service orchestration logic. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 222 –229, july 2009.

[32] Core J2EE Patterns - Data Access Object. [online] `http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html`.

[33] Anis Charfi and Mira Mezini. Hybrid web service composition: business processes meet business rules. In *Proc. of the 2nd International Conference on Service Oriented Computing*, ICSOC '04, pages 30–38, New York, NY, USA, 2004. ACM.

[34] The AspectJ Project. [online] `http://www.eclipse.org/aspectj/`.

[35] Raj Jain. *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling.* Wiley, 1991.

[36] Amazon EC2 Instance Types. [online] `http://aws.amazon.com/ec2/instance-types/`.

[37] ActiveBPEL. [online] `http://www.activebpel.com`.

[38] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic qos management and optimization in service-based systems. *IEEE Transactions on Software Engineering*, 37(3):387 –409, may-june 2011.

[39] Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. Model evolution by run-time parameter adaptation. In *ICSE '09*, pages 111–121, 2009.

[40] G. Dantzig. *Linear Programming and Extensions.* Landmarks in Physics and Mathematics. Princeton University Press, 1998.

[41] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A mathematical programming language.* AT&T Bell Laboratories, 1987.

[42] L. Osterweil. Software processes are software too. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 2–13, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[43] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. Language constructs for managing change in process-centered environments. *SIGSOFT Softw. Eng. Notes*, 15:206–217, October 1990.

[44] Robert Balzer. Tolerating inconsistency. In *Proceedings of the 13th international conference on Software engineering*, ICSE '91, pages 158–165, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[45] S. C. Bandinelli, A. Fuggetta, and C. Ghezzi. Software process model evolution in the spade environment. *IEEE Trans. Softw. Eng.*, 19:1128–1144, December 1993.

[46] Gianpaolo Cugola. Tolerating deviations in process support systems via flexible enactment of process models. *IEEE Trans. Softw. Eng.*, 24(11):982–1001, November 1998.

[47] Fabrizio Montesi, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Jolie: a java orchestration language interpreter engine. *Electron. Notes Theor. Comput. Sci.*, 181:19–33, June 2007.

[48] David Kitchin, Adrian Quark, William Cook, and Jayadev Misra. 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, pages 1–25, Berlin, Heidelberg, 2009. Springer-Verlag.

[49] Marco Montali, Maja Pesic, Wil M. P. van der Aalst, Federico Chesani, Paola Mello, and Sergio Storari. Declarative specification and verification of service choreographiess. *ACM Trans. Web*, 4(1):3:1–3:62, January 2010.

[50] W. M. P. van der Aalst and M. Pesic. Decserflow: towards a truly declarative service flow language. In *Proceedings of the Third international conference on Web Services and Formal Methods*, WS-FM'06, pages 1–23, Berlin, Heidelberg, 2006. Springer-Verlag.

[51] Dominic Greenwood and Giovanni Rimassa. Autonomic goal-oriented business process management. *Autonomic and Autonomous Systems, International Conference on*, 0:43, 2007.

[52] Birgit Burmeister, M. Arnold, Felicia Copaciu, and Giovanni Rimassa. 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, pages 37–44, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[53] Monique Calisti and Dominic Greenwood. Goal-oriented autonomic process modeling and execution for next generation networks. In Sven van der Meer, Mark Burgess, and Spyros Denazis, editors, *Modelling Autonomic Communications Environments*, volume 5276 of *Lecture Notes in Computer Science*, pages 38–49. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-87355-6_4.

[54] M. Birna Van Riemsdijk and Martin Wirsing. 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, pages 31–48, Berlin, Heidelberg, 2007. Springer-Verlag.

[55] Anand S. Rao. 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, pages 42–55, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[56] Jinghai Rao, Peep Küngas, and Mihhail Matskin. Composition of semantic web services using linear logic theorem proving. *Inf. Syst.*, 31(4):340–360, June 2006.

[57] Mark H. Burstein, Jerry R. Hobbs, Ora Lassila, David Martin, Drew V. McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Terry R. Payne, and Katia P. Sycara. Daml-s: Web service description for the semantic web. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, ISWC '02, pages 348–363, London, UK, UK, 2002. Springer-Verlag.

[58] Sheila A. McIlraith and Tran Cao Son. Adapting golog for composition of semantic web services. In *Proceedings of the 8th International Conference on Principles and Knowledge Representation and Reasoning (KR-02)*, pages 482–496, 2002.

[59] Dan Wu, Bijan Parsia, Evren Sirin, James Hendler, and Dana Nau. Automating daml-s web services composition using shop2. In Dieter Fensel, Katia Sycara, and John Mylopoulos, editors, *The Semantic Web - ISWC 2003*, volume 2870 of *Lecture Notes in Computer Science*, pages 195–210. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-39718-2_13.

[60] Piergiorgio Bertoli, Marco Pistore, and Paolo Traverso. Automated composition of web services via planning in asynchronous domains. *Artif. Intell.*, 174(3-4):316–361, March 2010.

[61] Shankar R. Ponnekanti and Armando Fox. SWORD: A developer toolkit for web service composition. In *Proceedings of the 11th International WWW Conference (WWW2002)*, Honolulu, HI, USA, 2002.

[62] Walter Binder, Ion Constantinescu, Boi Faltings, Klaus Haller, and Can Türker. A multiagent system for the reliable execution of automatically composed ad-hoc processes. *Autonomous Agents and Multi-Agent Systems*, 12:219–237, 2006. 10.1007/s10458-006-5836-0.

[63] Alexander Lazovik, Marco Aiello, and Mike Papazoglou. Planning and monitoring the execution of web service requests. *Int. J. Digit. Libr.*, 6(3):235–246, June 2006.

[64] Danilo Ardagna and Barbara Pernici. Adaptive service composition in flexible processes. *IEEE Transactions on Software Engineering*, 33:369–384, 2007.

[65] Rohit Aggarwal, Kunal Verma, John Miller, and William Milnor. Constraint Driven Web Service Composition in METEOR-S. In *Proceedings of the 2004 IEEE International Conference on Services Computing*, SCC '04, pages 23–30, Washington, DC, USA, 2004. IEEE Computer Society.

[66] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30:311–327, 2004.

[67] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *2007 Future of Software Engineering*, FOSE '07, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[68] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. 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, pages 1–8, New York, NY, USA, 2008. ACM.

[69] D.A. Menasce. QoS issues in Web services. *Internet Computing, IEEE*, 6(6):72 – 75, nov/dec 2002.

[70] Rohit Aggarwal, Kunal Verma, John Miller, and William Milnor. Constraint driven web service composition in meteor-s. In *Proceedings of the 2004 IEEE International Conference on Services Computing*, SCC '04, pages 23–30, Washington, DC, USA, 2004. IEEE Computer Society.

[71] L. Zeng, B. Benatallah, A.H.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.

[72] Xuan Thang Nguyen, R. Kowalczyk, and Manh Tan Phan. Modelling and solving qos composition problem using fuzzy discsp. In *Web Services, 2006. ICWS '06. International Conference on*, pages 55 –62, sept. 2006.

[73] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, Francesco Perfetto, and Maria Luisa Villani. Service composition (re)binding driven by application-specific qos. In *Proceedings of the 4th international conference on Service-Oriented Computing*, ICSOC'06, pages 141–152, Berlin, Heidelberg, 2006. Springer-Verlag.

[74] M.C. Jaeger, G. Rojec-Goldmann, and G. Muhl. Qos aggregation in web service compositions. In *e-Technology, e-Commerce and e-Service, 2005. EEE '05. Proceedings. The 2005 IEEE International Conference on*, pages 181 – 185, march-1 april 2005.

[75] A. Sheth, J. Cardoso, J. Miller, K. Kochut, and M. Kang. Qos for service-oriented middleware. In *Proceedings of the Conference on Systemics, Cybernetics and Informatics*, pages 130–141, 2002.

[76] Yutu Liu, Anne H. Ngu, and Liang Z. Zeng. Qos computation and policing in dynamic web service selection. WWW Alt. '04. ACM, 2004.

[77] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1069–1075, New York, NY, USA, 2005. ACM.

[78] G. Canfora, M. Di Penta, R. Esposito, and M.L. Villani. A framework for qos-aware binding and re-binding of composite web services. *Journal of Systems and Software*, 81(10):1754–1769, 2008.

[79] R. Berbner, M. Spahn, N. Repp, O. Heckmann, and R. Steinmetz. Heuristics for qos-aware web service composition. In *International Conference on Web Services, 2006. ICWS'06*, pages 72–82. IEEE, 2006.

[80] Fabio Casati, Ski Ilnicki, Li-jie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in eflow. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, CAiSE '00, pages 13–31, London, UK, UK, 2000. Springer-Verlag.

[81] Dragan Ivanovic, Manuel Carro, and Manuel Hermenegildo. Towards data-aware qos-driven adaptation for service orchestrations. In *Proceedings of the 2010 IEEE International Conference on Web Services*, ICWS '10, pages 107–114, Washington, DC, USA, 2010. IEEE Computer Society.

[82] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. Qos-aware replanning of composite web services. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '05, pages 121–129, Washington, DC, USA, 2005. IEEE Computer Society.

[83] J. Dehlinger and J. Dixon. Mobile application software engineering: Challenges and research directions. In *Workshop on Mobile Software Engineering*, 2011.

[84] Anthony I. Wasserman. Software engineering issues for mobile application development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 397–400, New York, NY, USA, 2010. ACM.

[85] Sebastián González, Nicolás Cardozo, Kim Mens, Alfredo Cádiz, Jean-Christophe Libbrecht, and Julien Goffaux. Subjective-c. In Brian Malloy, Steffen Staab, and Mark van den Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 246–265. Springer Berlin / Heidelberg, 2011. 10.1007/978-3-642-19440-5_15.

[86] Pekka Abrahamsson, Antti Hanhineva, Hanna Hulkko, Tuomas Ihme, Juho Jäälinoja, Mikko Korkala, Juha Koskela, Pekka Kyllönen, and Outi Salo. 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, pages 174–175, New York, NY, USA, 2004. ACM.

[87] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March 2008.

[88] Christine Julien and Gruia-Catalin Roman. Egospaces: Facilitating rapid development of context-aware mobile applications. *IEEE Transactions on Software Engineering*, 32:281–298, 2006.

[89] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. Carisma: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29:929–945, 2003.

[90] Malte Appeltauer, Robert Hirschfeld, and Tobias Rho. 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, pages 38–43, Washington, DC, USA, 2008. IEEE Computer Society.

[91] Bart van Wissen, Nicholas Palmer, Roelof Kemp, Thilo Kielmann, and Henri Bal. ContextDroid: an expression-based context framework for Android. In *Proceedings of PhoneSense 2010*, November 2010.

[92] Julian Ohrt and Volker Turau. Cross-platform development tools for smartphone applications. *Computer*, 99(PrePrints), 2012.

[93] PhoneGap. [online] `http://www.phonegap.com/`.

[94] Appcelerator | Titanium Mobile Development Platform. [online] `http://www.appcelerator.com/`.

[95] Dipanjan Chakraborty, Anupam Joshi, Tim Finin, and Yelena Yesha. Service composition for mobile environments. *Mob. Netw. Appl.*, 10(4):435–451, August 2005.

[96] The Eclipse Foundation. [online] `http://www.eclipse.org/`.