Predicting Quality of Service of Software Applications

Ahmad Ibrahim

Supervisor
Antonio Brogi

October, 2016
Abstract

The ability to a priori predict the Quality of Service (QoS) of a software application is crucial both in the design of applications and in the definition of their Service Level Agreements (SLA). QoS prediction is challenging because of the different possible results of service invocations, and of the nondeterminism, correlations and complex dependencies among activities.

In this thesis we present a technique to probabilistically predict the QoS of service based and parallel design patterns based applications by applying Monte Carlo simulations to a simple representation of the control-flow of the applications. A proof-of-concept implementation of the analyses is presented and applied to various examples.
Contents

1 Introduction 13
  1.1 Importance of a priori prediction of QoS of application .............. 13
  1.2 Challenges in predicting the QoS of an application .................. 14
  1.3 Main results of the thesis ........................................... 18
  1.4 Structure of the thesis ............................................... 19

I Service based applications 21

2 Background: WS-BPEL 25
  2.1 Sequence ................................................................. 25
  2.2 Flow ...................................................................... 26
  2.3 If/else ................................................................. 26
  2.4 While ................................................................. 27
  2.5 Scope and faultHandlers ............................................. 27
  2.6 Invoke, Receive, Reply ............................................. 28
  2.7 Assign ............................................................... 28

3 Motivating Example: A cloud-based storage service 29

4 Predicting QoS of service based applications 33
  4.1 Cost compositors ...................................................... 33
  4.2 Evaluation of activities ............................................ 35
  4.3 Statistical non-determinism ....................................... 40

5 The PASO prototype 45
  5.1 Input of PASO ....................................................... 46
  5.2 Behaviour of PASO .................................................. 47
6 Examples 49
   6.1 A cloud-based storage service (continued) 49
   6.2 Starting a manufacturing business 51
   6.3 Loan approval service 54
   6.4 Shipping service 59

7 Related Work 63

II Parallel design patterns based applications 69

8 Background: Parallel design patterns 73
   8.1 A core set of parallel design patterns 73

9 Motivating example: An image filtering application 77

10 Predicting QoS of parallel design patterns based applications 79
   10.1 Patterns syntax 79
   10.2 Cost compositors 80
   10.3 Evaluation of activities 81
   10.4 Statistical non-determinism 84

11 The PASA prototype 89
   11.1 Input of PASA 89
   11.2 Behaviour and output of PASA 90

12 Examples 91
   12.1 An image filtering application (continued) 91
   12.2 Dynamic reconfiguration 96

13 Related Work 101

III Conclusions 105

14 Conclusions 107

Bibliography 111
List of Figures

1.1 Examples of different results of service invocations. .......................... 15
1.2 Example of nondeterminism in a workflow. ........................................... 16
1.3 Example of complex dependencies among workflow activities. ................. 16
1.4 Example of complex dependencies in a pipe. ........................................ 17
1.5 Example of correlation among workflow activities. ............................... 17

3.1 A cloud-based storage service. .............................................................. 30
3.2 Extended cloud-based storage service. ................................................ 30

5.1 Bird-eye view of the input-output behaviour of the PASO analyser. ............ 45
5.2 Screenshot of a PASO output. ............................................................... 48

6.1 Snapshot of PASO results for (Q1), (Q2) and (Q3). ............................... 50
6.2 Business process to start a manufacturing business. ............................... 52
6.3 Snapshot of PASO results for (Q1) and (Q2). ........................................ 53
6.4 Loan approval service. ............................................................................ 54
6.5 Activity diagram for monitoring the loan approval service. ....................... 58
6.6 Response time comparison for loan approval service. ............................. 59
6.7 Shipping service. ..................................................................................... 60

7.1 Example of Cardoso reduction rules for sequence of activities. ................. 64

8.1 Node ....................................................................................................... 74
8.2 Execution order of activities inside Comp. ............................................. 74
8.3 Execution order of activities inside Pipe. .............................................. 75
8.4 Farm. ...................................................................................................... 76
8.5 Feedback................................................................................................. 76

9.1 An example of an image blurring application (with a feedback loop). ....... 77
11.1 Bird-eye view of the input-output behaviour of the PASA analyser. . . . . . 89

12.1 Growth trends of the considered multiple QoS attributes (with fixed percentage of heavy items and varying probability of the feedback condition "blurred?" to get satisfied by an input item). . . . . . . . . . . . . . . . . . . . . 93
12.2 An example of comparison for different composition of patterns. . . . . . . 95
12.3 Application pattern . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
12.4 Comparative analysis between energy consumption and completion time. . 99
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Probability distributions for the cloud storage services.</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Two alternative offerings for the cloud storage services.</td>
<td>32</td>
</tr>
<tr>
<td>3.3</td>
<td>Probability distribution of a third cloud storage service.</td>
<td>32</td>
</tr>
<tr>
<td>6.1</td>
<td>Results of PASO for (Q1).</td>
<td>50</td>
</tr>
<tr>
<td>6.2</td>
<td>Results of PASO for (Q4).</td>
<td>51</td>
</tr>
<tr>
<td>6.3</td>
<td>Results of PASO for (Q5).</td>
<td>51</td>
</tr>
<tr>
<td>6.4</td>
<td>Probability distributions for the services invoked by the process of Fig. 6.2</td>
<td>52</td>
</tr>
<tr>
<td>6.5</td>
<td>Summary of the results of PASO for (Q1) and (Q2).</td>
<td>53</td>
</tr>
<tr>
<td>6.6</td>
<td>Input distributions for the loan approval service.</td>
<td>56</td>
</tr>
<tr>
<td>6.7</td>
<td>Six runs of the loan approval example.</td>
<td>57</td>
</tr>
<tr>
<td>6.8</td>
<td>QoS estimations for different number of samples for the loan approval example</td>
<td>58</td>
</tr>
<tr>
<td>6.9</td>
<td>Reliability comparison for loan approval service.</td>
<td>59</td>
</tr>
<tr>
<td>6.10</td>
<td>Input distributions for the shipping service example.</td>
<td>60</td>
</tr>
<tr>
<td>6.11</td>
<td>Six runs of the shipping service example.</td>
<td>61</td>
</tr>
<tr>
<td>6.12</td>
<td>QoS estimations for different number of samples for the shipping service example</td>
<td>62</td>
</tr>
<tr>
<td>7.1</td>
<td>Qualitative comparison among related work.</td>
<td>67</td>
</tr>
<tr>
<td>9.1</td>
<td>(a) Energy and (b) time required by each application step for processing a given image frame.</td>
<td>78</td>
</tr>
<tr>
<td>12.1</td>
<td>Completion times for a pipelined implementation of our motivating example application.</td>
<td>92</td>
</tr>
<tr>
<td>12.2</td>
<td>Bandwidth categories.</td>
<td>96</td>
</tr>
<tr>
<td>12.3</td>
<td>List of configurations</td>
<td>98</td>
</tr>
<tr>
<td>12.4</td>
<td>PASA results for Dynamic reconfiguration</td>
<td>100</td>
</tr>
</tbody>
</table>
Acknowledgments

In the name of Allah, Most Gracious, Most Merciful

This work could not have been achieved without the help, support, and encouragement of many people.

I would like to express special thanks to my supervisor Prof. Antonio Brogi. He taught me what scientific research is and how to carry it out. Without his guidance and persistent help this dissertation would not have been possible. Thank you for giving me the opportunity to realize this dream.

I want to thank Leonardo Bartoloni, Jacopo Soldani, Daniele De Sensi, Massimo Torquati and Andrea Canciani for the help and suggestions they provided me in this research. I am grateful to Prof. Marco Danelutto and Prof. Antonio Cisternino for their valuable feedback and suggestions. I am also very grateful to the external reviewers for their valuable feedback.

I would like to thank Prof. Pierpaolo Degano and Giovanna Contaldo for all their academic and administrative support. I want to thank United Nations University (UNU) and University of Pisa for financial support during my PhD.

Finally, to all my family, friends and colleagues who encouraged and advised me during my PhD, thank you very much!
Chapter 1

Introduction

Quality of Service (QoS) refers to a set of non-functional attributes used to describe a system [1] such as response time, reliability or cost [2].

The QoS of an application does depend on the QoS of the services it uses. For instance, for service based applications [3, 4], the response time of a service orchestration obviously depends on the response times of the services it invokes - which may actually vary over time for different reasons (e.g., workload or network congestion).

A straightforward way to assess the QoS of an application would be to simply deploy it and then monitor the QoS parameters of interest over a sufficiently high number of runs. Unfortunately, such an approach can be time consuming, expensive (when non-free services are invoked) and may not be feasible when invocations have side-effects (e.g., as in the case of services enabling monetary transactions).

1.1 Importance of a priori prediction of QoS of application

The ability of a priori predicting the QoS of an application is very valuable for application designers. In this thesis we consider two classes of applications

1. Service based applications. Service based application are usually defined via workflows that implement business processes by orchestrating various (possibly third-party) services [3, 4].

2. Parallel design patterns based applications. Parallel design patterns (also called skeletons) are customisable, composable design patterns that can be fruitfully exploited to define parallel applications [5, 6].
In service based applications, predicting the QoS of a service orchestration is valuable both during the design of the orchestration and for defining its Service Level Agreement (SLA, [7]). It helps answering questions, like:

- What is the QoS of a given orchestration?
- What is the effect on the QoS of an orchestration of replacing one or more of the invoked services with alternative services, (e.g., offered by different providers)?
- How modifying the workflow of an orchestration impacts on its overall QoS?

Similarly, for parallel design pattern based applications, QoS prediction is valuable for comparing different deployments as well as for assessing the scalability of their parallelization. It helps answering questions, like:

- What is the QoS of a given parallel design pattern based application?
- Would a restructuring improve the QoS of a parallel design pattern based application for a steady state?
- What type of parallelization would achieve the best QoS for a dynamic state?

1.2 Challenges in predicting the QoS of an application

A priori predicting the QoS of an application is not easy mainly because of four main challenges.

1.2.1 Different possible results of service invocations

In service based applications, each invoked service can return a successful reply, a fault notification, or even no reply at all [8]. If a fault is returned, a fault handling routine will be executed instead of the normal control flow. If no reply is received, the orchestrator will get stuck (unless some parallel branch throws a fault). In either case, the resulting QoS of the orchestration will differ from the case of successful invocation.

Consider for instance the workflow in Fig. 1.1. If the invocation to S is successful (Fig. 1.1a) then activity C will be executed after B completes, while if the invocation to S returns a fault (Fig. 1.1b) a fault handling activity will be executed instead of activity C. Moreover, if no reply is received from S (Fig. 1.1c), then activity B will get stuck and neither C nor the fault handling activity will be executed.
1.2. CHALLENGES IN PREDICTING THE QOS OF AN APPLICATION

(a) Successful invocation

(b) Faulty invocation

(c) No response

Figure 1.1: Examples of different results of service invocations.

In the case of parallel design patterns based applications, the QoS of the activities executed depends for instance on the type of the inputs arriving in the input stream, as different inputs typically require different service times.

1.2.2 Nondeterminism in orchestration workflows

The control flow of a workflow defining a service based applications is in general nondeterministic. Besides the nondeterminism induced by the possible different results of service invocations, a workflow usually contains branching conditions that depend upon input data values. As a consequence, which branch will be executed in an alternative, or the number of iterations of a loop is not known a priori [9, 10] (Fig. 1.2).

In the case of parallel design patterns based applications, the execution of conditional loops defined by feedback patterns has a similar nondeterministic behaviour, as some outputs may be routed back depending upon the evaluation of conditions, and the number of iterations is not known a priori.
1.2.3 Complex dependencies among activities

Workflows can contain complex dependencies among activities, as for instance those defined by WS-BPEL synchronization links [11]. The control flow defined by synchronizations within parallel activities is more expressive than what is allowed by simple parallel execution with synchronization barriers at the end. This implies that workflows containing such complex synchronization structures cannot be always decomposed into parallel and sequential compositions, as shown in [12, 13].

Consider for instance the workflow in Fig. 1.3, where dashed arrows represent WS-BPEL synchronization links between activities. It is easy to see that the dependencies defined by the links cannot be decomposed into parallel and sequential compositions\(^1\).

---

\(^1\)If we denote by ";" and "\mid" standard sequential and parallel composition, respectively, it is easy to observe for instance that \( (A;C)|(B;D) \) does not model that \( C \) can be executed only after \( B \), or that \( (A|B);(C|D) \) incorrectly imposes that \( D \) can be executed only after \( A \).
1.2. CHALLENGES IN PREDICTING THE QOS OF AN APPLICATION

Consider for instance a pipe of three activities A, B and C and three data items in input stream (1,2,3) in Fig. 1.4, where the dashed arrows represent dependencies between activities. It is easy to see that for instance activity B$_2$ can execute only when both A$_2$ and B$_1$ have terminated.

![Diagram of complex dependencies in a pipe](image)

Figure 1.4: Example of complex dependencies in a pipe.

1.2.4 Correlations among workflow activities

Workflow nondeterminism cannot be modelled by simply assigning independent probabilities to activities without taking into account their correlation (e.g., as in [14]), as this may lead in some cases to incorrect results.

Consider for instance the workflow in Fig. 1.5a. If activity A will trigger, then either activity B or C will trigger with 50% probability, and activity D will be executed with 100% probability in either case. However, if one does not take into account the information that B and C are correlated parallel branches which both originated from activity A, and just considers their probabilities of being executed, then the probability that activity D will be executed will (incorrectly) become only 75% (Fig. 1.5b).

![Diagram of correlation among workflow activities](image)

(a) With correlation  
(b) Without correlation

Figure 1.5: Example of correlation among workflow activities.
1.2.5 Summary

As we will discuss in Chapter 7, there already exists many approaches to predict the QoS of service based applications at design time. Existing approaches can effectively predict the QoS of a given workflow composition when such workflow is well-structured, while the same does not hold when complex dependencies are present in a given workflow or when the workflow analysis requires to perform non-deterministic choices. Additionally, some of the existing approaches do not consider potential correlations among the activities composing a workflow or the fact that the results of service invocations can affect the overall QoS.

Concerning parallel design pattern based application, few approaches focus on QoS prediction at design time. Most of the existing approaches either focus on QoS optimization at compile time or monitoring QoS at runtime. As we will show in Chapter 13, the existing approaches which focus on QoS prediction at design time are focused on a limited set of patterns, and do not deal with patterns that may follow a non-deterministic behaviour.

To the best of our knowledge, none of the existing techniques for QoS prediction for service based and parallel design pattern based applications fully addresses all the aforementioned challenges respectively.

1.3 Main results of the thesis

In this thesis we present a design time QoS prediction algorithm for service based and parallel design patterns based applications. The QoS prediction techniques rely on two main ideas:

1. Expressing the control flow of applications in terms of two simple cost compositors (Both and Delay), and
2. Exploiting Monte Carlo simulations [15].

The results of the thesis are presented into two parts. Part I deals with service based applications while Part II deals with parallel design patterns based applications. In each part, we present an algorithm that suitably deals with all the challenges described in section 1.2 and that is capable of probabilistically predicting the QoS of an application.

For service based applications, the inputs of the algorithm are a WS-BPEL workflow, and probability distributions for the QoS properties of the invoked services as well as for the evaluation of the workflow branching conditions. The output of the algorithm is a probability distribution for the QoS properties (reliability, time and cost) of the orchestration.
For parallel design patterns based applications, the inputs of the algorithm are a parallel design pattern, and probability distributions for the QoS properties of the nodes, for the types of the input stream (optional) as well as for the evaluation of feedback conditions. The output of the algorithm is a probability distribution for the QoS properties (completion time and energy consumption) of the application.

1.4 Structure of the thesis

The thesis is organized into three parts. Part I deals with service based applications, Part II deals with parallel design patterns based applications and Part III provide discussion on limitations and future work.

Part I. Service based applications

Chapter 2 provides some background on WS-BPEL.
Chapter 3 introduces a simple yet motivating example.
Chapter 4 presents our algorithm to predict the QoS of WS-BPEL service orchestrations.
Chapter 5 describes a proof-of-concept implementation (PASO) of the QoS prediction algorithm.
Chapter 6 presents the results of our algorithm on the motivating example and on other three challenging examples.
Chapter 7 contains a comparative discussion of related work.

Part II. Parallel design patterns based applications

Chapter 8 provides some background on parallel design patterns.
Chapter 9 introduces a simple yet motivating example.
Chapter 10 presents our algorithm to predict the QoS of parallel design patterns based applications.
Chapter 11 describes a proof-of-concept implementation tool (PASA) of the algorithm.
Chapter 12 presents the results of our algorithm on the motivating example and on another interesting example.
Chapter 13 discusses related work.

Part III. Conclusions

Chapter 14 provides summary and discussion on limitations and future work.
Part I

Service based applications
The ability to “a priori” estimate the QoS of service based applications is valuable both during the design of an orchestration and for defining its Service Level Agreement (SLA). Assessing the QoS of a service orchestration by deploying and monitoring it is time consuming, expensive (when non-free services are invoked) and it may not be feasible when service invocations have side-effects (e.g., as in the case of services enabling monetary transactions). We present an algorithm to probabilistically predict the QoS of WS-BPEL [11] service orchestrations. We chose WS-BPEL since it is the OASIS standard for orchestrating Web services and since it natively features expressive synchronization mechanisms within parallel tasks (viz., synchronization links among parallel activities). The algorithm employs two simple cost compositors (Both and Delay) to represent the control flow of service based applications and it exploits Monte Carlo simulations [15].

The results presented in this part of the thesis were reported also in [16, 17, 18].
Chapter 2

Background: WS-BPEL

The Web Services Business Process Execution Language (WS-BPEL) [11], commonly known as BPEL, is an OASIS standard for web service compositions [3, 4, 19].

WS-BPEL defines a model and a grammar for describing the behavior of a business process based on interactions between the process and its partners using a XML-based language. Interactions with partners occur through Web Service interfaces, and the structure of the relationship at the interface level is encapsulated in Partnerlinks. links are used to describe cross enterprise business interactions in which the business processes of each enterprise interact through Web Service interfaces. PartnerLinks are used to directly model peer-to-peer conversational partner relationships while endpoint references allow WS-BPEL to dynamically select a provider for a particular type of service and to invoke their operations.

WS-BPEL can contain two types of activities, basic and structured. Basic activities are those which describe elementary steps of the process behavior. Structured activities encode control-flow logic, and therefore can contain other basic and/or structured activities recursively. The definitions and description used in following section are taken from [11].

The subset of WS-BPEL we consider includes structured (Sequence, Flow, IfThenElse, While, Scope and faultHandlers) and basic activities (Assign, OpaqueAssign, Invoke, Receive and Reply).

2.1 Sequence

A <sequence> activity contains one or more activities to be performed sequentially. A <sequence> activity completes when the last activity in the <sequence> has completed. For example, the following <sequence> shown below contains two structural activities <flow> and <scope>.
2.2 Flow

The `<flow>` activity provides both concurrency and synchronization. A `<flow>` completes when all of the activities enclosed by the `<flow>` have completed. A `<flow>` enables synchronization dependencies between activities that are nested within it to any depth. `<link>`s are used to express synchronization dependencies among activities in a `<flow>`. Each `<link>` has one source and one target, and it may be associated with a `<transitionCondition>` (predicate condition evaluated at the completion of the activity source of the link). Every activity that is the target of a link has an implicit or explicit `<joinCondition>` (a boolean expression on the status values of the incoming links) associated with it.

- Such `<joinCondition>` is evaluated once the status (TRUE/FALSE) of all the incoming links has been determined (e.g., when all corresponding source activities will have terminated).
- If the `<joinCondition>` evaluates to TRUE the activity can be executed, if it evaluates to FALSE a `<joinFailure>` fault may be thrown.
- If there is no explicit `<joinCondition>` specified, the implicit `<joinCondition>` requires the status of at least one incoming link to be TRUE.

In the following example, the two `<invoke>` activities are enabled to start concurrently when the `<flow>` starts. The “transferMoney” activity is executed after the `<flow>` completes.

```xml
<sequence>
  <flow>
    <invoke partnerLink="Seller" ... />
    <invoke partnerLink="Shipper" ... />
  </flow>
  <invoke partnerLink="Bank" name="transferMoney" ... />
</sequence>
```

2.3 If/else

`<if>` activity provides conditional behavior. The activity consists of conditional branches defined by the `<if>` and optional `<elseif>` elements, followed by an optional `<else>`
element. The `<if>` and `<elseif>` branches are considered in the order in which they appear. The first branch whose `<condition>` holds true is taken, and its activities are performed. If no branch with a condition is taken, then the `<else>` branch is taken if present. For example, the `<if>` shown below will execute `<flow>` in case boolean expression evaluates to true, and `<invoke>` otherwise.

```xml
<if>
  <condition>bool expression</condition>
  <flow>...</flow>
<else>
  <invoke .../>
</else>
</if>
```

### 2.4 While

The `<while>` activity provides for repeated execution of a contained activity. The contained activity is executed as long as the boolean `<condition>` evaluates to true at the beginning of each iteration. For example, the `<while>` shown below will execute `<sequence>` as long as the condition `Amount $leq 50` evaluates to true.

```xml
<while>
  <condition> Amount $leq 50 </condition>
  <sequence>...</sequence>
</while>
```

### 2.5 Scope and faultHandlers

Each `<scope>` has a required primary activity that defines its normal behavior. The primary activity can be a complex structured activity, with many nested activities to arbitrary depth. Fault handlers attached to a scope provide a way to define a set of custom fault-handling activities, defined by `<catch>` construct. Each `<catch>` construct is defined to intercept a specific kind of fault, defined by a fault name. For example the `<scope>` in the example below will execute `<invoke>`, and if a fault is thrown, the `<sequence>` inside `<catch>` will execute.

```xml
<scope>
  <faultHandlers>
    <catch faultName="...">
      <sequence>...</sequence>
    </catch>
  </faultHandlers>
</scope>
```
2.6 Invoke, Receive, Reply

The <invoke> activity is used to call Web Services offered by service providers. The typical use is invoking an operation on a service.

A <receive> activity is used to instantiate a business process and receive messages from partner. The <reply> activity is used to send a response to a request previously accepted through an inbound message activity.

2.7 Assign

The <assign> activity can be used to copy data from one variable to another, as well as to construct and insert new data using expressions. The <from> tag specifies the data source while <to> specifies the intended target. A typical syntax for <assign> can be

```xml
<assign>
  <copy>
    <from>...</from>
    <to>...</to>
  </copy>
</assign>
```

WS-BPEL allows opaque assignments. An opaque expression is a placeholder for a corresponding executable WS-BPEL expression. An example usage is that of copying a hidden value into a known variable.
Chapter 3

Motivating Example: A cloud-based storage service

We will use a motivating example to illustrate the need and the difficulties of predicting the QoS of (even simple) service orchestrations. The example is based on cloud storage and retrieval service.

Let us consider a simple service orchestration (Fig. 3.1) that allows customers to store and retrieve data.

The orchestrator exploits two cloud storage services ($C_1$ and $C_2$) as follows:

- If the customer sends a *store* request, the orchestrator tries in parallel to store the data both on $C_1$ and $C_2$. If the first storage request on $C_1$ fails (viz., a fault is returned), the orchestrator retries once (after some time) to store on $C_1$. If the data are successfully stored on both $C_1$ and $C_2$, the orchestrator replies positively to the customer. Otherwise it returns a fault to the customer.

- If the customer sends a *retrieve* request, the orchestrator first looks up the data in $C_1$. If the invocation to $C_1$ fails, it looks up the data in $C_2$. If both invocations (to $C_1$ and $C_2$) fail, the orchestrator returns a fault to the customer. Otherwise it returns to the customer the result of the lookup.

Let us assume the following probability distributions (Table 3.1) for the behaviour of the cloud storage services $C_1$ and $C_2$, in particular for their reliability, cost, and response time:
Figure 3.1: A cloud-based storage service.

Figure 3.2: Extended cloud-based storage service.
Table 3.1: Probability distributions for the cloud storage services.

<table>
<thead>
<tr>
<th></th>
<th>Success</th>
<th>Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_1$</td>
<td>(99%, 0.03$, 1 sec)</td>
<td>(1%, 0$, 2 sec)</td>
</tr>
<tr>
<td>$C_2$</td>
<td>(90%, 0.02$, 1 sec)</td>
<td>(10%, 0$, 2 sec)</td>
</tr>
</tbody>
</table>

- $C_1$ cost is 0.03$ per invocation (for both store and retrieve requests), it is highly reliable and it completes almost always (99%) in 1 sec. Only in very few cases (1%), it returns a fault in 2 sec at no cost (0$).

- $C_2$ cost is 0.02$ per invocation, slightly cheaper than $C_1$, it is less reliable than $C_1$, and in most cases (90%) it completes in 1 sec. Only in few cases (10%), it returns a fault in 2 sec at no cost (0$).

Let us also assume that:

- 40% of customer requests are store requests, and 60% are retrieve requests, and that
- the random wait (before retrying to store on $C_1$) will last 0, 1, 2, 3, or 4 seconds, each with probability 20%.

A first natural question is:

(Q1) What are the estimated reliability, cost, and response time of the orchestrator using services $C_1$ and $C_2$ of Table 3.1?

Other interesting questions on the QoS of the orchestrator of Fig. 3.1 are, for instance:

(Q2) What is the probability that the response time of the orchestrator will be more than 5 sec using services $C_1$ and $C_2$ of Table 3.1?

(Q3) Will the cost of the orchestrator exceed 0.04$ on average using services $C_1$ and $C_2$ of Table 3.1?

Another class of interesting questions concerns assessing how the QoS of different external services may impact on the overall QoS of an orchestrator. Consider for instance the two alternative offerings for $C_1$ and $C_2$ illustrated in Table 3.2.
A cloud-based storage service

Table 3.2: Two alternative offerings for the cloud storage services.

<table>
<thead>
<tr>
<th></th>
<th>Success</th>
<th>Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>C₁</td>
<td>(90%, 0.02$, 1 sec)</td>
<td>(10%, 0$, 2 sec)</td>
</tr>
<tr>
<td>C₂</td>
<td>(90%, 0.02$, 1 sec)</td>
<td>(10%, 0$, 2 sec)</td>
</tr>
</tbody>
</table>

(a) Offering 1.

<table>
<thead>
<tr>
<th></th>
<th>Success</th>
<th>Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>C₁</td>
<td>(99%, 0.03$, 1 sec)</td>
<td>(1%, 0$, 2 sec)</td>
</tr>
<tr>
<td>C₂</td>
<td>(81%, 0.01$, 1 sec)</td>
<td>(19%, 0$, 2 sec)</td>
</tr>
</tbody>
</table>

(b) Offering 2.

An example of such questions is:

(Q4) Which offering of Table 3.2 will yield the best QoS (reliability, cost, response time) for the orchestrator of Fig. 3.1?

A further class of interesting questions concerns assessing whether and how modifying the workflow of an orchestrator will impact on the overall QoS of the orchestrator. For instance:

(Q5) Extending the orchestrator (Figure 3.2) so as to exploit one more cloud storage service C₃ (e.g., like the one described in Table 3.3) will increase the reliability of the orchestrator?

<table>
<thead>
<tr>
<th></th>
<th>Success</th>
<th>Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>C₃</td>
<td>(81%, 0.01$, 1 sec)</td>
<td>(19%, 0$, 2 sec)</td>
</tr>
</tbody>
</table>

Table 3.3: Probability distribution of a third cloud storage service.

In Chapter 6, we will discuss the results of probabilistically predicting the QoS of this example and of other three more challenging examples.
Chapter 4

Predicting QoS of service based applications

A priori prediction of the QoS of a service orchestration is not easy mainly because of the four challenges (described in Chapter 1). In this chapter, we describe our algorithm to predict QoS for service based applications.

4.1 Cost compositors

We define a structurally recursive function that associates, in a compositional way, each WS-BPEL activity with a cost structure. Note that such cost structure is general, and it can be instantiated to define different QoS values, such as the time needed to complete an activity or the monetary expense associated with its execution.

Intuitively speaking, we need to estimate what is the cost, for instance, of a sequential composition $\text{Sequence}(A,B)$ as a function of the costs of $A$ and $B$. Similarly, we need to estimate the cost of a parallel composition $\text{Flow(activitySet)}$ as a function of the costs of the activities in $\text{activitySet}$. While defining the cost composition function for a Sequence is pretty straightforward, defining the cost composition function for generic Flows is more challenging since (as anticipated in Chapter 1) not all Flows can be decomposed into parallel and sequential compositions [12].

We define two functions as base cost composition operations:

- The first cost compositor is a parallel compositor. $\text{Both}(A,B)$ defines the cost associated with executing independently an activity with cost $A$ and an activity with cost $B$. 
The second cost compositor defines the “delayed” cost of an activity which must wait for the completion of another activity before starting. \( \text{Delay}(A, B) \) defines the cost associated with executing an activity of cost \( A \) which must wait for the completion of an activity with cost \( B \) before starting.

For instance, the cost completion time indicating the time required for an activity to complete can be defined as\(^1\):

\[
\text{let} \quad \text{Both}(a\text{Time}, b\text{Time}) = \max(a\text{Time}, b\text{Time}) \\
\text{let} \quad \text{Delay}(a\text{Time} \ b\text{Time}) = a\text{Time} + b\text{Time}
\]

Namely, completion time is the maximum\(^2\) of the costs of the two activities in the case of Both, while it is the sum of the costs of the two activities in the case of Delay, since the delayed activity will start only after the other activity completes.

As a further example, the cost expense indicating the monetary cost of executing an activity can be defined as:

\[
\text{let} \quad \text{Both}(a\text{Expense}, b\text{Expense}) = a\text{Expense} + b\text{Expense} \\
\text{let} \quad \text{Delay}(a\text{Expense}, b\text{Expense}) = a\text{Expense}
\]

Namely, the expense is the sum of the costs of the two activities in the case of Both, while it is just the cost of the delayed activity in the case of Delay, since delaying an activity does not increase its expense.

The compositor Both is commutative, Both and Delay are associative, and Delay is right-distributive over Both:

\[\forall A, B, C:\]

Both\((A, B) = \text{Both}(B, A),\] 
Both\((A, \text{Both}(B, C)) = \text{Both}(\text{Both}(A, B), C),\] 
Delay\((A, \text{Delay}(B, C)) = \text{Delay}(\text{Delay}(A, B), C), \text{ and}\]
Delay\((\text{Both}(A, B), C) = \text{Both}(\text{Delay}(A, C), \text{Delay}(B, C)).\]

We also explicitly name a neutral element Zero for Both and Delay\(^3\).

\(^1\)As we have prototyped our algorithm with F# [20], we employ F# syntax to describe pseudo-code.

\(^2\)Taking the \(\max\) of \(a\text{Time}\) and \(b\text{Time}\) gives an optimistic estimation when \(A\) and \(B\) are two activities executed in parallel since: \(\max(a\text{Time}, b\text{Time}) \leq \text{Both}(a\text{Time}, b\text{Time}) \leq a\text{Time} + b\text{Time}.\)

\(^3\)\(\forall A: \text{Both}(A, \text{Zero}) = A \text{ and } \text{Delay}(A, \text{Zero}) = A.\)
4.2 Evaluation of activities

In order to evaluate a WS-BPEL activity, it is necessary to take into account the results of the activities that have been previously evaluated. We employ two parameters, \textit{environment} and \textit{outcome}, to store such information.

It is worth observing that WS-BPEL features two types of control-flow mechanisms that may lead to skipping the execution of an activity:

- \textit{“Normal” control-flow activities} (alternatives, iterations, synchronization \texttt{links within Flows}), which can be simply handled by evaluating branching conditions (e.g., in \texttt{IfThenElse} and \texttt{While}) and transition conditions (in synchronization \texttt{links}).

- \textit{Fault management activities}. The execution of an activity may also be skipped because of a fault thrown by another activity, which triggers the execution of a fault handling routine.

In the first case, the cost of activities can be determined by exploiting the \textit{environment} parameter, which holds variable values over which condition are evaluated, as well as the status of synchronization \texttt{links}. In the second case, the \textit{outcome} parameter can be exploited to keep track of the result of an activity, which can be \texttt{Success}, \texttt{Fault} or \texttt{Stuck} (when a service invocation does not get any reply).

Due to the importance of the \textit{environment} and \textit{outcome} parameters, we will explicitly include them in the following pseudo-code. We describe next the pseudo-code of the (recursive) evaluation function \texttt{exec} implementing the probabilistic analysis of the WS-BPEL \texttt{Sequence, Scope, Assign, IfThenElse, While, and Flow} activities.

4.2.1 Sequence

\begin{verbatim}
Sequence (a1,a2) =>
  let env1,outcome1,cost1 = exec env a1
  if outcome1 = Success then
    let env2,outcome2, cost2 = exec env1 a2
    env2, outcome2, Both(cost1, Delay(cost2, cost1))
  else
    env1, outcome1, cost1
\end{verbatim}

The evaluation of Sequence \((a1,a2)\) is straightforward.

If the result \((env1, outcome1, cost1)\) of evaluating \(a1\) is successful (viz., \texttt{outcome1 = Success}), then \(a2\) is evaluated in the new environment \texttt{env1}. The \textit{environment} \texttt{(env2)}
and the outcome (outcome2) obtained by evaluating a2 are then returned, together with
the cost term $\text{Both}(\text{cost1}, \text{Delay}(\text{cost2}, \text{cost1}))$ composing the cost of executing
a1 and the cost of executing a2 after a1.

If the result $(\text{env1}, \text{outcome1}, \text{cost1})$ of evaluating a1 is instead not successful, then
such result is returned as the result of evaluating $\text{Sequence}(a1, a2)$.

4.2.2 Scope

$$\text{Scope} \ (a1, \text{faultHandler}) \rightarrow$$

$$\text{let} \ \text{env1}, \text{outcome1}, \text{cost1} = \text{exec env a1}$$

$$\text{if} \ \text{outcome1} = \text{Fault} \ \text{then}$$

$$\text{let} \ \text{env2}, \text{outcome2}, \text{cost2} = \text{exec env1 faultHandler}$$

$$\text{else}$$

$$\text{env1}, \text{outcome1}, \text{cost1}$$

A Scope is associated with a faultHandler which is executed only when a fault occurs
in activity a1.

If the result $(\text{env1}, \text{outcome1}, \text{cost1})$ of evaluating a1 yields a fault (viz., outcome1
= Fault), then the faultHandler is evaluated in the new environment env1. The environment
(env2) and the outcome (outcome2) obtained by evaluating the faultHandler are then returned, together with the cost term $\text{Both}(\text{cost1}, \text{Delay}(\text{cost2}, \text{cost1}))$
composing the cost of executing a1 and the cost of executing faultHandler after a1.

If the result $(\text{env1}, \text{outcome1}, \text{cost1})$ of evaluating a1 is instead not fault, then
such result is returned as the result of evaluating $\text{Scope}(a1, \text{faultHandler})$.

4.2.3 Assign

$$\text{Assign} \ (\text{name}, \text{expr}) \rightarrow$$

$$\text{env}.\text{Add(name, boolExprEval env expr), Success, Zero}$$

The Assign activity assigns (the result of evaluating) an expression expr to a variable
name, and adds it to the environment. We assume that the outcome of Assign is always
Success and has Zero cost.

4.2.4 IfThenElse

$$\text{IfThenElse} \ (\text{guard}, a1, a2) \rightarrow$$
4.2. EVALUATION OF ACTIVITIES

let guardValue = boolExprEval env guard
if guardValue then
  exec env a1
else
  exec env a2

The IfThenElse activity first evaluates its guard condition. If it evaluates to true, then exec evaluates a1. If it evaluates to false, then exec evaluates a2.

4.2.5 While

While(guard, body) ->
let guardValue = boolExprEval env guard
if guardValue then
  exec env (Sequence (body, While(guard, body)))
else
  env, Success, Zero

Similarly to IfThenElse, a While activity first evaluates its guard condition. If it evaluates to false, the body of the loop is skipped and outcome Success and cost Zero are returned. Otherwise, exec evaluates the body of the loop, and then the While again.

4.2.6 Flow

Flow (activitySet, linkSet) ->
let mutable env = env
let outcomeSet = new Dictionary<string,Outcome>
let costSet = new Dictionary<string,Cost>

(* Step 1. Sort activities using links to ensure that preceding activities are evaluated before current activity. *)
let activityList = topoSort activityList linkSet

(* Step 2. Iterate for all activities present in Flow *)
for activity in activityList do
  let incomingLinks = [ l in linkSet where l.target = activity.name ]
  let outgoingLinks = [ l in linkSet where l.source = activity.name ]
  (* Step 2.1. Identify all preceding activities of current activity. *)
  let mutable preceedingActivities = []
  for l in incomingLinks do
    if l.source = a.name then
preceedingActivities ← a::preceedingActivities

(* Step 2.2. Compute outcome of all preceding activities.*)

let mutable outcomei = Success

for a in preceedingActivities

(* Fault has precedence over stuck and success.
   Stuck has precedence over success.*)

if outcomei = Success || outcomeSet.[a.name] = Stuck then
    outcomei ← Stuck
if outcomei = Success || outcomei = Stuck ||
    outcomeSet.[a.name] = Fault then
    outcomei ← Fault

(* Step 2.3. Compute cost of all preceding activities *)

let mutable costi = Zero

for a in preceedingActivities do

    costi ← Both (costi, costSet.[a.name])

(* Step 2.4. Evaluate join condition for current activity and assign cost and outcome accordingly.*)

let joinCondition = evaluateBooleanExpression activity.joinCondition env

match outcomei, joinCondition, suppressJoinFailure with
| Success , true , _ →
    let e, outcome, cost = exec activity env
    costSet.insert(activity.name, Delay(cost, costi))
    outcomeSet.insert(activity.name, outcome)
    env ← e
| Success , false , true →
    costSet.insert(activity.name, Delay(Zero, costi))
    outcomeSet.insert(activity.name, Success)
| Success , false , false →
    costSet.insert(activity.name, Delay(Zero, costi))
    outcomeSet.insert(activity.name, Fault)
| other , _, _ →
    costSet.insert(activity.name, Delay(Zero, costi))
    outcomeSet.insert(activity.name, other)

(* Step 2.5. Evaluate transition condition for all outgoing links of current activity.*)

for l in outgoingLinks do

    let linkStatus = evaluateBooleanExpression l.transitionCond env
    env ← env.Add(l.name, linkStatus)

(* Step 3. Compute cost and outcome of Flow once all activities are analyzed.*)

let mutable outcome = Success

let mutable cost = Zero
4.2. EVALUATION OF ACTIVITIES

```plaintext
for activity in activityList do
    cost ← Both(cost, costSet[activity.name])
    (* Fault has precedence over stuck and success.
       Stuck has precedence over success.*)
    if outcome = Success || outcomeSet[activity.name] = Stuck then
        outcome ← Stuck
    if outcome = Success || outcome = Stuck ||
        outcomeSet[activity.name] = Fault then
        outcome ← Fault

env, outcome, cost
```

A Flow activity contains a set activitySet of activities to be run in parallel subject to a (possibly empty) set linkSet of synchronization links.

For each activity A in activitySet we compute: (1) the outcome of A, which depends also on the outcomes of its precedingActivities (if any), that is, of the activities which are sources of synchronization links having activity A as target, and (2) the delayed cost of A, which is the cost of A delayed by the costs of all its preceding activities (if any).

We pick a topological sort [21] activityList of activitySet — namely, any permutation of activitySet in which if activity A precedes activity B, then A is not the target of a link from B. It is worth noting that picking a topological sort guarantees that when an activity is evaluated, the outcomes and delayed costs of all its preceding activities have been already computed.

For each activity in activityList:

- We compute the list of its preceding activities and their overall outcome and cost $P = \text{Both}(\ldots (\text{Both}(\text{Zero}, c_1), c_2), \ldots), c_n)$ where $c_1, \ldots, c_n$ are the costs of the preceding activities.
- If the outcome of all preceding activities is Success and the joinCondition of activity is true, then activity is evaluated and its cost is set to $\text{Delay}(c, P)$, where $c$ is the cost of activity.
- If the outcome of all preceding activities is Success but the joinCondition of activity is false, then the cost of activity is set to $\text{Delay}(\text{Zero}, P)$, and its outcome to Success or Fault depending on whether its suppressJoinFailure attribute is true or false.
- If the outcome of some preceding activities is Fault or Stuck then the outcome of activity is Fault or Stuck, respectively\(^4\), and the cost of activity is set to

\(^4\)Fault has precedence over Stuck since a Stuck activity can be interrupted by a Fault from a parallel branch.
\text{Delay}(\text{Zero}, P).

- Finally, the status of the links outgoing from activity is determined.

If the outcome of some activity in the Flow is Fault then the outcome of the whole Flow is Fault. Otherwise, if the outcome of some activity in the Flow is Stuck then the outcome of the whole Flow is Stuck. Otherwise the outcome of the whole Flow is Success. The cost of the whole Flow is the Both composition of the costs all the activities in the Flow.

It is worth observing that, as the resolution of race condition is left unspecified in the WS-BPEL specification [11], we adopt a “pessimistic” approach\(^5\) in evaluating Flows and we skip an activity if one of its preceding activities is faulty or stuck.

### 4.3 Statistical non-determinism

In the previous section, for the sake of simplicity, we have implicitly assumed to be able to evaluate all conditions’ values. The evaluation of conditions usually depends on data values, which are unknown a priori, and invocation results can vary too from one execution to another. In this section we describe how Monte Carlo simulations can be employed to model different possible results of service invocations, as well as nondeterminism and correlations in workflows. We also present the pseudo-code for the analysis of \textit{Invoke} and \textit{OpaqueAssign} activities.

#### 4.3.1 Monte Carlo simulation

Monte Carlo simulation [15] is a technique that employs repeated random sampling to obtain numerical results. Generally, simulation is run multiple times in order to obtain the distribution of a probabilistic entity. We can compute an estimation of expected values (i.e., a probability weighted average of a function applied to all possible values) for many quantities by averaging the results of different iterations.

Monte Carlo simulation is useful for our algorithm in two ways. First, at each iteration of Monte Carlo we can sample the workflow branching conditions (for both alternatives and iterations) and deterministically decide what to execute. This, along with recursive sampling, allows us to address different possible results of service invocations, as well as nondeterminism and correlations in workflows. Second, many QoS properties can be

\(^5\text{In the terminology of [22], this makes our interpretation of WS-BPEL workflows “monotonic”, in the sense that the cost of an orchestrations does not decrease if the cost of an invoked service increases.}\)
written as expectation queries. For instance, reliability is the expected value of a sampling function that returns 1 if the outcome is success, and 0 otherwise. Average response time is the expected value of response time for the entire workflow. While Monte Carlo simulation will give an approximated result, it is possible to improve accuracy arbitrarily by increasing the number of samples. Moreover, the generation of samples is independent of previous iterations, and it can thus be run in parallel.

4.3.2 Sampling functions

A sampling function for a certain probability distribution is an algorithm which generates samples according to such a distribution. In a Monte Carlo simulation sampling functions are required to generate samples for random variables (i.e., variables which are assigned values randomly according to a certain distribution). In the context of WS-BPEL workflow analysis this is needed for the evaluation of OpaqueAssigns as well as for the outcome and cost of Invoke activities.

If we assume that the probability distributions of variable values and of invocation outcomes and costs (whatever definition of cost is employed to evaluate the QoS properties we are interested in) are known, then we can estimate the probability of an invocation throwing a fault or getting stuck, or the probability of a variable being true or false when evaluating an expression, as well as the total cost for the composition. By exploiting this information over distributions we can write a sampling function which generates each time a new sample for the value of the variable assignment (i.e., when evaluating assignment with non-deterministic expressions) or invocation outcome/cost (i.e., when evaluating Invoke activities), each time picking one of the possible values together with its probability. The sampling function for Invoke can generate samples for outcome in the form of Success, Fault and Stuck. On the other hand, the sampling functions for branching condition evaluation can generate samples in the form of true and false values which drive branch selection and loop continuation.

As an example, we show below the pseudocode of a sampling function for a Bernoulli distributed variable (that is, a boolean variable true with probability p) by exploiting a pseudo-random number generator:

```plaintext
let bernoulli p () =
  if generator.NextDouble() < p then
    true
  else
    false
```
SamplingFunCondition.[variableName]
< bernoulli probability

This can be similarly extended to create sampling functions for more complex domains (e.g., Invoke). In our proof-of-concept implementation (Chapter 5) we use a similar algorithm to generate a sampling function according to a list of possible values with associated probability that is specified as input.

In order to preserve correlations when implementing Monte Carlo algorithms, it is necessary to take care not to sample the same random variable twice during a single simulation run (that is, during a simulation run a variable should have a fixed value and not a different value each time it is evaluated). This is one of the reasons for which we store computed values for activities in a Flow instead of evaluating them when necessary. We only allow statistical non-determinism in two constructs of the language: Invoke and OpaqueAssign. For these activities the evaluation consists of just running the respective sampling functions, as we show below.

4.3.3 Invoke

Invoke (partnerLink) =>
let outcome, cost = SamplingFunInvoke.[partnerLink] ()
outcome, cost, env

The Invoke activity is used to call an external service via a partnerLink. We associate each endpoint with a sampling function. To evaluate the Invoke activity, the algorithm retrieves the sampling function for the associated endpoint from a dictionary, using its partnerLink, then executes it to generate a sample of outcome and cost.

4.3.4 OpaqueAssign

OpaqueAssign (variableName) =>
let value = SamplingFunCondition.[variableName] ()
Success, Zero, env.Add(variable, value)

WS-BPEL permits to specify hidden (opaque) assignments [11]. An opaque expression is a placeholder for a corresponding executable WS-BPEL expression. The OpaqueAssign activity that we consider represents a variable assignment using an opaque expression.

For the sake of simplicity, we assume here that different endpoints are identified by different partnerLink names.
The evaluation of \texttt{OpaqueAssign} initially calls the sampling function to generate a (true/false) sample, then it updates the environment by assigning the returned value to the variable. The \textit{outcome} of this activity is always a \texttt{Success} and it has \texttt{Zero cost}\footnote{In our implementation we actually allow opaque expressions also in \texttt{IfThenElse} and \texttt{While} guards. Indeed those opaque expressions are equivalent to an \texttt{OpaqueAssign} to a temporary variable, followed by a deterministic guard on such variable.}.

In our implementation (Chapter 5) we create sampling functions from a list of possible values and their probabilities. The algorithm however works with any procedure which allows to generate samples for the endpoint \textit{outcome} and \textit{cost}.

Keeping samples for activities instead of probabilities allows us to preserve correlations during the same Monte Carlo iteration (and the \texttt{environment} ensures that variable values are evaluated only once for each iteration).
Chapter 5

The PASO prototype

We have developed a proof-of-concept implementation of the algorithm described in Chapter 4 in the PASO (Probabilistic Analyser of Service Orchestration) open source application\(^1\). Although the algorithm can be implemented in any language, we chose F# because it is a strongly typed language and it conveniently supports the fast definition of inference rules.

![Diagram of PASO analyser with Workflow (.bpel), QoS and Probabilities (Annotations.xml), and Monte Carlo Simulation](image)

Figure 5.1: Bird-eye view of the input-output behaviour of the PASO analyser.

\(^1\)The source code of PASO is freely available at https://github.com/upi-bpel/paso.
5.1 Input of PASO

PASO inputs (Figure 5.1) a .bpel file containing a WS-BPEL process defining a service orchestration. As already mentioned, PASO is able to analyse a proper subset of WS-BPEL structured (Sequence, Flow, IfThenElse, While, Scope and faultHandlers) and basic activities (Assign, OpaqueAssign, Invoke, Receive and Reply\(^2\)).

PASO inputs also a .xml file containing annotations of probabilities for outcomes and costs of service invocations, as well as for the evaluation of the guard conditions of IfThenElse and While activities. Although not in the scope of this thesis, it is worth observing that such descriptions can be obtained from available repositories of service statistics\(^3\), from direct monitoring, or from SLAs [23]. While various proposals have been put forward for how to represent QoS values (e.g., [24, 25, 7, 26]), a simple XML structure was sufficient to represent QoS and probability distributions for our purposes.

The possible behaviour of service invocations is denoted by a list of event elements associating possible service outcomes (viz., success, fault or stuck) with probabilities and associated costs. For instance, the following annotation describes the behaviour of a service which successfully replies 79% of times in 1 second, 20% of times it replies in 2 seconds, and 1% it does not reply, and the expense associated with each invocation is 0.1$ in all cases.

```xml
<endpoint>
  <name>riskAssessor</name>
  <partnerLink>assessor</partnerLink>
  <event outcome="success" probability="0.79">
    <expense>0.1$</expense>
    <time>1 sec</time>
  </event>
  <event outcome="success" probability="0.20">
    <expense>0.1$</expense>
    <time>2 sec</time>
  </event>
  <event outcome="stuck" probability="0.01">
    <expense>0.1$</expense>
    <time>0 sec</time>
  </event>
</endpoint>
```

\(^2\)PASO assigns cost Zero and outcome Success to both Receive and Reply.

\(^3\)Such as:

http://www.wsdream.net/dataset.html,
https://www.planet-lab.org/planetlablogs,
http://www.uoguelph.ca/~qmahmoud/qws/
The probability that a guard condition of an IfThenElse or While activity evaluates to true is denoted by a condition element associating the condition with a probability. For instance, the following annotation states that the condition LoanRequest ≥ 10000$ will evaluate to true 50% of times.

```
<condition>
  <name>LoanRequest &gt;= 10000$</name>
  <value probability="0.5" >true</value>
</condition>
```

5.2 Behaviour of PASO

The code of PASO consists of a front-end executable, which parses the .bpel and the Annotations.xml files and generates the abstract syntax tree of the workflow and the sampling functions from the annotations. These are passed to a back-end library, which implements the exec function (to compute outcome and cost of the workflow) and the Monte Carlo algorithm. The resulting data are then passed back to the front-end executable, which displays them.

The overall behaviour of PASO consists of three main steps:

1. Simulation. In order to simulate multiple executions of the given workflow, PASO generates n samples (where n is the total number of Monte Carlo iterations), each denoting an execution trace of the workflow. The samples are generated with sampling functions from the input annotations. The result of each sample is a pair denoting outcome and cost of an execution trace.

2. Processing. The results of the samples are processed to calculate the desired QoS values (as illustrated in Sect. 6).

3. Output generation. Finally, PASO generates its output in the form of histograms and pie charts summarizing the results of the performed analysis. For instance, Figure 5.2 shows the screenshot of a PASO output where reliability is represented by a pie chart while expense and response time are represented by histograms.

---

4PASO does not require all guard conditions to be annotated. Non-annotated boolean conditions are
evaluated by PASO with respect to the current envirnoment.

\textsuperscript{3}PASO can be configured to generate also textual output and logs.

\textsuperscript{6}The x-axes represent QoS values in bins while the y-axes represent number of results which fall in that bin. Each histogram shows a distribution of a QoS attribute along with its average value (denoted by a vertical red line).
Chapter 6

Examples

In this chapter we show how the PASO analyser can be fruitfully exploited to get answers for the questions that we raised on the motivating example presented in Chapter 3 along with other three challenging examples. We will use a loan approval service (Section 6.3) and a shipping service example (Section 6.4), both adapted from two well-known examples provided in the original WS-BPEL specification [11] to illustrate how our approach go beyond the state of the art. We will also compare, in Section 6.3, the results predicted by PASO with experimental results for the loan approval service.

6.1 A cloud-based storage service (continued)

The first three questions raised in Chapter 3 concerned the quality of service of the orchestrator of Figure 3.1:

(Q1) What are the estimated reliability, cost, and response time of the orchestrator using services $C_1$ and $C_2$ of Table 3.1?

(Q2) What is the probability that the response time of the orchestrator will be more than 5 sec using services $C_1$ and $C_2$ of Table 3.1?

(Q3) Will the cost of the orchestrator exceed 0.04$ on average using services $C_1$ and $C_2$ of Table 3.1?

The results obtained by running PASO$^1$ on the orchestrator of Figure 3.1 and on the offerings of Table 3.1 are illustrated in Table 6.1 and Figure 6.1. The results reported in Table 6.1 are interesting as, for instance, we see that the estimated reliability of the orchestrator (99.53%) is higher than the reliability of both $C_1$ (99%) and $C_2$ (90%). This

$^1$We performed one million iterations of PASO for each group of questions.
is due to the fact that in the (less frequent, 40% of times) case of store requests the orchestrator tries twice to store on each \(C_i\) (if needed), and in the (more frequent, 60% of times) case of retrieve requests it succeeds if just one the \(C_i\) responds.

<table>
<thead>
<tr>
<th>Reliability</th>
<th>Cost</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.53%</td>
<td>0.038$</td>
<td>1.21 sec</td>
</tr>
</tbody>
</table>

Table 6.1: Results of PASO for (Q1).

Moreover, the histograms of Figure 6.1 show that:

- While the average response time (1.21 sec) of the orchestrator will be almost always (97.0%) less than 5 sec, there is a noticeable probability (about 3%) that it will exceed the maximum allowed time.

- The average cost is 0.038$, which is slightly below the target average expense of 0.04$.

Another class of interesting questions mentioned in Chapter 3 concerns comparing the effects of employing different external services on the QoS of an orchestrator:

(Q4) *Which offering of Table 3.2 will yield the best QoS (reliability, cost, response time) for the orchestrator of Fig. 3.1?*

The results obtained by running PASO on the orchestrator of Figure 3.1 and on the offerings of Table 3.2 are summarised in Table 6.2. Also in this case the results are interesting as for instance, despite the different reliabilities of the two offerings (90% and 90% vs. 99%
and 81%), we see that the reliability of the orchestrator is practically the same with either offering (while cost and response time differ).

<table>
<thead>
<tr>
<th></th>
<th>Reliability</th>
<th>Cost</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offer 1</td>
<td>98.3%</td>
<td>0.027$</td>
<td>1.48 sec</td>
</tr>
<tr>
<td>Offer 2</td>
<td>98.4%</td>
<td>0.034$</td>
<td>1.36 sec</td>
</tr>
</tbody>
</table>

Table 6.2: Results of PASO for (Q4).

A further class of interesting questions mentioned in Chapter 3 concerns assessing whether and how modifying the workflow of an orchestrator will impact on the overall QoS of the orchestrator:

(Q5) Extending the orchestrator so as to exploit one more cloud storage service $C_3$ (e.g., like the one described in Table 3.3) will increase the reliability of the orchestrator?

To answer this question, we used two alternative offerings for $C_1$ and $C_2$ (Table 3.2) and one offering for $C_3$ (Table 3.3). The results obtained by running PASO are summarised in Table 6.3. By comparing Tables 6.2 and 6.3, it is easy to conclude that adding a third storage service to the workflow is not a good idea as it decreases the QoS of the orchestrator.

<table>
<thead>
<tr>
<th></th>
<th>Reliability</th>
<th>Cost</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offer 1</td>
<td>96.9%</td>
<td>0.032$</td>
<td>1.82 sec</td>
</tr>
<tr>
<td>Offer 2</td>
<td>97.1%</td>
<td>0.037$</td>
<td>1.66 sec</td>
</tr>
</tbody>
</table>

Table 6.3: Results of PASO for (Q5).

6.2 Starting a manufacturing business

Let us consider a business process defining how to start a manufacturing business (Figure 6.2). The process, after receiving a user request, starts three activities in parallel:

- It invokes a RentalAgency service to find a suitable location for manufacturing the desired product,
- It invokes a LoanAgent service to ask for a loan to fund the business start up, and
- It invoke a HumanResourceAgency service to find personnel with relevant skills.
Only after the LoanAgent secures the loan, a BuySupplies service will be invoked. Furthermore, the process will invoke a RentLocation service only after both invocations to the RentalAgency service and to the LoanAgent service will have completed.

Similarly, the process will invoke a HireStaff service only after both invocations to the HumanResourceAgency service and to the LoanAgent service will have completed.

Finally, the process will reply to the user only after the invocations to the RentLocation service, to the BuySupplies service and to the HireStaff service will have completed.

Let us assume the following probability distributions (Table 6.4) for the completion time of the aforementioned activities: For instance, the HireStaff service is guaranteed to complete within 2 to 15 days. In most of cases (35%), it completes in 4 days. It can also complete in 2, 6, 7, 10, 12 or 15 days with probability of 10%, 10%, 15%, 15%, 10%, 5% respectively.

<table>
<thead>
<tr>
<th>Service</th>
<th>1 day</th>
<th>2 days</th>
<th>4 days</th>
<th>6 days</th>
<th>7 days</th>
<th>10 days</th>
<th>12 days</th>
<th>15 days</th>
</tr>
</thead>
<tbody>
<tr>
<td>RentalAgency</td>
<td>10%</td>
<td>30%</td>
<td>40%</td>
<td></td>
<td></td>
<td>20%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoanAgent</td>
<td>5%</td>
<td>20%</td>
<td>35%</td>
<td>20%</td>
<td>10%</td>
<td>5%</td>
<td>5%</td>
<td></td>
</tr>
<tr>
<td>HumanResourceAgency</td>
<td>10%</td>
<td>30%</td>
<td>10%</td>
<td>30%</td>
<td>20%</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RentLocation</td>
<td></td>
<td></td>
<td></td>
<td>10%</td>
<td>40%</td>
<td>10%</td>
<td>40%</td>
<td></td>
</tr>
<tr>
<td>BuySupplies</td>
<td>20%</td>
<td>15%</td>
<td>35%</td>
<td></td>
<td></td>
<td>20%</td>
<td>10%</td>
<td></td>
</tr>
<tr>
<td>HireStaff</td>
<td>10%</td>
<td>35%</td>
<td>10%</td>
<td>15%</td>
<td>15%</td>
<td>10%</td>
<td>5%</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.4: Probability distributions for the services invoked by the process of Fig. 6.2.
6.2. STARTING A MANUFACTURING BUSINESS

A natural question for this example is to estimate the time needed to complete the execution of the whole business process. It is worth observing that, since all the invoked services have complex dependencies with each other, answering questions such as:

(Q1) What is the expected time needed to execute the business process of Figure 6.2 under the hypotheses of Table 6.4?

(Q2) What is the probability that the business process will not complete in time for the advertised launch date (e.g., in 24 days)?

may be not easy.

The results obtained by running PASO on the orchestrator of Figure 6.2 and on the offerings of Table 6.4 are illustrated in Figure 6.3 and summarised in Table 6.5.

![Figure 6.3: Snapshot of PASO results for (Q1) and (Q2).](image)

<table>
<thead>
<tr>
<th>Probability of failing deadline</th>
<th>Average time</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.7%</td>
<td>18.68 days</td>
</tr>
</tbody>
</table>

Table 6.5: Summary of the results of PASO for (Q1) and (Q2).

The results are interesting as, for instance, we see that the estimated completion time of the business process is 18.68 days. The results also show that while the probability that the business process will complete in 24 days is 86.3%, there is a noticeable probability (13.7%) that it will not do so.
6.3 Loan approval service

We consider the following QoS properties\(^2\) for loan approval service:

- **Reliability** [2], intended as the probability of a workflow execution to be *successful*.

- **Amortized expense for successful execution** [2, 27], intended as the average expense for obtaining a successful execution of the workflow—which also includes the cost of unsuccessful attempts. For instance, if we consider 10 invocations, each with a $2 expense, and only 8 of them are successful, then the amortized expense for successful execution will be: \( \frac{2 \times 10}{8} = 2.5\$ \).

- **Average response time** [2, 27], computed only over successful executions.

The loan approval service (Figure 6.4) starts by receiving from a customer a loan request containing the amount requested together with some data of the customer.

---

\(^2\)Same properties also holds for shipping service example (Section 6.4)
If the loan requested is less than 10,000$ then the service invokes a *Risk Assessor* service to get an evaluation of the probability that the customer will not be able to repay the loan. If such probability is not higher than 10% then the service approves the loan request and replies to the customer.

Otherwise (viz., if the loan requested is not less than 10,000$ or if the probability that the customer will not be able to repay the loan is higher than 10%) then the service forwards the request to a *Loan Approver* service (a human accountant), and forwards to the customer the reply received. If the invoked *Loan Approver* returns a *fault*, the service will invoke a different *Loan Approver* endpoint until the latter successfully executes.

We assume the following input distributions:

- Workflow control-flow (Table 6.6a): Condition $\text{LoanRequest} \geq 10,000$ true 50% of times, condition $\text{riskAssessment} > 10\%$ true 60% of times.
- Risk Assessor service (Table 6.6b): Each invocation costs 0.1$. It usually (79%) completes in 1 second. Sometimes (20%) it takes 2 seconds because of congestion. Rarely (1%) it does not reply. In the latter case we (pessimistically) assume that usage is charged anyway.
- Loan Approver (Table 6.6c): The availability of this service is bound to the availability of a human accountant. If the latter is busy (15%), the invocation takes 5 minutes and returns a *fault*. Otherwise it takes a longer time and a cost proportional to it: 30% probability of completing in 10 minutes with 5$ expense, 35% probability of completing in 20 minutes with 10$ expense, and 20% probability of completing in 30 minutes with 15$ expense.

We now illustrate the results obtained by running PASO on the given example. The three desired QoS properties, namely, *reliability*, *amortized expense for successful execution*, and *average response time*, are defined as follows:

```ocaml
let mutable totalExpense = 0
let mutable successTime = 0
let mutable successCount = 0
for i = 1 to iterationCount do
  let outcome, (expense, time) = exec emptyEnv workflow
  totalExpense <- totalExpense + expense
  if outcome = Success then
    successTime <- successTime + time
    successCount <- successCount + 1
```

<table>
<thead>
<tr>
<th>condition</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoanRequest &gt;= 10,000$</td>
<td>50%</td>
</tr>
<tr>
<td>riskAssessment &gt; 10%</td>
<td>60%</td>
</tr>
</tbody>
</table>

(a) Workflow control-flow.

<table>
<thead>
<tr>
<th>outcome</th>
<th>expense</th>
<th>time</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>0.1$</td>
<td>1 sec</td>
<td>79%</td>
</tr>
<tr>
<td>Success</td>
<td>0.1$</td>
<td>2 sec</td>
<td>20%</td>
</tr>
<tr>
<td>Stuck</td>
<td>0.1$</td>
<td>-</td>
<td>1%</td>
</tr>
</tbody>
</table>

(b) Risk Assessor service

<table>
<thead>
<tr>
<th>outcome</th>
<th>expense</th>
<th>time</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>5$</td>
<td>10 min</td>
<td>30%</td>
</tr>
<tr>
<td>Success</td>
<td>10$</td>
<td>20 min</td>
<td>35%</td>
</tr>
<tr>
<td>Success</td>
<td>15$</td>
<td>30 min</td>
<td>20%</td>
</tr>
<tr>
<td>Fault</td>
<td>0$</td>
<td>5 min</td>
<td>15%</td>
</tr>
</tbody>
</table>

(c) Loan Approver service

Table 6.6: Input distributions for the loan approval service.

```plaintext
let reliability = successCount / iterationCount
let amortizedExpense = totalExpense / successCount
let averageResponseTime = successTime / successCount
```

We now describe step-by-step how the `exec` function computes one sample for the cost and outcome parameters. We then show how PASO performs Monte Carlo sampling to estimate the required QoS properties. Finally we show the result of performing such sampling over large numbers of samples.

In order to evaluate cost and outcome for the outermost Flow activity, PASO first computes the (delayed) costs of all the activities within the Flow.

The first executable activity in the Flow is the top-most Sequence of Figure 6.7 containing a Receive followed by an Assign. As the outcome of both Receive and Assign is Success, and they have Zero cost and no incoming links, PASO computes the delayed cost of the initial Sequence activity as \( \text{Both}(\text{Zero}, \text{Delay} (\text{Zero}, \text{Zero})) = \text{Zero} \).

PASO then samples a value for the bigAmount condition — say true in this evaluation instance — and stores it into the environment parameter, together with the status of the outgoing links (determined by evaluating their transition conditions) receiveToAssess and receiveToApproval, which is set to false and true, respectively.

Consider now the Sequence on the right of Figure 6.7. Since the status of its incoming
receiveToApproval link is true, its (implicit) joinCondition evaluates to true, and the Sequence is evaluated. Suppose that PASO samples \(<\text{Fault}, 0\$, 5 \text{ min}>\) as the result of the first invocation. This will cause a new invocation (to a different endpoint, assigned by the faultHandler). Suppose that PASO samples \(<\text{Success}, 5\$, 10\text{ min}>\) as the result of the second invocation, hence setting the While guard to false. PASO hence determines the delayed cost of the whole Sequence as:

\[
\text{Both}(<0\$, 5\text{min}>, \text{Delay}(<5\$, 10\text{min}>, <0\$, 5\text{min}>)) \\
= <5\$, 15\text{min}>
\]

Consider now the Sequence on the left of Figure 6.7. As the status of its incoming link is false, its (implicit) joinCondition evaluates to false. Since suppressJoinFailure is true [11], dead-path elimination is implemented by setting the outcome of the Sequence to Success, its cost to Zero, and the status of all its outgoing links to false. The delayed cost of the Sequence is hence \text{Both}(\text{Zero}, \text{Delay}(\text{Zero}, \text{Zero})) = \text{Zero}.

Consider finally the Reply action on the bottom. While Reply has no cost, its delayed cost is determined as:

\[
\text{Delay}(\text{Zero}, \text{Both}(<5\$, 15\text{min}>, \text{Zero})) \\
= <0\$, 15\text{min}>
\]

The overall cost of the Flow is hence the Both composition of the delayed costs of all the activities in the Flow:

\[
\text{Both}(\text{Zero}, \text{Both}(<5\$, 15\text{min}>, \text{Both}(\text{Both}(\text{Zero}, <0\$, 15\text{min}>)))) \\
= <5\$, 15\text{min}>
\]

Table 6.7 summarizes the previously described trace along with other five runs of the exec function on the example.

<table>
<thead>
<tr>
<th>bigAmount</th>
<th>highRisk</th>
<th>Risk Assessor</th>
<th>Loan Approver</th>
<th>Orchestration</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td></td>
<td>&lt;Fault, 0$, 5 \text{ min}&gt;; &lt;Success, 5$, 10 \text{ min}&gt;</td>
<td>&lt;Success, 5$, 15 \text{ min}&gt;</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>&lt;Success, 0.1$, 2 \text{ sec}&gt;</td>
<td>&lt;Success, 0.1$, 2 \text{ sec}&gt;</td>
<td>&lt;Success, 0.1$, 2 \text{ sec}&gt;</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>&lt;Success, 0.1$, 1 \text{ sec}&gt;</td>
<td>&lt;Success, 15$, 30 \text{ min}&gt;</td>
<td>&lt;Success, 15.1$, 181 \text{ sec}&gt;</td>
</tr>
<tr>
<td>true</td>
<td></td>
<td>&lt;Fault, 0$, 5 \text{ min}&gt;; &lt;Success, 10$, 20 \text{ min}&gt;</td>
<td>&lt;Success, 10$, 25 \text{ min}&gt;</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td></td>
<td>&lt;Success, 15$, 30 \text{ min}&gt;</td>
<td>&lt;Success, 15$, 30 \text{ min}&gt;</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
<td>&lt;Stuck, 0.1$, 0&gt;</td>
<td>&lt;Stuck, 0.1$, 0&gt;</td>
<td>&lt;Stuck, 0.1$, 0&gt;</td>
</tr>
</tbody>
</table>

Table 6.7: Six runs of the loan approval example.
The values of the QoS properties \textit{reliability}, \textit{amortized expense for successful execution} and \textit{average response time} computed from the samples of Table 6.7 are:

\[
\text{successTime} = (15 \cdot 60 + 2 + 181 + 25 \cdot 60 + 30 \cdot 60 + 0) \text{ sec}
= 6003 \text{ sec}
\]

\[
\text{totalExpense} = (5 + 0.1 + 15.1 + 10 + 15 + 0.1)\$ = 45.3\$
\]

\[
\text{reliability} = \frac{5}{6} = 83\%
\]

\[
\text{amortizedExpense} = \frac{45.3\$}{5} = 9.06\$
\]

\[
\text{averageResponseTime} = \frac{6003 \text{ sec}}{5} = 1200.6 \text{ sec}
\]

Table 6.8 shows how the estimations of the QoS parameters considered progressively converges by increasing the number of samples.

<table>
<thead>
<tr>
<th>samples</th>
<th>reliability</th>
<th>amortizedExpense</th>
<th>averageResponseTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>83%</td>
<td>9.06$</td>
<td>1200.6 sec</td>
</tr>
<tr>
<td>100</td>
<td>100%</td>
<td>6.80$</td>
<td>867.66 sec</td>
</tr>
<tr>
<td>10,000</td>
<td>99.3%</td>
<td>7.58$</td>
<td>940.75 sec</td>
</tr>
<tr>
<td>1,000,000</td>
<td>99.29%</td>
<td>7.60$</td>
<td>944.10 sec</td>
</tr>
</tbody>
</table>

Table 6.8: QoS estimations for different number of samples for the loan approval example.

**Experimental results**

To validate the results predicted by PASO we implemented and deployed the loan approval example by using Apache ODE [28] and Tomcat server [29] on a local server. A Java client program was used to invoke and monitor \textit{response time} and \textit{reliability} of both services (Figure 6.5).

![Figure 6.5: Activity diagram for monitoring the loan approval service.](image-url)
To reduce the total time to conduct the experiment, we limited the total number of runs to 10,000 and changed the time in Table 6.6c to 2, 4, 6 and 8 seconds. The results after monitoring response time for the loan approval service are plotted, along with PASO predicted result, in Figure 6.6. It is easy to see that PASO predicted results correspond to the monitored results. The slight variation in response time is due to the overhead associated with monitoring (which was not considered by PASO).

We also evaluated reliability and compared PASO results with measured values. Properly evaluating reliability is difficult in such a small number of runs: the number of unsuccessful executions was small, so the relative error on the measured value is big. The probability of unsuccessful execution predicted by PASO has a relative difference to the actual result of $0.17^3$, which is comparable with the error in measuring reliability with just 10,000 samples (Table 6.9).

<table>
<thead>
<tr>
<th>Measured value</th>
<th>99.47%</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASO prediction</td>
<td>99.56%</td>
</tr>
</tbody>
</table>

Table 6.9: Reliability comparison for loan approval service.

## 6.4 Shipping service

The shipping service (Figure 6.7) starts by receiving a shipping order from the customer. A shipping order contains a list of requested items together with the indication of whether the requested items must be shipped separately or all together. The service ships the orders as indicated and then it replies to the customer with a shipment complete notification.

\[^3\frac{0.44-0.53}{0.53} = 0.17, \text{ where } 100-99.56=0.44 \text{ and } 100-99.47=0.53\]
Figure 6.7: Shipping service.

We assume the following input distributions:

- **Workflow control-flow** (Table 6.10.(a)): Condition `ShipIndividual` true 70% of times, condition `Item<TotalItems` true 80% of times.
- **ShipItem service** (Table 6.10.(b)): It usually (80%) completes in 2 seconds with 0.5$ expense. Sometimes (20%) it returns a fault after 3 seconds.

<table>
<thead>
<tr>
<th>condition</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ShipIndividual</code></td>
<td>70%</td>
</tr>
<tr>
<td><code>Item&lt;TotalItems</code></td>
<td>80%</td>
</tr>
</tbody>
</table>

(a) Workflow control-flow

<table>
<thead>
<tr>
<th>outcome</th>
<th>expense</th>
<th>time</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Success</td>
<td>0.5$</td>
<td>2 sec</td>
<td>80%</td>
</tr>
<tr>
<td>Fault</td>
<td>0.0$</td>
<td>3 sec</td>
<td>20%</td>
</tr>
</tbody>
</table>

(b) ShipItem service

Table 6.10: Input distributions for the shipping service example.
We now describe step-by-step analysis of PASO for the shipping service example. For instance, if PASO samples the value false for ShipIndividual and <Success, 0.5$, 2 sec> for the invocation, then the total cost of the workflow is:

\[
\text{Both(Zero, Both(<0.5$, 2 sec>, Both(Zero, Zero)))} = <0.5$, 2 sec>
\]

On the other hand, if PASO samples the value false for ShipIndividual and <Fault, 0$, 3 sec> for the invocation, then the total cost of the workflow is:

\[
\text{Both(Zero, Both(<0$, 3 sec>, Both(Zero, Zero)))} = <0$, 3 sec>
\]

If instead PASO samples the value true for ShipIndividual and the body of the While is executed three times with <Success, 0.5$, 2 sec> for the invocation, then the cost of the While, which is also the cost of the whole workflow, is

\[
\text{Both(x, Delay(Both(x, Delay(x, x)), x))} = <1.5$, 6 sec>
\]

(where x= <0.5$, 2 sec>).

Table 6.11 summarizes the previously described three traces along with other three runs of the exec function on the example.

<table>
<thead>
<tr>
<th>ShipIndividual</th>
<th>item&lt;TotalItems</th>
<th>ShipItem</th>
<th>Orchestration</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td></td>
<td>&lt;Success, 0.5$, 2 sec&gt;</td>
<td>&lt;Success, 0.5$, 2 sec&gt;</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>&lt;Fault, 0$, 3 sec&gt;</td>
<td>&lt;Fault, 0$, 3 sec&gt;</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>&lt;Success, 0.5$, 2 sec&gt;</td>
<td>&lt;Success, 0.5$, 2 sec&gt;</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>&lt;Success, 0.5$, 2 sec&gt;</td>
<td>&lt;Success, 0.5$, 2 sec&gt;</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>&lt;Success, 0.5$, 2 sec&gt;</td>
<td>&lt;Success, 1.5$, 6 sec&gt;</td>
</tr>
<tr>
<td>false</td>
<td></td>
<td>&lt;Success, 0.5$, 2 sec&gt;</td>
<td>&lt;Success, 0.5$, 2 sec&gt;</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td></td>
<td>&lt;Success, 0$, 0 sec&gt;</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>&lt;Fault, 0$, 3 sec&gt;</td>
<td>&lt;Fault, 0$, 3 sec&gt;</td>
</tr>
</tbody>
</table>

Table 6.11: Six runs of the shipping service example.

The values of the QoS properties reliability, amortized expense for successful execution and average response time computed from the samples of Table 6.11 are:

- \(\text{successTime} = (2 + 3 + 6 + 2 + 0 + 3) \text{ sec} = 16 \text{ sec}\)
- \(\text{totalExpense} = (0.5 + 0 + 1.5 + 0.5 + 0 + 0)\$ = 2.5\$
- \(\text{reliability} = 4/6 = 66.6\%\)
- \(\text{amortizedExpense} = (2.5\$/4) = 0.6\$
- \(\text{averageResponseTime} = (16 \text{ sec})/4 = 4 \text{ sec}\)
Table 6.12 shows how the estimations of the QoS parameters considered progressively converges by increasing the number of samples.

<table>
<thead>
<tr>
<th>samples</th>
<th>reliability</th>
<th>amortizedExpense</th>
<th>averageResponseTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>66.6%</td>
<td>0.6 $</td>
<td>4 sec</td>
</tr>
<tr>
<td>100</td>
<td>68%</td>
<td>0.8 $</td>
<td>4.2 sec</td>
</tr>
<tr>
<td>10,000</td>
<td>62.8%</td>
<td>0.75 $</td>
<td>4.1 sec</td>
</tr>
<tr>
<td>1,000,000</td>
<td>62.9%</td>
<td>0.74 $</td>
<td>4.1 sec</td>
</tr>
</tbody>
</table>

Table 6.12: QoS estimations for different number of samples for the shipping service example.
Chapter 7

Related Work

The concept of Quality of Service (QoS) is not new and has been proposed in different domains (e.g., non-functional requirements in software engineering, QoS in networks and grid computing [30]). In the area of Web services and service-oriented computing, Papazoglou [31] and Menasce [32] were among the pioneering works which emphasised the need and challenges of QoS analysis. One of the strong applications of QoS in the area of Web services is to ensure quality guarantees in Service Level Agreements (SLA) (e.g., Michlmayr [33], Oriol [34]). Most of these approaches use some kind of monitoring service for sending notifications or taking corrective actions in case of degradation of QoS value(s). A limitation of these approaches is that not only they are time consuming but also they do not work for applications with side effects and they may involve monetary costs. In the following, we discuss only those approaches which predict QoS by analysing the workflow defining the application.

Some approaches reduce the given workflow to a single atomic activity by iteratively applying reduction rules. The most widely known is Cardoso et al. [35, 36]. Other approaches that are based on Cardoso et al. [35, 36] include Jaeger et al. [37], Hwang et al. [38] and Canfora et al. [39].

Cardoso et al [35, 36] developed one of the first approaches to QoS prediction for workflow systems. The composition workflow was a graph-like structure where nodes represent activities (or invoked services) while arrows represent transitions with associated probabilities. The three QoS attributes considered were time, cost and reliability, associated with each task. Their algorithm (called Stochastic Workflow Reduction, SWR) works by repeatedly applying a set of reduction rules on the given workflow until the entire workflow is reduced to one atomic activity. The set of reduction rules were grouped into six
CHAPTER 7. RELATED WORK

Figure 7.1: Example of Cardoso reduction rules for sequence of activities.

categories (1) sequential, (2) parallel, (3) conditional, (4) loop, (5) fault-tolerant (i.e., fault handling), and (6) network (i.e., nested structures). An example of reduction rules for sequence of activities (Fig. 7.1a) is shown in Fig. 7.1b.

Jaeger et al. [37] have proposed a similar technique. Their Stepwise Graph Transformation algorithm employs similar set of reduction rules as Cardoso et al. [35, 36]. However, with respect to [35, 36] their workflow model consists of 21 compositional structures based upon workflow patterns (van der Aalst [40]). The QoS attributes considered were time, cost, throughput and uptime.

Hwang et al. [38] proposed a similar workflow reduction algorithm as Cardoso et al. [35, 36]. The key improvement was in the generated output. While [35, 36] can output only average values, their algorithm can also generate output in terms of probability distribution functions (e.g., histograms) like our approach.

A major limitation of the previous workflow reduction approaches is that they do not deal with complex dependencies (e.g., those defined by WS-BPEL synchronization <link>s), which cannot be always reduced to parallel and sequential compositions (as mentioned in Chapter 1 and described by Mukherjee [12, 14] and Dumas [13]).

More recently Dumas et al. [13] and Mukherjee et al. [12, 14] tried to address the limitations of handling complex dependencies in workflow reduction techniques. Dumas et al. [13] technique requires the service orchestration to be represented as process graph while Mukherjee et al. [12, 14] approach directly inputs WS-BPEL.

In Dumas et al. [13] approach, each process graph is decomposed into a subgraph (single-entry and single-exit point SESE) in a tree structure (Refined Process Structure Tree (RPST) [41]). The QoS of each subgraph is individually calculated using Cardoso et al. [35, 36] aggregation function. The QoS of the service orchestration is then determined as the QoS of the root of the tree. Dumas et al. neither consider different results of service invocations nor non-determinism in workflows.
Mukherjee et al. [12, 14] approach transforms a WS-BPEL process into an activity graph where nodes are activities and links are dependencies between the activities. The activity graph is annotated by assigning probabilities of being executed to each activity. The dependency relationships among child activities are used to determine the probability of execution of the parent activity. The approach recursively applies aggregation formulas to determine the overall QoS. The QoS attributes considered by them include reliability, response time and cost. While Mukherjee’s approach can treat arbitrary complex dependency structures as well as fault-driven control-flow, they do not consider correlations among activities which do not have a direct dependency, and this may lead in some cases to incorrect results, as we mentioned in Chapter 1 (Figure 1.5).

Zeng et al. [42, 43] developed a middleware platform capable of driving service selection so as to maximize the QoS of the desired composition. Service compositions are represented by state charts. The composition is assumed to be well-structured and without any cycle. Parsing a given directed acyclic graph yields an execution path for which QoS attributes (reliability, time, cost, availability etc) were predicted. The main focus of their work was in the area of service selection and QoS prediction was needed to facilitate service selection. Canfora et al. [39] is another approach which is strongly based on aggregation functions proposed by Cardoso et al. [35, 36] and Zeng et al. [42, 43]. Both Zeng et al. and Canfora et al. did not consider different results of service invocations, neither non-determinism nor complex dependency in the workflow.

Zheng et al. [44, 45, 46] approach is similar Zeng et al. [42, 43], as QoS is calculated on the basis of evaluation of different execution paths in the service graph. In the initial step, a service graph is generated from given workflow. Then all sequential, loop and parallel nodes are replaced by a single node respectively. The service graph is transformed into a rooted tree where aggregation of QoS of every tree path yields overall QoS of the service composition. Zheng extended their approach to consider more complex dependency structures in [45, 46]. On the other hand Zheng et al. do not consider arbitrarily complex structures (such as those specified by WS-BPEL synchronization links), nor fault handling, and needs to fix an upper bound to the number of iterations of loops in order to allow decomposition into acyclic graphs. Zheng et al. also assumes service invocations to be deterministic (always successful with the same QoS).

Like Hwang et al. [38], Zheng et al. [44, 45, 46], and more recently Ivanović et al. [47] highlighted the importance of keeping results as distributions (rather than aggregating them into a single number) to be able to identify unlikely problematic cases which would be hidden by averages. Ivanović et al. [47] addressed the problem of correlation similarly
to us, namely by representing data as distributions over the possible environment states. They enumerate the possible states and evaluate explicitly their probability, which is fast and precise when the state space is small, but does not scale well for unbound states (e.g., for variable increasing loops). In contrast, we use only Boolean variables to keep the analysis simple, while still permitting to describe correlation. Ivanović et al. [48] is another approach that also makes use of simulation similarly to us. Their approach translates a given business process into a Petri net and then derives a composition model from it. The model is then annotated with QoS and probability information obtained from execution logs and monitored data. The model is then simulated to obtain distribution on execution times. Ivanović et al. [47, 48] do not consider parallel execution, nor complex dependency structures and fault handling.

Other approaches model a given application at various level of abstraction and then apply performance models (e.g., Markov Chains [49], Queuing Networks [50]) to analyse QoS properties. The approaches proposed by Marzolla [51], Gallotti [52], Klatt [53] and Franceschelli et al [54] fall in this category.

Marzolla [51] proposed queuing theory based techniques to derive performance bounds for response time and throughput. Gallotti [52] extended the idea by proposing a model-driven service composition technique which at abstract level makes use of UML activity diagrams to represent service orchestration while at implementation level uses WS-BPEL. Their approach transforms a given UML activity diagram into a markovian model which can be analysed by the PRISM model checker [55]. Their approach can predict reliability and time. A similar method was proposed by Gilmore [56] which models UML state charts as stochastic process algebra (PEPA) models [57] which are then analysed by PRISM [55].

Palladio Component Model [58] is another generic framework to model performance and reliability of component-based applications. Some approaches have used Palladio to predict QoS for service compositions. Most notable among them include Klatt [53] and Franceschelli et al [54]. Both of them use Queuing Network for performance modelling.

Like workflow reduction approaches, model-driven approaches are designed to work with well-structured orchestration models. Thus they may not work in the presence of complex dependencies in workflows. Furthermore, these approaches does not consider nondeterministic scenarios in the workflow.
Concluding remarks

Summing up, to the best of our knowledge, none of the existing approaches fully addresses all the four characteristics of service orchestrations that we listed in Chapter 1, namely: (1) different possible results of service invocations, (2) nondeterminism in orchestration workflows, (3) correlations among workflow activities, and (4) complex dependencies among activities. Moreover, all the aforementioned approaches require to know a priori an upper bound for each loop in order to estimate QoS values. Table 7.1 tries to summarise a qualitative comparison between our approach and the four most related approaches, viz., Cardoso et al [35, 36], Mukherjee et al. [12, 14], Zheng et al. [44, 45], Ivanović et al. [47].

<table>
<thead>
<tr>
<th>1. Different result of service invocation</th>
<th>[35, 36]</th>
<th>[12, 14]</th>
<th>[44, 45]</th>
<th>[47]</th>
<th>PASO</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Non-Determinism in the orchestration workflow</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>3. Correlations among workflow activities</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>partly</td>
<td>yes</td>
</tr>
<tr>
<td>4. Complex Dependency among workflow activities</td>
<td>no</td>
<td>yes</td>
<td>partly</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 7.1: Qualitative comparison among related work.
Part II

Parallel design patterns based applications
Developers writing parallel design patterns based applications would need to predict the QoS of their applications to better assess different design decisions. As the experimental evaluation of deployed applications is time and resource expensive, and it may not be always feasible, we propose an alternative approach to predict the QoS of parallel design patterns based applications based on Monte Carlo simulations [15]. Our analysis relies on expressing parallel design patterns based application in terms of the two cost compositors Both and Delay introduced in Part I.

This Part-II of the thesis is organized as follows: Chapter 8 covers background information on parallel design patterns while Chapter 9 discusses a motivating example. We present our algorithm in Chapter 10, and its proof-of-concept implementation (PASA) in Chapter 11. Chapter 12 discusses the results obtained by applying PASA on two motivating examples, while related work is discussed in Chapter 13.

The results presented in this part of the thesis was reported also in [59].
Chapter 8

Background: Parallel design patterns

Parallel design patterns (also called algorithmic skeletons) are pre-defined patterns encapsulating the structure of a parallel computation that are provided as building blocks to write applications [5, 6]. By definition, an algorithmic skeleton is a parametric, reusable and portable programming abstraction modelling a known, common and efficient parallelism exploitation pattern [60].

Three classes of parallel design patterns have been proposed: stream parallel, data parallel and control parallel [60].

- Stream parallel patterns deal with computations done on streams of data. A stream is a possibly infinite sequence of values, all of them having the same data type, e.g., a stream of files, network packets, bits, etc. Examples of stream parallel patterns include pipeline and farm.
- Data parallel patterns are used to speed up a single computation into parallel sub-tasks. Examples of data parallel patterns include map and reduce.
- Control parallel patterns are used to coordinate with other parallel design patterns. Examples of control parallel patterns include sequential and conditional patterns.

A parallel design patterns based application can be composed by employing patterns belonging to any class. In this thesis, we consider a core set of patterns belonging to stream parallel and control parallel classes.

8.1 A core set of parallel design patterns

The subset of parallel design patterns we consider is syntactically defined as follows:

\[
P ::= 0 \mid N \mid \text{Comp}(P,\ldots,P) \mid \text{Pipe}(P,\ldots,P) \mid \text{Farm}(P,n) \mid \text{Feedback}(P,\text{cond})
\]
where \( N \) is node, \( n \) represents number of workers in a farm and \( \text{cond} \) is the boolean condition for repetition in a feedback.

### 8.1.1 Node

Node is a basic activity which processes a stream of data items arriving on the input channel and delivers the results on the output channel (Fig. 8.1).

![Figure 8.1: Node](image)

### 8.1.2 Comp

Comp defines a sequence of activities to be sequentially executed one after the other. For example, consider the Comp of three nodes A, B and C (Fig. 8.2) with the size of input stream equal to 2 and where A, B, ... are nodes or parallel patterns. If the data items in the input stream are represented by 1 and 2, then execution order of Comp will be:

- A\(_1\) will process input 1.
- B\(_1\) will process input 1 (as soon as A\(_1\) finishes).
- C\(_1\) will process input 1 (as soon as B\(_1\) finishes).
- A\(_2\) will process input 2 (as soon as C\(_1\) finishes).
- B\(_2\) will process input 2 (as soon as A\(_2\) finishes).
- C\(_2\) will process input 2 (as soon as B\(_2\) finishes).

![Figure 8.2: Execution order of activities inside Comp.](image)

### 8.1.3 Pipe

Pipe is a pattern to represent pipelines of parallel (and potentially heterogeneous) activities. Similarly to Comp, activities are executed in order (i.e., once an activity has completed
processing a given input, such input can start being processed by the subsequent activity). It differs from Comp since an activity can start processing a new input once it has completed processing a given input, and while such given input is being processed by the subsequent activity\(^1\).

For example, consider a Pipe of three nodes A, B and C (Fig. 8.3) with the size of input stream equal to 2 and where A, B, ... are a nodes or parallel patterns. If the data items in the input stream are represented by 1 and 2, then the execution order of Pipe will be:

- A\(_1\) will process input 1.
- When A\(_1\) finishes, B\(_1\) can process input 1 and A\(_2\) can process input 2.
- When B\(_1\) finishes, C\(_1\) can process input 1.
- After both B\(_1\) and A\(_2\) finish, B\(_2\) can process input 2.
- After both C\(_1\) and B\(_2\) finish, C\(_2\) can process input 2.

![Figure 8.3: Execution order of activities inside Pipe.](image)

### 8.1.4 Farm

Farm (also called task-farm, or master-worker) represents multiple instances of an activity running in parallel. The instances are also called workers. Fig. 8.4 shows several instances of activity A running in Farm.

\(^1\)In a composition where pipe is inside a comp, pipe will behave as comp due to unavailability of input.
8.1.5 Feedback

Feedback (Fig. 8.5) is a pattern that can be used to selectively route back results to the start of the stream (i.e., once a given input has been processed, it is either returned to the output stream or routed back to the input stream, if condition condition holds).

Figure 8.5: Feedback.
Predicting the QoS of parallel design patterns based applications at design time can greatly facilitate application developers. On the other hand, predicting QoS is challenging for instance due to the non-deterministic nature of feedback loops. A feedback can occur zero or multiple times thus affecting the final QoS of an application. Similarly, the designer of an application has various choices between different design patterns at design time, and such choices can considerably affect the final QoS of the application.

To concretely illustrate these challenges, we now discuss an image filtering application (Fig. 9.1) [61], as an example of steady state where the number of workers is known in advance. Such application processes a video sequence (i) by reading each of its image frames, (ii) by applying two subsequent blurring filters $\text{Blur}$ and $\text{Blur2}$ to each image frame, and (iii) by writing each filtered image frame back to the disk. If the blurring of an image frame is not satisfactory, the frame has to be blurred again and is returned (through a feedback loop) to the input stream of the first blurring filter.

![Figure 9.1: An example of an image blurring application (with a feedback loop).](image)

A parallel implementation of such application can be developed by exploiting different configurations of parallel design patterns (e.g., via a pipeline of the four steps, via a farm
Table 9.1: (a) Energy and (b) time required by each application step for processing a given image frame.

<table>
<thead>
<tr>
<th>Step</th>
<th>Heavy</th>
<th>Light</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>1.84 mJ</td>
<td>1.84 mJ</td>
</tr>
<tr>
<td>Blur</td>
<td>14.44 mJ</td>
<td>4.99 mJ</td>
</tr>
<tr>
<td>Blur2</td>
<td>13.45 mJ</td>
<td>4.41 mJ</td>
</tr>
<tr>
<td>Write</td>
<td>1.95 mJ</td>
<td>1.95 mJ</td>
</tr>
</tbody>
</table>

Suppose, for instance, that we need to process two different types of image frames (viz., heavy and light), whose processing consumes different amounts of millijoules, and which require different time intervals for being completed by each step (Table 9.1).

Suppose also that our application is in a steady state where input frames arrive one right after the other (with negligible delay), that 10% of the image frames are heavy, and that the probability of a frame to be returned to the input stream is 0.3.

In the above illustrated situation, a natural question we would like to answer is the following:

(Q1) Given a certain parallelisation of our application, what are its energy consumption and completion time?

In Chapter 12 we will show the results of applying our algorithm to answer the above question and of one more challenging example.
Chapter 10

Predicting QoS of parallel design patterns based applications

10.1 Patterns syntax

To model parallel design patterns based compositions, we consider a subset of core activities (i.e., Node, Comp, Pipe, Farm, Feedback). An abstract syntax for them (written in F# [20]) is:

```fsharp
type QoS = float * float

type Activity =
| Node of (InputType->QoS)
| Comp of Activity list
| Pipe of Activity list
| Farm of Activity * int
| Feedback of Activity * string * float
```

where Comp and Pipe input a list of type Activity, Farm inputs an Activity and a number of workers, while Feedback inputs an activity, a guard condition (string) and an associated probability (float).

QoS is used to represent multiple QoS attributes (e.g., energy consumption and completion time) of an activity. For instance consider the application in our motivating example (Chapter 9). The (QoS of the) Blur node can be defined as follows:

```fsharp
let evaluateBlurQoS (inputType) : QoS =
    match inputType with
    | Heavy  -> (14.44, 3.991)
    | Light  -> (4.99, 0.377)

let Blur = Node(evaluateBlurQoS)
```
By defining also Read, Blur2, and Write, and by assuming 0.3 as the probability for the condition "blurred?" to get satisfied by an input item, we can for instance define the whole application as follows:

```plaintext
let motivatingExample = Pipe(Read, Feedback(Pipe(Blur, Blur2), "blurred?", 0.3), Write)
```

Then, `motivatingExample` can be given input to our probabilistic analysis, to estimate its QoS (as we will discuss in the remainder of this chapter).

### 10.2 Cost compositors

To estimate the QoS of parallel design patterns based applications, we will define (in Sect. 10.3) structurally recursive functions that associate, in a compositional way, each parallel design pattern with a cost structure. Please note that such cost structure is general, and it can be instantiated to define different QoS attributes, e.g., the time needed to complete an activity or the energy associated with its execution. We employ the same two cost compositors that we introduced in Chapter 4:

- The first cost compositor is a parallel compositor. \( \text{Both}(A, B) \) defines the cost associated with executing independently an activity with cost \( A \) and an activity with cost \( B \).
- The second cost compositor defines the delayed cost of an activity which must wait for the completion of another activity before starting. \( \text{Delay}(A, B) \) defines the cost associated with executing an activity of cost \( A \) which must wait for the completion of an activity with cost \( B \) before starting.

Let us denote with \( aTime \) and \( bTime \) the completion time of the two activities. The completion time for running both activities in parallel is given by the maximum between \( aTime \) and \( bTime \). Instead, the completion time for delaying one activity after the other is obtained by summing \( aTime \) and \( bTime \), as the delayed activity can start only after the first activity is completed.

```plaintext
let Both(aTime, bTime) = Max(aTime, bTime)
let Delay(aTime, bTime) = aTime + bTime
```

Let us denote with \( aEnergy \) and \( bEnergy \) the energy consumption of two activities. The energy consumed for independently executing \( \text{Both} \) activities can be approximated with the sum of their energy consumptions. On the other hand, the energy consumed to \( \text{Delay} \)
one activity after the other can be approximated by the energy consumption of the delayed activity (since — by abstracting from idle times — delaying an activity does not increase the energy it consumes).

\[
\text{let } \Both(a\text{Energy}, b\text{Energy}) = a\text{Energy} + b\text{Energy}
\]
\[
\text{let } \Delay(a\text{Energy}, b\text{Energy}) = a\text{Energy}
\]

As discussed in Part-I (Chapter 4), function \Both{} is commutative, \Both{} and \Delay{} are associative, and \Delay{} is right-distributive over \Both{}. We also explicitly name a neutral element Zero (i.e., \Both{}(A, Zero)=A and \Delay{}(A, Zero)=A), which can be useful at the time of variable initialization and start of iteration.

### 10.3 Evaluation of activities

We hereby introduce a recursive evaluation function \exec{} that estimates the QoS of a parallel application (defined as a combination of the core parallel design patterns \Node{}, \Comp{}, \Farm{}, \Pipe{} and \Feedback{} — Sect. 10.1). Essentially, \exec{} (i) reduces a parallel design patterns based application to a term of the cost operators \Both{} and \Delay{}, and (ii) determines the QoS to be returned by evaluating the obtained term according to the cost composition rules described in Sect. 10.2.

\[
\text{let rec } \exec \, (a:\text{Activity}, \, \text{startIndex: int}, \, \text{endIndex: int}):\text{QoS} =
\]
\[
\text{match } a \text{ with}
\]
\[
\text{| Node(evaluateQoS) } \rightarrow \ldots
\]
\[
\text{| Comp(aList)} \rightarrow \ldots
\]
\[
\text{| Pipe(aList)} \rightarrow \ldots
\]
\[
\text{| Farm \, (a,n) } \rightarrow \ldots
\]
\[
\text{| Feedback(a,cond,prob) } \rightarrow \ldots
\]

The \exec{} function inputs an \Activity{} a and the \startIndex{} and \ endIndex{} identifying the portion of the input stream to be processed. We assume the availability of a (global) list \inputStream{} of \InputTypes, which models the input stream to be processed.

#### 10.3.1 Node

The evaluation of a \Node{} activity consists of estimating the QoS of such node for processing the portion of the \inputStream{} identified by \startIndex{} and \ endIndex{}:

\[
\text{| Node(evaluateQoS) } \rightarrow
\]
\[
\text{let mutable nodeQoS = Zero}
\]
\[
\text{for i=\startIndex{} to \ endIndex{} do}
\]
let iQoS = evaluateQoS(inputStream.[i])
nodeQoS ← Both(iQoS,Delay(nodeQoS,iQoS))

Namely, we initially set the estimated nodeQoS to Zero. Then, for each item i from startIndex to endIndex, we evaluate the QoS iQoS of the current Node for processing item i, and we update nodeQoS with the cost of executing the current item i delayed after the previous items. Once all items from startIndex to endIndex have been processed, nodeQoS is returned.

10.3.2 Comp

A Comp activity inputs a list of activities (aList). Its QoS can be estimated (i) by simulating the processing of each item of the given portion of the input stream by such a sequence of activities, and (ii) by ensuring that an item i starts being processed by the first activity in aList only when the last activity in aList has completed processing the preceding input i-1.

| Comp(aList) =>
| let mutable compQoS = Zero
| for i = startIndex to endIndex do
| | let mutable aListQoS = Zero
| | for a in aList do
| | | let aQoS = exec (a,i,i)
| | | aListQoS ← Both(aListQoS,Delay(aQoS,aListQoS))
| | | compQoS ← Both(compQoS, Delay(aListQoS,compQoS))
| | compQoS

More precisely, the estimated compQoS is initially set to Zero. Then, we compute the QoS (aListQoS) for processing each item i in the portion of the input stream identified by startIndex and endIndex. Such aListQoS is simply obtained by executing each activity a in aList on the given input stream item i, and by delaying the cost of each activity after the preceding one. Then, compQoS is updated with the cost aListQoS for the current item i delayed after the cost for processing the previous items of the input stream. As soon as all items from startIndex to endIndex have been processed, compQoS is returned.

10.3.3 Pipe

A Pipe activity also inputs a list of activities. Its QoS can be estimated (i) by determining the cost of each of its stages for processing all items of the input stream from startIndex
to endIndex, and (ii) by composing such costs with Both.

\[
\text{Pipe}(aList) \rightarrow
\begin{align*}
\text{let mutable pipeQoS} &= \text{Zero} \\
\text{for a in aList do} & \\
\text{let } aQoS &= \text{exec(a, startIndex, endIndex)} \\
\text{pipeQoS} &= \text{Both(pipeQoS, aQoS)} \\
\text{pipeQoS}
\end{align*}
\]

Namely, we estimate the cost \(\text{pipeQoS}\) of a Pipe by computing, for each activity \(a\) in \(aList\), the cost \(aQoS\) for executing \(a\) over all items in the given portion of the input stream (from \(\text{startIndex}\) to \(\text{endIndex}\)). Then, we compose all computed \(aQoS\) with Both.

The reason why we can compose all activities’ costs with Both is that the cost of a Pipe can be approximated by the cost of independently executing all its stages on all the items in the given portion of the input stream\(^1\).

### 10.3.4 Farm

To estimate the QoS of a Farm (with a round robin scheduling policy) with \(n\) workers\(^2\), (i) we partition the input stream among its workers, and (ii) we compute the cost for concurrently executing such workers, each processing the portion of the input stream it has been assigned to.

\[
\text{Farm}(a, n) \rightarrow
\begin{align*}
\text{let mutable farmQoS} &= \text{Zero} \\
\text{let workerStreamSize} &= (\text{endIndex} - \text{startIndex} + 1) / n \\
\text{for } w=0 \text{ to } n-1 \text{ do} & \\
\text{let } wStartIndex &= \text{startIndex} + w \times \text{workerStreamSize} \\
\text{let } wEndIndex &= wStartIndex + \text{workerStreamSize} - 1 \\
\text{let mutable wQoS} &= \text{Zero} \\
\text{if } (wEndIndex < \text{inputStream.Length}) \text{ then} & \\
\text{wQoS} &= \text{exec(a, wStartIndex, wEndIndex)}
\end{align*}
\]

---

\(^1\)For instance, Both takes the maximum among the values of completion time, and this means that we estimate the completion time of a Pipe as the maximum among the completion times of its stages. Since completion times corresponds to multiplying the stages’ service times by the size of the stream to be processed, we are actually estimating the completion time of a Pipe as the maximum among service times multiplied by the size of the stream to be processed. We also abstract from the waiting time for the item to arrive as it is negligible due to large size of the input stream.

\(^2\)For the sake of simplicity, when evaluating a farm with \(n\) workers, we assume the availability of \(n\) processing cores, thus we abstract from any delay due to the scheduling of \(n\) workers over \(m < n\) processing cores.
else
wQoS <- exec(a, wStartIndex, inputStream.Length)
farmQoS <- Both(farmQoS, wQoS)
farmQoS

Namely, we compute the size (workerStreamSize) of the portion of the input stream to be assigned to each worker. Then, for each worker \( w \), we evaluate its QoS by estimating the cost required by the activity \( a \) (which is running on \( w \)) for processing the portion of the input stream assigned to \( w \). Since all workers run in parallel, the cost (farmQoS) of the considered Farm is obtained by composing with Both all the costs wQoS required by the workers.

One may note that, strictly speaking, the above snippet does not implement a round robin policy, as the input stream is split into contiguous segments assigned to different workers. It is worth noting that, as we are setting up a Monte Carlo simulation (Section 10.4), the inputStream will be sampled according to a given probabilistic distribution. Moreover, we are interested on predicting the QoS based on the distribution of the input types in the input stream, rather than on the “identity” of each input stream item. This, along with the facts that the size of the stream is usually much bigger than the amount of workers, and that each simulation is going to be repeated a huge number of times [15], justifies evaluating a Farm by simply splitting the input stream among the workers.

10.4 Statistical non-determinism

In this section we describe how Monte Carlo simulations can be employed to model different types of inputs in input stream, as well as nondeterminism in parallel design patterns. We also present the pseudo-code for the analysis of Feedback activities.

As discussed in Part-I (Chapter 4), Monte Carlo simulation [15] is a technique that employs repeated random sampling to obtain numerical results. Generally, simulation is run multiple times in order to obtain the distribution of a probabilistic entity. We can compute an estimation of expected values (i.e., a probability weighted average of a function applied to all possible values) for many quantities by averaging the results of different iterations.

Monte Carlo simulation is useful for our algorithm in the following ways. At each iteration of Monte Carlo we can sample the conditions for the types of input stream and for feedback conditions. This allows us to model different possible results of node executions.
due to different types of input, as well as feedback nondeterminism. While Monte Carlo simulation will give an approximated result, it is possible to improve accuracy arbitrarily by increasing the number of samples. Moreover, the generation of samples is independent of previous iterations, and it can thus be run in parallel.

10.4.1 Sampling functions

As discussed in Part-I (Chapter 4), a sampling function for a certain probability distribution is an algorithm which generates samples according to such a distribution. In a Monte Carlo simulation sampling functions are required to generate samples for random variables (i.e., variables which are assigned values randomly according to a certain distribution). In the context of parallel design patterns, they are needed for:

- **Modelling the types of inputs in input stream.**

  In section 10.3, we assumed the availability of a (global) list `inputStream` of `InputTypes`, which models the input stream to be processed. The purpose of `inputStream` was to distinguish between different categories of input items, each requiring different values of energy consumption and completion time. Such `inputStream` can be generated by exploiting a sampling function.

  Consider the input stream in our motivating example (Chapter. 9), where 10% of the input image frames are *heavy*, and the remaining 90% are *light*. Suppose also that we want to build a small stream containing of 10000 image frames. The F# code to create a corresponding `inputStream` is the following.

  ```fsharp
  type InputType = Heavy | Light
  let rand = new System.Random()
  let samplingFunction i =
    if rand.NextDouble() <= 0.1 then Heavy
    else Light
  let inputStream = Array.init 10000 samplingFunction
  
  Namely, we declare the `InputTypes` (Heavy and Light) and we define a sampling function, which returns `Heavy` with probability 0.1 and `Light` with probability 0.9. Then, we create an `inputStream` whose size is 10000 and whose content is filled according to the specified `samplingFunction`.

- **Sampling function for feedback conditions.** Please see in next subsection.
CHAPTER 10. PREDICTING QOS OF PARALLEL DESIGN PATTERNS BASED APPLICATIONS

10.4.2 Feedback

A Feedback activity inputs the Activity a to be executed, and a condition cond that causes a processed input item to be routed back to the input stream with a certain probability prob. The actual satisfaction of a condition cond is unknown a priori (as it depends on the actual items to be processed), and this introduces non-determinism when analysing a Feedback activity. To deal with such non-determinism, we rely on the probability prob to determine how many items are probably routed back to the input stream and to repeat the analysis over such items.

\[
\text{Feedback}(a, \text{cond}, \text{prob}) \rightarrow
\]

\begin{verbatim}
let mutable feedbackQoS = exec(a, startIndex, endIndex)
let mutable f = 0
for i=startIndex to endIndex do
  if (sampleCondition(cond, prob)) then f <- f+1
  if f > 0 then
    let fQoS = exec(Feedback(a, cond, prob), endIndex-f+1, endIndex)
    feedbackQoS <- Both(feedbackQoS, Delay(fQoS, feedbackQoS))
feedbackQoS
\end{verbatim}

More precisely, we initially set the cost feedbackQoS of the evaluated Feedback activity to the cost required by the inner activity a for processing the given portion of the input stream. We then compute f as the amount of items of the input stream that have to be routed back to the input stream. Then, to simulate the analysis of the f feedbacked items, we re-evaluate the current Feedback over the last f items of the input stream\(^3\). Such analysis results in a cost fQoS that is added to feedbackQoS (i.e., the cost of the feedbacked f items is delayed after the previously computed cost feedbackQoS) to compute the overall cost of the current Feedback activity.

To compute the amount f of items to be fed back to the input stream, we exploit another sampling function, which simulates the non-deterministic behaviour of the feedback condition. It returns true (cond holds) with probability prob. It returns false otherwise.

\begin{verbatim}
let sampleCondition(cond:string, prob:float) =
  let rand = new System.Random()
  if rand.NextDouble() <= prob
  then true
  else false
\end{verbatim}

\(^3\)As we highlighted in Sect. 10.3.4, we are interested on predicting the QoS based on the distribution of the input types in the input stream, rather than on the “identity” of each input stream item. Hence, it is enough to repeat our analysis on f items (most probably) satisfying such a probability distribution, such as the last f items of the given portion of the input stream.
10.4. STATISTICAL NON-DETERMINISM

if rand.NextDouble() <= prob then true
else false

10.4.3 Setting up the Monte Carlo simulations

In the previous sections we have shown how to generate the inputStream and how to evaluate the QoS of an Activity a (i.e., a given composition of parallel design patterns) for processing a given portion of such inputStream. We now need to put the pieces altogether for setting up a Monte Carlo simulation that permits predicting QoS of parallel design patterns based applications.

let predictQoS(a: Activity, size: int, samplingFunction: int -> InputType) =
let inputStream = Array.init size samplingFunction
let rec exec (a: Activity, startIndex: int, endIndex: int): QoS = . . .
exec(a, 0, size - 1)

The function predictQoS inputs the Activity a whose QoS has to be predicted, and the size of the inputStream to be processed. It assumes the availability of a global samplingFunction that permits populating the input stream according to a given distribution of input types. After defining the function exec, predictQoS simply invokes such function to estimate the QoS of a for processing the whole inputStream.

Obviously, one invocation of predictQoS simulates a single execution of a. According to the Monte Carlo simulation theory [15], a reliable prediction can be obtained if we repeat a simulation a huge amount of times (each of which re-samples both the input stream and the feedback conditions) and properly aggregate the computed results. As one can expect, the higher the amount of iterations, the better the accuracy of the performed Monte Carlo simulation [15].

Consider again our motivating example (Chapter 9). As stated at the start of this chapter, we can define our application as follows:

let motivatingExample =
Pipe(Read, Feedback(Pipe(Blur, Blur2), "blurred?", 0.3), Write)

Suppose that, for instance, we now want to predict its energy consumption and completion time over an input stream. To do it, we can iterate a huge\(^4\) amount of times the invocation of predictQoS and then average all estimated values of energy consumption

\(^4\)1 million and higher
and completion time.

```fsharp
let iterations = 1000000
let mutable avgEnergy = 0.0
let mutable avgTime = 0.0
for i = 0 to iterations - 1 do
    let iQoS = predictQoS(motivatingExample, 10000, samplingFunction)
    avgEnergy <- avgEnergy + (fst iQoS)
    avgTime <- avgTime + (snd iQoS)
    avgEnergy <- avgEnergy / (float iterations)
    avgTime <- avgTime / (float iterations)
```

The above snippet of F# code shows how to compute the expected energy consumption \( \text{avgEnergy} \) and completion time \( \text{avgTime} \) of our motivating application for processing an input stream that contains 10000 images, and whose desired input type distribution is obtained by exploiting the given \text{samplingFunction}.
Chapter 11

The PASA prototype

We have developed a proof-of-concept implementation of the algorithm described in Chapter 10 in the PASA (Probabilistic Analyser of Skeleton based Applications) open source\(^1\) application (Fig. 11.1). Like for PASO\(^2\), we also chose F# for PASA because it is a strongly typed language and it conveniently supports a fast definition of inference rules.

![Figure 11.1: Bird-eye view of the input-output behaviour of the PASA analyser.](image)

11.1 Input of PASA

As previously mentioned (Chapter 10), PASA is able to analyse a subset of stream parallel patterns (Comp, Pipe, Farm, Feedback) and basic activities (Node). PASA requires the

\(^1\)The source code of PASA is publicly available at [https://github.com/ahmad1245/PASA](https://github.com/ahmad1245/PASA).

\(^2\)While PASA employs the same Both and Delay structures of PASO we preferred to have separate prototype.
following inputs:

1. The description of a parallel application defined as a combination of basic activities (i.e., Nodes) with the stream parallel design patterns Comp, Pipe, Farm, Feedback.

2. The size and the optional classification of the input stream (e.g., in our motivating example, the data items of the input stream are categorised as heavy or light).

3. The QoS required by each node to process each type of input, and

4. The probabilities of a given input type to occur and of a given Feedback condition to get satisfied.

11.2 Behaviour and output of PASA

Given the above input, PASA predicts the QoS of the specified application for processing the input stream. More precisely, the input is passed to a back-end, which implements the recursive function exec and the Monte Carlo algorithm. The latter permits simulating multiple execution of the given composition of parallel design patterns on the given input stream. PASA generates \( n \) samples\(^3\) (where \( n \) is exploited by users to indicate the total number of Monte Carlo iterations to be performed), each denoting an execution trace of the given composition. By applying exec to each sample, PASA computes the QoS (i.e., energy consumption and completion time) of each execution trace. The QoS computed for all traces are then aggregated to derive the desired prediction of QoS. For instance, by simply averaging all QoS values, it is possible to estimate the average energy consumption and the average completion time required by the specified application for processing the given input stream.

PASA also permits displaying the results of the performed analysis in different formats. For instance, it is possible to display such results in the form histograms summarising the probability distribution of the estimated QoS — Fig. 11.1).

---

\(^3\)The samples are generated with sampling functions following the probability distributions of input types and of Feedback conditions' satisfaction.
Chapter 12

Examples

In this chapter we show how the PASA analyser (Chapter 11) can be fruitfully exploited to get answers for the questions that we raised on the motivating example presented in Chapter 9. We will also compare, in Section 12.1, the results predicted by PASA with experimental results for an image filtering example. We will then present a second example of dynamic reconfiguration example (Section 12.2) to illustrate a non-steady state scenario.

12.1 An image filtering application (continued)

Consider again the image processing application in our motivating example (Chapter 9), which has to process a stream containing 1000 image frames of two different types (viz., heavy and light), each requiring different QoS to be processed. Suppose that we want to parallelise our application, and to qualitatively compare different compositions of parallel design patterns to select the one most probably yielding the higher degree of QoS.

First, we measured the energy and time required by each application step of our application to process an image frame of a given type. All measurements were performed in isolation on a sequential version of the image processing application. The resulting values for the considered QoS attributes are reported in Table 9.1.

We defined a first parallel version of our application in PASA as follows:

```haskell
let motivatingExample =
    Pipe(Read, Feedback(Pipe(Blur, Blur2), "blurred?", probF), Write)
```

where probF is the probability of the "blurred?" condition to get satisfied by a processed item.
### Table 12.1: Completion times for a pipelined implementation of our motivating example application.

<table>
<thead>
<tr>
<th>Perc. of heavy items</th>
<th>Prob. of feedback</th>
<th>Predicted time (msec)</th>
<th>Measured time (msec)</th>
<th>Relative error</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>0.1</td>
<td>8201.0</td>
<td>8344.4</td>
<td>1.718%</td>
</tr>
<tr>
<td>10%</td>
<td>0.2</td>
<td>9214.5</td>
<td>9301.5</td>
<td>0.935%</td>
</tr>
<tr>
<td>10%</td>
<td>0.3</td>
<td>10536.4</td>
<td>11256.9</td>
<td>6.400%</td>
</tr>
<tr>
<td>10%</td>
<td>0.4</td>
<td>12275.5</td>
<td>12528.9</td>
<td>2.023%</td>
</tr>
<tr>
<td>20%</td>
<td>0.1</td>
<td>12236.8</td>
<td>12395.5</td>
<td>1.280%</td>
</tr>
<tr>
<td>20%</td>
<td>0.2</td>
<td>13738.1</td>
<td>14324.5</td>
<td>4.093%</td>
</tr>
<tr>
<td>20%</td>
<td>0.3</td>
<td>15703.9</td>
<td>16457.9</td>
<td>4.582%</td>
</tr>
<tr>
<td>20%</td>
<td>0.4</td>
<td>18329.2</td>
<td>18657.8</td>
<td>1.761%</td>
</tr>
<tr>
<td>30%</td>
<td>0.1</td>
<td>16238.4</td>
<td>16606.4</td>
<td>2.216%</td>
</tr>
<tr>
<td>30%</td>
<td>0.2</td>
<td>18250.3</td>
<td>18464.0</td>
<td>1.157%</td>
</tr>
<tr>
<td>30%</td>
<td>0.3</td>
<td>20851.6</td>
<td>20822.8</td>
<td>0.138%</td>
</tr>
<tr>
<td>30%</td>
<td>0.4</td>
<td>24341.3</td>
<td>24280.2</td>
<td>0.252%</td>
</tr>
<tr>
<td>40%</td>
<td>0.1</td>
<td>20259.8</td>
<td>20404.9</td>
<td>0.711%</td>
</tr>
<tr>
<td>40%</td>
<td>0.2</td>
<td>22775.6</td>
<td>23109.0</td>
<td>1.443%</td>
</tr>
<tr>
<td>40%</td>
<td>0.3</td>
<td>26069.7</td>
<td>26023.0</td>
<td>0.180%</td>
</tr>
<tr>
<td>40%</td>
<td>0.4</td>
<td>30391.6</td>
<td>30342.9</td>
<td>0.160%</td>
</tr>
</tbody>
</table>

We compared the values of QoS predicted by PASA\(^1\) (for different values of \(\text{probF}\)) with respect to those directly measured on an equivalent, synthetic implementation of the same application in FastFlow [62]. As shown in Table 12.1, the predicted value for completion time is quite accurate\(^2\).

Consider now Fig. 12.1. The figure shows that, if we fix the amount of expected heavy input stream items (either to 10% or to 20%), and if we vary the probability of the feedback condition "blurred?" to get satisfied by a given item, PASA is capable of effectively identifying the trend of the multiple QoS attributes it simultaneously analyses. Since this holds independently of the magnitude of the relative error, PASA can be exploited by parallel application developers who aims at qualitatively analysing multiple dimensions of the QoS of different parallel design patterns compositions.

---

\(^1\)Such values have been obtained by performing 1000 iterations of the Monte Carlo simulation.

\(^2\)Similar results are achieved when predicting the energy consumption of our application, even if with an average relative error ten times higher with respect to that affecting the predicted completion time (due to the extremely simplistic model we employed for estimating energy consumption).
12.1. AN IMAGE FILTERING APPLICATION (CONTINUED)

Figure 12.1: Growth trends of the considered multiple QoS attributes (with fixed percentage of heavy items and varying probability of the feedback condition "blurred?" to get satisfied by an input item).
To show an example of a qualitative analysis that we can perform with PASA, suppose that we want to assess the advantages of parallelising our motivating example application. Suppose that we have to process 1000 image frames (45% of which are heavy), and that we can exploit at most 16 processing units. Obviously, we can implement our application as (C) a composition of Comp and Feedback, or as (P) a composition of Pipe and Feedback:

```plaintext
let C = Comp(Read, Feedback(Comp(Blur, Blur2), "blurred?", 0.2), Write)
let P = Pipe(Read, Feedback(Pipe(Blur, Blur2), "blurred?", 0.2), Write)
```

We can also decide to build a Farm, each of whose workers is a composition of Comp and Feedback. In this case, since we can exploit at most 16 processing units, since each worker is a Comp (thus requiring a single processing unit), and since we have to spend two processing units for the Farm's emitter and collector, we can instantiate from 2 to 14 workers:

```plaintext
let F2C = Farm(Comp(Read, Feedback(Comp(Blur, Blur2), "blurred?", 0.2), Write), 2)
...
let F14C = Farm(Comp(Read, Feedback(Comp(Blur, Blur2), "blurred?", 0.2), Write), 14)
```

Similarly, we can build a Farm whose workers are compositions of Pipe and Feedback. In this case, since each worker requires 4 processing units (one for each stage of the Pipe), and since we have to spend two processing units for the Farm's emitter and collector, we can only instantiate either 2 or 3 workers:

```plaintext
let F2P = Farm(Pipe(Read, Feedback(Pipe(Blur, Blur2), "blurred?", 0.2), Write), 2)
let F3P = Farm(Pipe(Read, Feedback(Pipe(Blur, Blur2), "blurred?", 0.2), Write), 3)
```

Another possible combination of patterns is a Pipe whose first and last stages are Read and Write, and with an intermediate stage that is a Farm whose workers are a Comp of Blur and Blur2. Due to the same restriction on processing units as above, we can instantiate from 2 to 12 workers in the intermediate Farm:

```plaintext
let PF2C = Pipe(Read, Farm(Feedback(Comp(Blur, Blur2), "blurred?", 0.2), 2), Write)
...
let PF12C = Pipe(Read, Farm(Feedback(Comp(Blur, Blur2), "blurred?", 0.2), 12), Write)
```

Finally, we can derive a different combination of patterns by simply exploiting Pipes instead of Comps to implement the workers in the intermediate Farm of the above Pipe.
Since we increase the amount of processing units required by each worker, we need to decrease the maximum amount of workers that we can instantiate to 6:

```
let PF2P = Pipe(Read, Farm(Feedback(Pipe(Blur, Blur2), "blurred?", 0.2), 2), Write)
...
let PF6P = Pipe(Read, Farm(Feedback(Pipe(Blur, Blur2), "blurred?", 0.2), 6), Write)
```

By running PASA, we can estimate the expected completion time for all identified combinations of parallel design patterns. We can then compare such combinations in the bidimensional space in Fig. 12.2 (where the x axis denotes the predicted completion time, while the y axis denotes the amount of processing units needed to run a given pattern combination) to assess the effect of increasing the amount of employed computing resources. Furthermore, by including additional QoS properties (e.g., energy consumption)

![Figure 12.2: An example of comparison for different composition of patterns.](image)

in the analysis (thus transforming the bidimensional space in Fig. 12.2 in an n-dimensional space) we can gain further information to decide whether to use one pattern combination or another.
12.2 Dynamic reconfiguration

As a second example, we consider a non-steady state of an application that dynamically adapts its configuration when the input bandwidth changes in order to optimize completion time and energy consumption.

Suppose that bandwidth\(^3\) is classified into three categories (“Low”, “Medium”, and “High”) with a probability distribution, as illustrated in Table 12.2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bandwidth Range</th>
<th>Probability(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>[0, 76.77)</td>
<td>90.029</td>
</tr>
<tr>
<td>Medium</td>
<td>[76.77, 135.93)</td>
<td>9.513</td>
</tr>
<tr>
<td>High</td>
<td>[135.93, 203.53)</td>
<td>0.457</td>
</tr>
</tbody>
</table>

Table 12.2: Bandwidth categories.

Suppose also that the application is defined by the pattern `Farm(A,n)` where \(n\) is the number of workers for activity \(A\) (Fig. 12.3), and consider the possible configurations\(^4\) listed in Table 12.3 and the following cost model for configuration switching:

\[
\frac{f_{\text{max}}}{f_{\text{previous}}} \times L \times (1 + \frac{1}{W_{\text{previous}}})
\]

where

- \(f_{\text{max}}\) is the maximum frequency for all configurations
- \(f_{\text{previous}}\) is the frequency of the previous configuration
- \(L\) is the average latency required to process a task (we assume \(L = 80\text{ms}\) in this example)
- \(W_{\text{previous}}\) is the number of workers of the previous configuration

\(^3\)Bandwidth is expressed in \# tasks/secs

\(^4\)Frequencies are expressed in Hz, completion time in secs, energy consumption in joules.

The values for different configurations came from experiments executed by implementing the application in FastFlow. The energy was measured on the real runs by using Mammut library [63]. Out of total 286 configurations, we randomly selected 39 configurations to reduce time to run the experiment.
A natural question is:

(Q1) Which are the configurations that provide optimal QoS (i.e., minimal completion time and energy consumption) for each bandwidth type?

It is worth noting that to answer the above question it is necessary to take into account both the costs of configurations and the costs of configuration switching, and that the way in which the application will switch configurations is not known a priori as it depends on the input.

The results obtained by running PASA on the configurations listed in Table 12.3 for the inputs of Table 12.2 are illustrated Table 12.4. We performed 10,000 iterations of PASA on input streams of 100,000 elements. Overall PASA simulated a total of 131,182 possible combinations.

Table 12.4 shows that the minimum energy consumption and completion time were 10,395 joules and 148.8 seconds, respectively. While the combination (C2 C19 C27) turned out to be the best to minimize energy consumption, and (C13 C25 C39) turned out to be the best to minimize completion time, none of the combinations succeeds in minimizing minimizes both QoS attributes.

To show that the results illustrated in Table 12.4 are meaningful and combinations chosen are the best ones, we selected one of the winner combination (C13 C25 C39) and compared it with non-winner combination (C19 C2 C35). To do that, we manually calculated the
<table>
<thead>
<tr>
<th>Id</th>
<th>Workers</th>
<th>Frequency</th>
<th>Completion Time</th>
<th>Energy</th>
<th>Bandwidth</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>1</td>
<td>1200000</td>
<td>0.16024875</td>
<td>0.248465687</td>
<td>6.240298286</td>
<td>Low</td>
</tr>
<tr>
<td>C2</td>
<td>2</td>
<td>1300000</td>
<td>0.074086875</td>
<td>0.089645119</td>
<td>13.49766743</td>
<td>Low</td>
</tr>
<tr>
<td>C3</td>
<td>3</td>
<td>1400000</td>
<td>0.0459875</td>
<td>0.296095117</td>
<td>21.74503941</td>
<td>Low</td>
</tr>
<tr>
<td>C4</td>
<td>4</td>
<td>1500000</td>
<td>0.032219375</td>
<td>0.241867626</td>
<td>31.03722527</td>
<td>Low</td>
</tr>
<tr>
<td>C5</td>
<td>5</td>
<td>1600000</td>
<td>0.024209375</td>
<td>0.263853136</td>
<td>41.30631212</td>
<td>Low</td>
</tr>
<tr>
<td>C6</td>
<td>6</td>
<td>1700000</td>
<td>0.0190525</td>
<td>0.296095117</td>
<td>52.48655032</td>
<td>Low</td>
</tr>
<tr>
<td>C7</td>
<td>9</td>
<td>1200000</td>
<td>0.018093125</td>
<td>0.257235386</td>
<td>55.26961208</td>
<td>Low</td>
</tr>
<tr>
<td>C8</td>
<td>9</td>
<td>1300000</td>
<td>0.016705</td>
<td>0.220582843</td>
<td>59.86231667</td>
<td>Low</td>
</tr>
<tr>
<td>C9</td>
<td>10</td>
<td>1200000</td>
<td>0.01631125</td>
<td>0.2537166</td>
<td>61.30737988</td>
<td>Low</td>
</tr>
<tr>
<td>C10</td>
<td>9</td>
<td>1400000</td>
<td>0.015529375</td>
<td>0.250029149</td>
<td>64.39409184</td>
<td>Low</td>
</tr>
<tr>
<td>C11</td>
<td>7</td>
<td>1800000</td>
<td>0.01545375</td>
<td>0.263593068</td>
<td>64.76839831</td>
<td>Low</td>
</tr>
<tr>
<td>C12</td>
<td>9</td>
<td>1500000</td>
<td>0.01474375</td>
<td>0.257614926</td>
<td>69.08761173</td>
<td>Low</td>
</tr>
<tr>
<td>C13</td>
<td>9</td>
<td>1600000</td>
<td>0.0135725</td>
<td>0.259225832</td>
<td>73.67839831</td>
<td>Low</td>
</tr>
<tr>
<td>C14</td>
<td>9</td>
<td>1700000</td>
<td>0.012776875</td>
<td>0.257235386</td>
<td>78.26639926</td>
<td>Low</td>
</tr>
<tr>
<td>C15</td>
<td>13</td>
<td>1200000</td>
<td>0.012649375</td>
<td>0.243189294</td>
<td>79.0528929</td>
<td>Medium</td>
</tr>
<tr>
<td>C16</td>
<td>10</td>
<td>1700000</td>
<td>0.01152125</td>
<td>0.258352288</td>
<td>86.79613757</td>
<td>Medium</td>
</tr>
<tr>
<td>C17</td>
<td>11</td>
<td>1600000</td>
<td>0.0111975</td>
<td>0.23940367</td>
<td>89.30564858</td>
<td>Medium</td>
</tr>
<tr>
<td>C18</td>
<td>12</td>
<td>1500000</td>
<td>0.010990625</td>
<td>0.25825233</td>
<td>90.98663634</td>
<td>Medium</td>
</tr>
<tr>
<td>C19</td>
<td>14</td>
<td>1300000</td>
<td>0.01090125</td>
<td>0.236869991</td>
<td>91.3259947</td>
<td>Medium</td>
</tr>
<tr>
<td>C20</td>
<td>14</td>
<td>1400000</td>
<td>0.010834375</td>
<td>0.238451448</td>
<td>93.3698371</td>
<td>Medium</td>
</tr>
<tr>
<td>C21</td>
<td>15</td>
<td>1300000</td>
<td>0.008835</td>
<td>0.263578089</td>
<td>113.1861913</td>
<td>Medium</td>
</tr>
<tr>
<td>C22</td>
<td>15</td>
<td>1600000</td>
<td>0.0082675</td>
<td>0.26048425</td>
<td>120.9555488</td>
<td>Medium</td>
</tr>
<tr>
<td>C23</td>
<td>19</td>
<td>1800000</td>
<td>0.00811875</td>
<td>0.32883256</td>
<td>123.1716705</td>
<td>Medium</td>
</tr>
<tr>
<td>C24</td>
<td>15</td>
<td>1700000</td>
<td>0.00778625</td>
<td>0.24961861</td>
<td>128.4315299</td>
<td>Medium</td>
</tr>
<tr>
<td>C25</td>
<td>22</td>
<td>1700000</td>
<td>0.007419375</td>
<td>0.34734748</td>
<td>134.7824242</td>
<td>Medium</td>
</tr>
<tr>
<td>C26</td>
<td>12</td>
<td>2300000</td>
<td>0.0073675</td>
<td>0.257360773</td>
<td>135.7312521</td>
<td>Medium</td>
</tr>
<tr>
<td>C27</td>
<td>16</td>
<td>1700000</td>
<td>0.00732375</td>
<td>0.253156404</td>
<td>136.542072</td>
<td>High</td>
</tr>
<tr>
<td>C28</td>
<td>20</td>
<td>1700000</td>
<td>0.00727875</td>
<td>0.297557484</td>
<td>137.386227</td>
<td>High</td>
</tr>
<tr>
<td>C29</td>
<td>13</td>
<td>2100000</td>
<td>0.007234375</td>
<td>0.279772814</td>
<td>138.2299417</td>
<td>High</td>
</tr>
<tr>
<td>C30</td>
<td>20</td>
<td>1800000</td>
<td>0.007096875</td>
<td>0.307307462</td>
<td>140.9070894</td>
<td>High</td>
</tr>
<tr>
<td>C31</td>
<td>18</td>
<td>1600000</td>
<td>0.007080625</td>
<td>0.261873375</td>
<td>141.2304705</td>
<td>High</td>
</tr>
<tr>
<td>C32</td>
<td>21</td>
<td>1700000</td>
<td>0.007076875</td>
<td>0.2841981</td>
<td>141.3053078</td>
<td>High</td>
</tr>
<tr>
<td>C33</td>
<td>16</td>
<td>1800000</td>
<td>0.006921875</td>
<td>0.261062669</td>
<td>144.469526</td>
<td>High</td>
</tr>
<tr>
<td>C34</td>
<td>13</td>
<td>2200000</td>
<td>0.0069075</td>
<td>0.260766414</td>
<td>144.7701773</td>
<td>High</td>
</tr>
<tr>
<td>C35</td>
<td>12</td>
<td>2400000</td>
<td>0.006844375</td>
<td>0.252762769</td>
<td>146.1053785</td>
<td>High</td>
</tr>
<tr>
<td>C36</td>
<td>17</td>
<td>1900000</td>
<td>0.00625625</td>
<td>0.278315341</td>
<td>150.9921576</td>
<td>High</td>
</tr>
<tr>
<td>C37</td>
<td>14</td>
<td>2400000</td>
<td>0.00527875</td>
<td>0.27760364</td>
<td>168.696184</td>
<td>High</td>
</tr>
<tr>
<td>C38</td>
<td>20</td>
<td>2400000</td>
<td>0.0051125</td>
<td>0.336842886</td>
<td>195.599022</td>
<td>High</td>
</tr>
<tr>
<td>C39</td>
<td>22</td>
<td>2400000</td>
<td>0.004913125</td>
<td>0.318326282</td>
<td>203.5364457</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 12.3: List of configurations
completion time for both combinations taking into account the switching costs. \(CT\) represent completion time while \(SC\) represent switching cost in calculation. The calculation clearly shows that the completion time for combination \((C13\ C25\ C39)\) \(0.277 \text{ sec}\) is less than \((C19\ C2\ C35)\) \(0.471 \text{ sec}\) and follows the same patterns as estimated in Table. 12.4.

\[
(C13\ C25\ C39) = CT(C13) + SC(\text{from } C13 \text{ to } C25) + CT(C25) + SC(\text{from } C25 \text{ to } C39) + CT(C39) \\
= 0.0135725 + [(24/16) \times 0.08 \times (1+1/9)] + 0.007419375 \\
+ [(24/17) \times 0.08 \times (1+1/22)] + 0.004913125 \\
= 0.277 \text{ sec}
\]

\[
(C19\ C2\ C35) = CT(C19) + SC(\text{from } C19 \text{ to } C2) + CT(C2) + SC(\text{from } C2 \text{ to } C35) + CT(C35) \\
= 0.01090125 + [(24/13) \times 0.08 \times (1+1/14)] + 0.074086875 \\
+ [(24/13) \times 0.08 \times (1+1/2)] + 0.006844375 \\
= 0.471 \text{ sec}
\]

To better visualize Table. 12.4, selective results were plotted in Figure. 12.4. The x-axis represents id’s of different combinations while y-axis represent respective values\(^5\). The figure reinstates the findings of Table. 12.4, that none of the combinations succeeds in minimizing both QoS attributes. However, combinations with id’s range between 20 and 30 represent better trade-off between energy consumption and completion time and can be selected.

---

\(^5\)Combinations are ordered on x-axis according to completion time. The value for energy consumption has been scaled down by 10 to compare to allow suitable comparison with time.
<table>
<thead>
<tr>
<th>Id</th>
<th>Combination</th>
<th>Energy (Joule)</th>
<th>Completion Time (Second)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C1 C14 C27</td>
<td>24930.35</td>
<td>14471.78</td>
</tr>
<tr>
<td>1</td>
<td>C1 C14 C28</td>
<td>24955.09</td>
<td>14471.74</td>
</tr>
<tr>
<td>2</td>
<td>C1 C14 C29</td>
<td>24943.91</td>
<td>14471.78</td>
</tr>
<tr>
<td>3</td>
<td>C1 C14 C30</td>
<td>24959.52</td>
<td>14471.72</td>
</tr>
<tr>
<td>4</td>
<td>C1 C14 C31</td>
<td>24934.16</td>
<td>14471.77</td>
</tr>
<tr>
<td>5</td>
<td>C1 C14 C32</td>
<td>24942.04</td>
<td>14471.72</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1507</td>
<td>C13 C25 C39</td>
<td>26296.19</td>
<td>148.84</td>
</tr>
<tr>
<td>1688</td>
<td>C13 C39 C25</td>
<td>26296.19</td>
<td>148.84</td>
</tr>
<tr>
<td>3445</td>
<td>C19 C2 C27</td>
<td>10395.59</td>
<td>3361.95</td>
</tr>
<tr>
<td>3453</td>
<td>C19 C2 C35</td>
<td>10397.76</td>
<td>3361.93</td>
</tr>
<tr>
<td>3463</td>
<td>C19 C27 C2</td>
<td>10395.59</td>
<td>3361.95</td>
</tr>
<tr>
<td>3580</td>
<td>C19 C35 C2</td>
<td>10397.76</td>
<td>3361.93</td>
</tr>
<tr>
<td>3783</td>
<td>C2 C19 C27</td>
<td>10395.59</td>
<td>3361.95</td>
</tr>
<tr>
<td>3791</td>
<td>C2 C19 C35</td>
<td>10397.76</td>
<td>3361.93</td>
</tr>
<tr>
<td>3892</td>
<td>C2 C27 C19</td>
<td>10395.59</td>
<td>3361.95</td>
</tr>
<tr>
<td>3996</td>
<td>C2 C35 C19</td>
<td>10397.76</td>
<td>3361.93</td>
</tr>
<tr>
<td>5810</td>
<td>C25 C13 C39</td>
<td>26296.19</td>
<td>148.84</td>
</tr>
<tr>
<td>5997</td>
<td>C25 C39 C13</td>
<td>26296.19</td>
<td>148.84</td>
</tr>
<tr>
<td>6557</td>
<td>C27 C19 C2</td>
<td>10395.59</td>
<td>3361.95</td>
</tr>
<tr>
<td>6570</td>
<td>C27 C2 C19</td>
<td>10395.59</td>
<td>3361.95</td>
</tr>
<tr>
<td>9599</td>
<td>C35 C19 C2</td>
<td>10397.76</td>
<td>3361.93</td>
</tr>
<tr>
<td>9612</td>
<td>C35 C2 C19</td>
<td>10397.76</td>
<td>3361.93</td>
</tr>
<tr>
<td>10879</td>
<td>C39 C13 C25</td>
<td>26296.19</td>
<td>148.84</td>
</tr>
<tr>
<td>11041</td>
<td>C39 C25 C13</td>
<td>26296.19</td>
<td>148.84</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>13178</td>
<td>C9 C39 C23</td>
<td>25969.76</td>
<td>163.74</td>
</tr>
<tr>
<td>13179</td>
<td>C9 C39 C24</td>
<td>25313.66</td>
<td>165.05</td>
</tr>
<tr>
<td>13180</td>
<td>C9 C39 C25</td>
<td>26033.42</td>
<td>163.19</td>
</tr>
<tr>
<td>13181</td>
<td>C9 C39 C26</td>
<td>25397.49</td>
<td>164.67</td>
</tr>
</tbody>
</table>

Table 12.4: PASA results for Dynamic reconfiguration
The importance of estimating the QoS of parallel applications is a widely recognised issue in the parallel design pattern community.

Jay [64] and Hayashi et al. [65] were among the earliest works to predict QoS of skeleton applications. Their main motivation was to allow the developer to experiment with different but equivalent programs to assess QoS. Both of these approaches are designed for data parallel skeletons (i.e., map, fold etc). Jay [64] highlighted the need to consider application workflows. He outlined an algorithm called GoldFISH. The algorithm initially identifies the skeletons employed by the current program. With the help of differential equation solver, it generates alternative programs and a cost model to allow comparison between them. Hayashi et al. [65] proposed a QoS prediction algorithm (called VEC-BSP) based on abstract interpretation. The approach works by translating the given program into an intermediate language (called Msize). The intermediate form only stores the workflow information. The final step makes use of this information to predict QoS (execution time). Neither Jay [64] nor Hayashi et al. [65] however address non-determinism in skeleton based application.

Other early work (e.g., Gorlatch et al. [66] and Aldinucci et al. [67, 68]) done in this area focused on proposing transformation rules for different parallel design patterns to optimise the QoS of their compositions. Gorlatch et al. [66] proposed two transformation rules to cover the data parallel patterns scan and reduce. The first transformation rule converts a composition of scan and reduce to scan while the second rule converts a composition of scan and scan to scan. By applying such rules to a network topology example, Gorlatch et al. [66] show that the resulting transformation yields a better QoS with respect to
the originally specified composition of patterns. They extended their approach to *allpairs* skeleton in [69]. Aldinucci et al. [67] extend the idea of Gorlatch [66] from data parallel patterns to stream parallel patterns (*pipe* and *farm*). Namely, Aldinucci et al. [67] propose transformation rules to convert a composition of *pipes* and *farms* to an equivalent normal form, which is essentially a *farm* of *comps*. Experimental results demonstrate that the obtained normal forms provide better QoS with respect to original compositions. Aldinucci et al. [68] proposed another transformational framework that permits improving the (time) performances of parallel design patterns based application by transforming a given composition of patterns into a functionally equivalent, but more efficient composition. Emoto et al. [70] also proposed transformation rules for programs containing *shift*, *zip* and *map* skeletons to yield more efficient program.

All the aforementioned approaches assign the responsibility of optimising performances to the framework implementing the parallel design patterns, which however can optimise application performances up to a certain extent. Application developers are in a better position to optimise their code, but too-low level information cannot be conveyed to a developer, and this makes it difficult for her to optimise her code. Aldinucci et al. [71, 72], Caromel et al. [73], Kandasamy et al. [74], and Khargharia et al. [75] follow this motivation, and propose approach that are suitable for scenarios where execution environment has to be hidden to developers. Aldinucci et al. [72] permits developers to provide a contract defining the desired QoS of a parallel design patterns based application. The runtime environment continuously monitors an application, and in case of a mismatch between the QoS contract and the actual application performances, a reconfiguration of the application is planned and executed. Aldinucci et al. [71] extends [72] by introducing autonomic managers ensuring that QoS requirements are satisfied at runtime, and which can not only be attached to an application but also to the single activities building an application. Caromel et al. [73] not only detects performance degradations in parallel design patterns based applications, but also provide developers with explanations and suggestions on how to address them. Performance degradations and improvement suggestions are obtained by continuously monitoring applications and by using ad-hoc performance metrics. Kandasamy et al. [74] and Khargharia et al. [75] are other examples of hierarchical autonomic management where autonomic managers cooperate to ensure a certain QoS.

Other approaches worth mentioning are for instance Benoit et al. [76], Castro et al. [77], Danelutto et al. [78] and De Sensi [79]. Benoit et al. [76] proposed using process algebras
to predict QoS of skeleton applications. The main motivation was to analyze the performance of Grid applications with the use of algorithmic skeletons and process algebras. Their algorithm AMoGeT works by first generating PEPA models \[57\] from the given program. Solving the models and then comparing results provide performance information \((\text{throughput})\). Their approach only covered the \textit{Pipe} pattern, though. Castro et al. \[77\] proposed a QoS prediction algorithm based on denotational semantics. The skeletons they considered were \textit{sequence}, \textit{pipe} and \textit{farm}. Their algorithm works in three steps. It starts by generating all possible alternatives for a given program. Then minimum number of worker threads for each alternatives is calculated. The alternative with lowest number of cores is selected at the end. The approach by Castro et al. is deterministic and they assume that all activities will take same amount of time to execute, in their proposed cost model, which can be unrealistic in some scenario. Danelutto et al. \[78\] and De Sensi \[79\] instead focus on identifying the most appropriate configuration (in terms of CPU frequency and number of employed cores) to meet some given performance goals. Danelutto et al. \[78\] propose an approach to reconfigure stream parallel patterns based applications at runtime. Essentially, they monitor system utilisation and dynamically select a new configuration if such configuration minimises energy consumption. De Sensi \[79\] shows that testing all possible configurations to identify that yielding the optimal QoS requires a long time. To address this problem, De Sensi \[79\] proposes an approach that can execute and monitor the application on few configurations, and then performs a linear regression on the monitored data to estimate the QoS of all remaining configurations.

Summing up, Gorlatch et al. \[66\] and Aldinucci et al. \[67, 68\] focus on optimizing the QoS of parallel design patterns based application at compile time, hiding all QoS information to the developer. Aldinucci et al. \[72, 71\], Caromal et al. \[73\], Kandasamy et al. \[74\], Khargharia et al. \[75\], and Danelutto et al. \[78\] instead monitor QoS at runtime, and reconfigure the running application if given QoS requirements are violated. The only approaches focusing on the design time prediction of QoS are

- Benoit et al. \[76\] which only cover one pattern.

- Jay \[64\] and Hayashi et al. \[65\] both of which covers only data parallel patterns and have limitations in terms of non-determinism like Castro et al. \[77\].

- De Sensi \[79\], which however still requires to run some application configurations, to monitor their performances, and then to exploit the retrieved information to estimate the performances of other possible configurations.
Our approach differs from all the aforementioned approaches since it tackles the problem from a different perspective. Its novelty indeed resides in reducing whatever composition of parallel design patterns to the composition of two simple cost compositors (i.e., Both and Delay), and in exploiting Monte Carlo simulations to deal with the non-determinism introduced by input types distribution and by Feedback loops.
Part III

Conclusions
Chapter 14

Conclusions

In this thesis we presented two algorithms that are capable of probabilistically predicting the Quality of Service (QoS) of two classes of applications: Service based applications [3, 4] and Parallel design patterns based applications [5].

The algorithms are able to handle compositions containing arbitrary dependency structures, unbounded loops, fault handling, to preserve correlations among workflow activities, and to take into account different possible results of service invocations.

For service based applications, the algorithm was implemented in a proof-of-concept tool called PASO, which is able to analyse a proper subset of WS-BPEL [11]. The input to PASO are a WS-BPEL workflow, and probability distributions for the QoS properties of the invoked services as well as for the evaluation of the workflow branching conditions. The output of the algorithm is a probability distribution for the QoS properties (time, reliability and cost) of the orchestration.

For parallel design patterns based applications, the algorithm was implemented in a proof-of-concept tool called PASA. The input to PASA are a parallel design pattern based application, and probability distributions for the QoS properties of the nodes, for the types of inputs in the input stream as well as for the evaluation of the feedback conditions. The output of the algorithm is a probability distribution for the QoS properties (completion time and energy) of the application.

The prediction technique prototyped in PASO and PASA advance the state of the art for the following reasons:
• They successfully handle all the four challenges we described in Section 1.2: They deal with different results of service invocations. A composition invoking external services behave differently if the service returns a successful reply, a fault notification, or no reply at all. The resulting QoS of the composition differs in each case. PASO and PASA model such behaviour.

They deal with application nondeterminism. Control flow structures (alternatives and iterations) depend on input data which may differ in different runs. This may lead, for instance, to different numbers of iterations or to different branches executed in alternatives. Moreover, certain QoS properties of invoked services can vary from one run to another (e.g., response time). PASO and PASA model such behaviour.

They deal with complex dependencies among activities. Workflows with complex dependencies (e.g., synchronization links) cannot be always decomposed into parallel and sequential compositions. PASO and PASA model such behaviour.

Last, but not least, they deal with correlations among workflow activities.

• PASO and PASA can provide results both in terms of average values of QoS properties and as distributions in the form of histograms, which provide valuable information to users, as we illustrated in Section 6.1. Indeed, designers are not interested only in getting estimated average values for the QoS but they are usually interested also in the distribution of values for such answers (e.g., “What is the probability that the response time of this orchestration will be more than 2 seconds?”).

• PASO and PASA can provide estimations efficiently, and with few computing resources, if compared to simulating (when possible) actual deployments. For example PASO takes few seconds to generate results for 100,000 Monte Carlo simulations.

• PASO and PASA can be easily extended to analyse other QoS attributes by simply defining the way in which attributes are composed by **Both** and **Delay**.

• PASO and PASA can be also configured by tuning the accuracy of the analyses by modifying the number of Monte Carlo simulations to be performed.

PASO and PASA can support application designers in quickly predicting the effects of modifying parts of an application. For instance, a designer may wish to assess which could be the effects of replacing a service with a newly offered alternative, as discussed in [17], or which could be the effects of refactoring an application. In such perspective, PASO and PASA can hence support designers in improving cost, time or energy consumption of their
applications.

The approach that we presented can be extended in various ways:

- **PASO** and **PASA** can be extended to support new activities and languages. **PASO** is currently capable of analyzing a proper subset WS-BPEL. Support for other interesting WS-BPEL activities (e.g., Pick, Event Handlers) and workflow languages (e.g., YAWL [80], BPMN [81]) could be included in **PASO**. Similarly, **PASA** currently models a simple set of parallel design patterns. Support for other interesting data parallel patterns (e.g., map, reduce) and frameworks (e.g., FastFlow [62]) could be included in **PASA**.

- The naive Monte Carlo implementation employed in **PASO** and **PASA** can be improved in various ways (e.g., [82, 83, 84, 85]).

- Some forms of correlations could be introduced in the samplings. For instance, it would be interesting to consider some correlation among service invocations (e.g., if a service invocation returns a fault because it is “down for maintenance” it may be probable that the same result will be obtained in the next invocation) and in the input stream of parallel applications (e.g., bursts of data).

- A possible extension could be to calculate, by using program profiling techniques, bounds for variable values and number of loop iterations in workflows. For instance, the techniques proposed in [86, 87, 88, 89, 90, 91, 92, 93] can help estimating worst-case execution times (WCET) and worst-case execution paths (WCEP), thus reducing the amount of info needed for static QoS prediction.

- Last, but not least, the definition of the cost compositors **Both** and **Delay** can be extended to support new QoS properties (e.g., throughput, availability, and so on). An interesting extension in this perspective would be to provide users with a query language to specify their own QoS properties.
Bibliography


[18] Leonardo Bartoloni, Antonio Brogi, Ahmad Ibrahim: Predicting the QoS of service orchestrations. Submitted for publication (2016)


14.0. BIBLIOGRAPHY


