COST MANAGEMENT OF SERVICE COMPOSITIONS

Robson W. A. de Medeiros¹,², Nelson S. Rosa³, Luís Ferreira Pires³
¹Federal University of Pernambuco, Centre of Informatics (CIn), Recife, Brazil
²Rural Federal University of Pernambuco, Department of Statistics and Informatics (DEINFO), Recife, Brazil
³Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente, Enschede, The Netherlands

{rwam, nsn}@cin.ufpe.br
l.ferreirapires@utwente.nl

Abstract
Nowadays, several organisations provide services on the Internet with similar functionality but different price and pricing schemes. Since these services are used in service compositions, the providers of composed services have to face the challenge of managing the services they use, in order to decrease the cost of the service compositions without compromising their quality. Several techniques to compute the cost of service compositions are currently available, but none of them take into account all classes of cost behaviour. This paper proposes an approach to service composition that consists of algorithms and an architecture to support the cost management of service compositions by considering several classes of cost behaviour. We developed a prototype in order to evaluate our approach with experiments that show that it properly supports cost management of service compositions throughout the service composition life-cycle. Our approach is quite promising as it is expected to dramatically improve the cost management of service compositions.

Keywords: Service-Oriented Architecture, Service Composition, Services Management, Cost Management, Cost Estimation.

1. INTRODUCTION
Service-oriented computing (SOC) is a paradigm that allows organisations to expose their competencies as services in order to facilitate the cooperation with their partners. This paradigm makes it possible to aggregate multiple component services in order to create more complex ones, which are called service compositions (Michael P Papazoglou, Traverso, Dustdar, & Leymann, 2007). SOC created a new marketplace where services are provided on the Internet that offer similar functionality but with different quality of service (QoS). Service compositions can be specified in an abstract way in terms of combinations of logically related abstract services, which have to be replaced by concrete services when they are executed. For the sake of brevity, we use the term service to denote a concrete service in the remainder of this paper.

Organisations that provide computational services need to properly manage their finance in order to stay profitable and survive in the marketplace. This means that service providers should keep their income higher than their costs, which is determined by the expenses required to create, advertise and provide services or products (Bragg, 2014). To manage the cost of service compositions, one needs to take into account service composition requirements and services available in the marketplace. In a service composition, an abstract service can be implemented by a service that belongs to a set of candidate services. Moreover, a (concrete) service can in principle be used to implement more than one abstract service, under the condition that the service is able to support the functionality required by the abstract services.

In the service marketplace, services may have similar functionality but different qualities, such as reliability and price. The reliability and price of the services are two quality attributes that directly affect the cost of a service composition. Concerning reliability, a service failure may demand more resources for the execution of some corrective behaviour, and may lead to a cost increase. Price is the amount of money paid by a customer in order to use the service, and cost is often defined as the entire expenditure required to create and sell products and services (Bragg, 2014). Therefore, from the point of view of the service composition, the price charged by the provider of a component service is the cost to use the service, while the pricing model determines the cost behaviour of this service, i.e., how the cost of the service changes as a function of its usage.

Cost management is a process that helps business experts to plan, control and take decisions about the operational and financial situation of organisations (Hansen & Mowen, 2012). Cost management in service composition has some open challenges as identified in (Albuquerque de Medeiros, Rosa, Campos, & Ferreira Pires, 2014a), mainly when complex cost behaviours are taken into account to predict, simulate, control and report cost of service compositions. Cost behaviours can be classified as fixed, variable, mixed and step cost (Hansen, Hansen, Mowen, Mowen, & Madison, 2010; T Horngren, Datar, & Rajan, 2011). Even when they belong to the same cost behaviour class, costs may be computed differently. For instance, two services can have variable cost behaviours, but one can be charged based on the number of invocations while the other
can be charged according to the message size. Moreover, in order to stimulate service consumption and remain competitive, providers can offer discounts that vary according to the volume of use of these services.

Cost management has been extensively discussed in service-oriented computing (Albuquerque de Medeiros, Rosa, Campos, & Ferreira Pires, 2014a) and generally in industry (Siguenza Guzman, Van den Abbeele, Vandewalle, Verhaaren, & Cattrysse, 2013). Outside service-oriented computing, the resources used to provide services or products are usually internal resources controlled by the provider. In this case, the cost behaviour of each resource (person, machine, and so on) is defined internally by the provider of the service or product. Therefore, cost behaviours of similar resources tend to be similar. In contrast, in SOC most of the resources (services) are provided by third-parties and, in the worst case, each service can have a different cost behaviour. Although cost management in different phases of the service composition life-cycle has been extensively discussed before (Albuquerque de Medeiros, Rosa, Campos, & Ferreira Pires, 2014a), to the best our knowledge different classes of cost behaviours have not been properly addressed yet in the literature.

This paper extends our previous work (Albuquerque de Medeiros, Rosa, & Ferreira Pires, 2015), which proposed an analytical approach to predict the cost of service compositions taking into account all aforementioned classes of cost behaviour. This paper proposes an approach to manage the cost of service compositions throughout the service composition life-cycle. Our approach prescribes that service cost behaviours have to be defined by the service providers, and exploits the opportunity of using a service to implement more than one abstract service. This paper also introduces an architecture that supports service composition management with special attention to cost management.

In order to demonstrate and evaluate our solution, we implemented the business process of the service-oriented pizza delivery system introduced in (Baresi, Ghezzi, & Guinea, 2004), along with the component services of this service composition. Based on this example, we compared the costs computed by our approach with the costs computed separately by each service and by using approaches found in the literature (Magnani & Montesi, 2007; Saedidi, Zhao, & Sampaio, 2010).

This paper is further organised as follows: Section 2 introduces the conceptual background on cost and cost management, Section 3 describes our approach to manage the cost of service compositions, Section 4 evaluates our approach, Section 5 discusses related work and Section 6 gives our conclusions and some directions for future work.

2. Basic Concepts

This section introduces the concepts of cost and reliability used throughout the paper.

2.1 Cost

Cost is the entire expenditure required to create and sell products and services (Bragg, 2014). It can be classified according to its behaviour, which is usually referred to as fixed, variable, mixed and step cost (T Horngren et al., 2011).

Fixed cost is a behaviour that remains constant independently of the use of the service within a period of time, such as day, month or year, as show in Figure 1(a). In contrast, variable cost is only computed if the service is used and its value is directly proportional to the level of use of the service, as show in Figure 1(b). Factors that cause changes in the cost behaviour are known as cost drivers. A cost driver is a variable of an activity that changes the cost of the service when the activity is executed, such as number of invocations and message size.

Mixed cost behaviours combine variable and fixed costs. Mixed cost behaviour starts with a fixed value that increases proportionally to the use of the service, as shown in Figure 1(c).

Finally, step cost is a class of cost behaviour that can be divided into three patterns:

- The cost of an unit of a cost driver starts with a particular value, and changes when the volume of the cost driver reaches some threshold in a period of time, such as, e.g., the periods of time delimited by time moments \((x,y,z)\) in Figure 1(d);
- A specific cost is defined for a volume unit of a cost driver between the thresholds for a period of time, as shown in Figure 1(e); and
• In a period of time, the cost is fixed within ranges of use of the service, and after reaching a threshold the fixed cost is modified to a new value, as depicted in Figure 1(f).

Cost management is a process that involves the planning, control, and decision making about the operational and financial situation of organisations (Hansen & Mowen, 2012). Cost management should span the whole service or product production life-cycle, by recording, measuring, estimating and analysing costs. A Cost management system is a tool used to collect, categorise, summarise, and analyse costs to produce helpful information to managers (Horngren, Foster, Datar, & Rajan, 2010). This system must provide relevant information that can be used to:
• Measure the effect of decisions concerning the service;
• Ensure that activities that transform computational resources into services are performed as efficiently as possible;
• Plan the service budget for a certain period of time;
• Identify problems that can make the financial aspects of the service deviate from the planned situation.

To manage cost in service-oriented composition, we need to consider the phases of service composition life-cycle. Many service composition life-cycles have been proposed in the literature (Baryannis, Carro, Danylevych, & Dustdar, 2008; Benatallah, Dumas, Fauvet, Rabhi, & Sheng, 2002; Pessoa et al., 2008) and in most of them these life-cycles have phases such as planning, verification, discovery, deployment and execution. We based our work on the service composition life-cycle proposed by (Yang & Papazoglou, 2004), which consists of four phases, namely Planning, Definition, Scheduling and Construction, and Execution, as shown in Figure 2.

In the Planning phase, all business interactions, activity sequences and non-functional requirements (e.g., cost) are planned, analysed and described by business experts. In the Definition phase, a service composition and its constraints are defined by using standard techniques such as WSDL (W3C, n.d.) and WS-BPEL (OASIS, n.d.) according to the planned service description. In the Scheduling phase, one determines how and when the service is supposed to run. In this phase, alternative schedules can be generated and proposed according to the planning, so that they can be chosen before the service execution. In the Construction phase, the service composition is made ready for execution. Finally, in the Execution phase the service is executed and can be monitored and controlled according to the needs of the service provider in order to manage its qualities, e.g., to maintain the cost of the service composition below some threshold.

2.2 Reliability

Unreliable services can harm the entire cost of a service composition (Saeedi et al., 2010; Sampathkumaran, 2013), since if a service fails, all services invoked previously have already been accounted to calculate the actual cost of the composition.

Reliability is often defined as the probability of a system to perform its intended function free of failures within a specified period of time (M. Xie, Poh, & Dai, 2004). The reliability of a service composition can be computed by recursively applying rules on the workflow patterns, such as the sequential, parallel, loop and conditional branching patterns (Cardoso, Sheth, Miller, Arnold, & Kochut, 2004; C. Xie & Ren, 2014). The use of these patterns facilitates the calculation of the reliability of the entire service composition. For example, Figure 3 shows n services ordered sequentially in a process, each one with reliability R(si). In this case, the reliability of the entire service composition, denoted as sr, can be computed by multiplying the reliability of all services, since failure probabilities are cumulative, as shown in Equation (1).

\[
R(s_r) = \prod_{i=1}^{n} R(s_i)
\]  

(1)

Figure 2. Service composition life-cycle

Figure 3. Sequential workflow reduction
When two or more services are executed in parallel, as shown in Figure 4, the sequence of execution cannot be guaranteed at design time. However, like the sequential pattern, all services must be executed correctly in each parallel branch to enable the service composition to continue executing. Therefore, the reliability of this workflow pattern can also be computed by multiplying the individual probabilities of the $n$ parallel services by applying Equation (1).

![Figure 4. Parallel workflow reduction](image)

In the conditional branching pattern, in contrast, only one path is executed in a workflow execution. To analyse this pattern, we consider the probability of each path being executed. For instance, in Figure 5, the probability of each service be executed in the workflow is $w_i$, whilst the sum of probabilities must be 100%, since each service has a different probability of be executed and only one service is actually executed.

![Figure 5. Conditional workflow reduction](image)

Therefore, the reliability of the entire service composition $s_r$ can be computed as in Equation (2).

$$R(s_r) = \sum_{i=1}^n w_i R(s_i)$$  \hspace{1cm} (2)

When a loop pattern has one or more services, these services can be invoked multiple times during a certain process execution, depending on the condition defined in the service composition (see Figure 6).

![Figure 6. Loop workflow reduction](image)

To analyse this pattern, we consider either the number of iterations $n$, or the probability that the process moves either forward $w_f$ or backward $w_b$ to invoke the services in the loop again. Equation (3) shows that the reliability of a service $s_i$ can be computed by taking into account the number of iterations $n$. This equation can be applied when the average number of iterations is known. Otherwise, Equation (4) can be applied to compute the reliability taking into account the probability $w_i$ of a service $s_i$ getting into the loop.

$$R(s_r) = R(s_i)^n$$  \hspace{1cm} (3)

$$R(s_r) = \frac{(1-w_f)R(s_i)}{1-w_fR(s_i)}$$  \hspace{1cm} (4)

3. COST MANAGEMENT APPROACH

In order to develop our cost management approach, we initially identified the following key requirements:

- **R1**: To support all classes of cost behaviours;
- **R2**: To compute and predict service costs with little human intervention;
- **R3**: To allow runtime adaption of service compositions based on cost requirements; and
- **R4**: To allow the integration of the proposed solution with existing service composition management systems.

Our approach has been developed to fulfil these requirements, and consists of a generic architecture for cost management systems and a set of algorithms to predict and compute cost of service compositions. We also defined a metamodel that allows cost experts to express different cost behaviours and enforce them in our architecture. Figure 7 shows that the architecture comprises seven main components, namely Modeller, Knowledge Repositories, Execution Engine, Service and Cost Monitors, Adapter, Cost Analyser and Listeners.

The Modeller allows service designers to model service compositions annotated with cost behaviour and reliability information. The cost behaviours are modelled as instances of our metamodel, and are stored in the Knowledge Repositories, which ensure that all components of the architecture can access the cost behaviours used in the architecture (R1).

After modelling the service composition and cost behaviours, one can analyse the composition cost and deploy the service composition in an Execution Engine. The Cost Analyser analyses the service composition cost in two alternative ways: using the abstract service composition annotated with expected cost behaviours and reliability information; or using the event log of the service composition simulation. Since we use the cost behaviour of the services instead of their average cost, we can analyse
cost of the service composition in different scenarios without changing the cost of each individual service of the composition (R2). For example, one can analyse the service composition cost by changing its processing demand, changing candidate services, or even changing the behaviour of the service composition.

![Generic Architecture of Cost Management Systems](image)

**Figure 7. Generic Architecture of Cost Management Systems**

After the service composition is deployed, it is executed by the Execution Engine and monitored by the Service and Cost Monitors. The Execution Engine can be an existing service composition engine or a business process engine that supports Web service invocations, such as Apache ODE\(^1\) or Activiti\(^2\), which are monitored by Listeners provided by our solution (R4). The monitors record the information used by the Cost Analyser to compute the cost of both the service composition and the component services. Since an abstract service can have more than one candidate service, the Service and Cost Monitors ask the Cost Analyser to evaluate the cost of each candidate service, and then the Adapter acts to reconfigure the service composition to use the service with the lowest cost (R3).

In order to present our approach more rigorously, we formalised some of these concepts as follows:

**Definition 1 (Annotation)** An annotation is a tuple \(a = \langle id, v \rangle\), where
- \(id\) is the annotation identifier, and
- \(v\) is the annotation value.

**Definition 2 (Abstract Service)** An abstract service is denoted as a tuple \(as = \langle id, A \rangle\), where
- \(id\) is the abstract service identifier, and
- \(A\) is a set of annotations.

**Definition 3 (Abstract Service Composition)** An abstract service composition is denoted as a set \(ASC \subseteq AS\), where
- \(AS = \{as_1, ..., as_n\}\) is a set of abstract services.

**Definition 4 (Service)** A service is denoted as a tuple \(s = \langle id, NFA \rangle\), where
- \(id\) is the service identifier, and
- \(NFA\) is a set of non-functional attributes associated to the service, e.g., \(NFA = \{\text{cost, reliability, performance}\}\).

**Definition 5 (Candidate Services)** Candidate services are denoted as \(CS \subseteq S\), where
- \(S = \{s_1, ..., s_n\}\) is a set of services with similar functionality that can be used to implement an abstract service \(as_i\).

### 3.1 Modeller

The Modeller is used by the cost/service experts to model cost behaviours and service compositions with cost information. Additionally, it allows cost/service experts to interact with both the Cost Analyser and the Execution Engine to analyse costs and deploy service compositions annotated with cost information, respectively.

Each service used in the composition has a cost defined by using our cost behaviour metamodel (Albuquerque de Medeiros, Rosa, Rosa, & Ferreira Pires, 2014b) in terms of some basic concepts: Cost Behaviour, Cost Driver, Cost Function, and Rules. The Cost Behaviour defines the cost behaviour of a service, i.e., fixed, variable, mixed and step cost as shown in Figure 1. The Cost Driver is the variable (typically level or volume of activity) that has direct effects on costs, for instance, message size and number of invocations. A Cost Driver is directly associated to a business attribute and a unit. The business attribute represents some process attribute (e.g., number of bytes transferred in the service invocation) and a unit informs how to compute the variation of the cost driver according to its value. The Cost Function defines how to compute the service cost. The Rule is informally defined as the rule that should be applied to choose the appropriate cost function.

These elements are formally defined as follows:

**Definition 6 (Cost Driver)** A cost driver is denoted as a tuple \(CD = \langle v, e, u \rangle\), where
- \(v\) is a business attribute used to compute the cost driver,
- \(e\) is the estimated consumptions of the cost driver in a period of time, and
- \(u\) is a unit that informs how to compute the variation of cost driver according to its value.

**Definition 7 (Cost Function)** A cost function is a tuple \(C = \langle p, cd, a, va \rangle\), where
- \(p\) is the cost period,
• $cd$ is the cost driver,
• $a$ is the amount of cost driver, and
• $va$ is the value for this amount ($a$) of cost driver.

**Definition 8 (Rule)** Rule is a tuple $R = <C, Ch>$, where
• $C$ is a set of cost functions, and
• $Ch$ defines the set of cost functions considered in the computation of the total cost only if a logical condition holds.

**Definition 9 (Cost Behaviour)** Cost behaviour is a tuple $CB = <V, CD, R>$, where
• $V$ is the set of attributes used to compute the service cost, e.g., $V = \{message\ size, \ activity\ identification\}$,
• $CD$ is a set of cost drivers, and
• $R$ is a set of rules.

By using the cost behaviour metamodel, Listing 1 shows a cost behaviour expressed in XML format in which instances of our metamodel are serialized. This cost varies according to a number of invocations. Therefore, its cost behaviour is based on the “numberOfInvocation” cost driver, defined in Lines 8 and 9. Moreover, the cost function is defined as $0.05$ per each invocation, where $cd = numberOfInvocation$, $a = 1$ and $va = 0.05$.

Listing 1. Variable Cost Behaviour Example

3.2 Cost Analyser

The Cost Analyser is responsible for computing the cost of both individual services and the entire composition. In order to analyse the expected cost of an abstract service composition, experts can compute the expected costs of the abstract services. The expected cost behaviour of an abstract service can be defined according to the cost behaviour of any of its candidate services. Moreover, it is also possible to analyse different expected cost behaviours for each abstract service in order to find out which service best satisfies the needs of the provider.

In this paper, we describe three strategies to compute the cost of service compositions: from models of the service compositions with reliability and cost annotations; from event logs obtained either from simulation or from real execution; or from models of service composition behaviours with candidate service annotations and event logs obtained either from simulation or real execution.

3.2.1 Computing Cost using Models

Figure 8 shows the steps to compute the cost of a service composition modelled in Business Process Model and Notation (BPMN) (Object Management Group, 2014). Firstly, it is necessary to model the abstract service composition (Model Service Composition) and expected cost behaviour of each individual abstract service (Model Cost Behaviours). Secondly, the abstract service composition is annotated with service reliability parameters (Annotate Reliability), probability of each alternative path (Annotate Conditional Statement Probabilities), and expected cost behaviours of each abstract service (Annotate Cost Behaviour Identification). Thirdly, the cost drivers of all expected cost behaviours annotated in the abstract service composition are computed (Compute Cost Driver Values). Finally, the cost of the entire abstract service composition is computed (Compute Cost of Service Composition).

We adopted BPMN to perform step Model Service Composition because BPMN has become a de facto standard for modelling business processes, and an abstract
service composition can be actually seen as a business process where each task of the business process represents an abstract service. In step Model Cost Behaviours, the cost behaviour of each individual abstract service is modelled by using the cost behaviour metamodel proposed in (Albuquerque de Medeiros, Rosa, Rosa, & Ferreira Pires, 2014b).

Step Compute Cost Driver Values is performed by using Algorithm 1. This algorithm takes into account the probability that the service executes and terminates correctly, which depends on its reliability and the probability associated to the path that includes the invocation of this service. The input of Algorithm (1) is the Abstract Service composition (ASC) and the average number of service composition executions for the period being considered. Since there is no guarantee that a service is invoked when the composition is executed, the algorithm visits all abstract service as₂ ∈ ASC to compute the average of each cost driver value for each execution of the service composition (Lines 3 - 10). For each as₂ in the ASC, the cost behaviour (cb₂) and the annotated cost drivers (ACD₂) are obtained. Next, the cost driver average execution value is computed (Line 7) taking into account the probability that the service terminates correctly during the execution of the service composition (PCE), the value of the cost driver (acd₂) annotated in the business process, and the average number of executions (n) in the period, e.g., 10 executions per day.

Given an abstract service as₂ with expected reliability R(as₂), as₂ an abstract service executed before as₂ with expected reliability R(as₂), and wj the probability that as₂ is executed after as₂, the PCE (Probability of Correct Execution) of abstract service as₂ is computed according to Equation (5).

\[
PCE(as₂) = PCE(as₁) \times w₁ \times R(as₁),
\]

where \( PCE(as₁) \) is 1.0 if as₁ is the first abstract service. Since the same service can be used to perform one or more abstract services, the average execution value of the cost driver is represented as a tuple <key, value>, where key is defined in terms of the cost behaviour (cb₂) and cost driver identifications (cd₂), and value is computed by multiplying the value of the cost driver by the PCE of the respective abstract service. This tuple is stored in a set of cost driver values (CDV) whose key must be unique. Therefore, when a new tuple needs to be stored and its key already exists in CDV, its value is added to the existing one. Otherwise, a new tuple is added to CDV.

Next, step Compute Cost of Service Composition is performed using Algorithm 2. This algorithm uses the average of the execution values of the cost drivers (CDV), and the period of analysis (periodOfAnalysis) to compute the cost of the service composition. CDV has no duplicate keys and this fact ensures that if one service is adopted to perform more than one abstract service, it is considered only once, rather than multiple times. Additionally, CDV contains the sum of all values of the cost drivers spent to invoke the service that will replace the abstract service, which ensures the correct cost calculation. To compute the cost of each service, Algorithm 2 calls Algorithm 3 (Line 5) and puts the cost of each service in a set of service cost values ServiceCost, where each element is represented as a tuple <as₂, cost>. Finally, with the expected cost of all services, the expected cost of the service composition is calculated by summing up all costs in ServiceCost.

In Algorithm 3, each cost behaviour (cb₂) has a set of cost functions (cᵢ) that define how to compute the service cost. Moreover, cᵢ can be related to a cost driver. In this case, the value of the cost driver is calculated as shown in Lines 5-7. The algorithm first looks for the cost driver value in CDV. If the cost driver is not found, it takes the default value of this cost driver annotated in the cost behaviour definition. In addition, conditions can be used to define the proper function to be applied. In this case, the condition has to be evaluated before using the cost function as shown in Line 10. When the cost function can be used, the algorithm verifies if there is a cost driver associated to the cost function (Line 11). In case cᵢ has a cost driver, the cost function has a variable behaviour (Line 12); otherwise, it has a fixed behaviour (Line 14). In the case of variable behaviour, the cost is computed as shown in Equation (6). In the case of fixed behaviour, the cost function only depends on the period of analysis (periodRate). Moreover, it has the same value (va), which is independent from the use of the service, as shown in Equation (7).

\[
VariableCost = \left(\frac{costDriverValue}{a}\right) \times va
\]

\[
FixedCost = va \times periodRate
\]
3.2.2 Computing Cost using Logs

Another approach to compute the cost of service compositions is by using event logs (LOG), which is formally defined as follows:

**Definition 10 (Cost Attribute)** The cost attribute is a tuple \( ca = \langle name, value \rangle \), where
- \( name \) is the name of the cost attribute, and
- \( value \) is the value of the cost attribute.

**Definition 11 (Log)** Log is a tuple \( LOG = \langle as, s, CA \rangle \), where
- \( as \) is the abstract service,
- \( s \) is the service that implements the abstract service, and
- \( CA \) is the set of cost attributes required to compute the cost of all candidate services, where \( CA = \{ ca_1, ..., ca_n \} \).

Using LOG, the cost driver values (CDV) are computed by Algorithm 4. Next, Algorithm 2 receives the cost driver values CDV and computes the cost of the entire service composition. The input of Algorithm 4 is LOG. Using LOG, the algorithm computes the cost driver values cdv\(_{jw}\) of all cost drivers cd\(_{jw}\) by getting the attribute (cda\(_{jw}\)) in reg\(_j\) that has the same name of the business attribute associated to cd\(_{jw}\). Then, the value of the cost driver (cd\(_{jw}\)) is added to CDV according to the following rules:
- If the unit of cd\(_{jw}\) is Quantity, the added value is 1,
- If the unit of cd\(_{jw}\) is NOT Quantity, the added value is cda\(_{jw}\).

**Algorithm 2 Compute cost of service composition**

1. procedure COMPUTESERVICECOMPOSITIONCOST(CDV, periodOfAnalysis)
2. \( CB := \) get the cost behaviors in CDV
3. \( costValue := 0.0 \)
4. for all \( cb_k \) in \( CB \) do
5. \( costValue := computeServiceCost(cb_k, CDV, periodOfAnalysis) \) \( \triangleright \) Call the Algorithm 3
6. ServiceCost.add(cb\(_k\), service, costValue)
7. end for
8. return ServiceCost
9. end procedure

**Algorithm 3 Compute cost of service**

1. procedure COMPUTESERVICECOST(cb\(_k\), CDV, periodOfAnalysis)
2. \( \quad \) for all \( c_i \) in \( cb_k \) do
3. \( \quad \) periodRate := \( periodOfAnalysis \)
4. \( \quad \) if (\( cd_i \) of \( c_i \) is NOT NULL) then
5. \( \quad \quad \) costDriver := CDV.getValue(cb\(_k\), cd\(_i\).name) \times periodRate
6. \( \quad \quad \) if (costDriver is NULL) then
7. \( \quad \quad \quad \) costDriver := \( c \) of \( cd_i \) \times periodRate
8. \( \quad \quad \) end if
9. \( \quad \) end if
10. if ((\( c_i \) is in a choose element) AND (condition is true)) OR (\( c_i \) is NOT in a choose element) then
11. \( \quad \quad \) if \( cd_i \) is NOT NULL then \( \triangleright \) Variable cost
12. \( \quad \quad \quad \) costValue := costValue + (costDriver \( \triangleq \) (\( a \) of \( c_i \) \) \( \triangleq \) (\( va \) of \( c_i \) \))
13. \( \quad \quad \) else \( \triangleright \) Fixed cost
14. \( \quad \quad \quad \) costValue := costValue + (\( va \) of \( c_i \) \) \times periodRate
15. \( \quad \) end if
16. \( \quad \) end if
17. \( \) end for
18. \( \) return costValue
19. end procedure

**Algorithm 4 Compute cost driver using LOG**

1. procedure COMPUTECOSTDRIVERVALUEPERLOG(LOG)
2. \( CDV := \) empty \( \triangleright \) A set of cost driver values per cost behavior
3. \( \quad \) for all \( reg_j \) in \( LOG \) do
4. \( \quad \quad \) cb\(_j\) := get cost behavior of \( s \) in \( reg_j \)
5. \( \quad \quad \) CD\(_j\) := get cost drivers of cb\(_j\)
6. \( \quad \) for all cd\(_{jw}\) in \( CD\(_j\) \) do
7. \( \quad \quad \) cda\(_{jw}\) := \( reg_j \).getAttributes(cd\(_{jw}\).variable)
8. \( \quad \quad \) if \( u \) of \( cd_{jw} \) is Quantity then
9. \( \quad \quad \quad \) CDV.add(cb\(_j\), cd\(_{jw}\).name, 1)
10. \( \quad \quad \) else
11. \( \quad \quad \quad \) CDV.add(cb\(_j\), cd\(_{jw}\).name, cda\(_{jw}\))
12. \( \quad \quad \) end if
13. \( \quad \) end for
14. \( \) end for
15. \( \) Return CDV
16. \( \) end procedure
Recall that $CDV$ is a set of cost driver values in which each element is represented as a tuple $<key, value>$, where $key$ must be unique and is defined in terms of the cost behaviour ($cb$) and cost driver identifications ($cd$). Therefore, when a new tuple needs to be stored and its $key$ already exists, its $value$ is related to the existing key. Otherwise, a new tuple is added to $CDV$.

### 3.2.3 Compute Cost using Models and Logs

In this approach, we use the annotated service composition and the event log. Similarly to the previous approaches, to compute the cost of all candidate services ($S_i$) of an abstract service ($as_i$) we first compute their cost driver values ($CDV$). The $CDV$ of the $S_i$ is computed by $Algorithm 5$, which gets as input $as_i$, $ASC$ and $LOG$. Since all candidate services are annotated in $ASC$, in Line 3 the algorithm gets candidate services $S_i$ for each abstract service $as_i$. Then, each event $reg_j \in LOG$ is visited in order to get all attributes associated to all cost drivers of these services. After computing $CDV$ for $S_i$, the cost of each service in $S_i$ is computed using $Algorithm 2$, which returns the cost of all services in $CDV$. Therefore, the Service Selector can now select the service with the lowest cost in $ServiceCost$ to replace abstract service $as_i$.

#### Algorithm 5 Compute cost driver of Candidate Services

1: procedure COMPUTECOSTDRIVERVALUEPERLOG($as_i$, $ASC$, $LOG$)
2: $CDV := empty$ \> A set of cost driver values per cost behavior
3: $S_i :=$ get the candidate services of $as_i \in ASC$
4: for all $reg_j \in LOG$ do
5: for all $s_j \in S_i$ do
6: $cb_j :=$ get cost behavior of $s_j$
7: $CD_j :=$ get cost drivers of $cb_j$
8: for all $cd_{j,u}$ in $CD_j$ do
9: $cd_{j,u} :=$ getAttribute($cd_{j,u}$,variable);
10: if $u$ of $cd_{j,u}$ is Quantity then
11: $CDV.add(cb_j, cd_{j,u},$name$, 1)$
12: else
13: $CDV.add(cb_j, cd_{j,u},$name$, cd_{j,u})$
14: end if
15: end for
16: end for
17: Return $CDV$
18: end procedure

### 3.3 Service Selector

Before executing a service composition, each abstract service must be replaced by a service in the Scheduling phase (see $Figure 2$). The cost management system is responsible for selecting the set of services with the lowest cost. In our proposed architecture shown in $Figure 7$, the services are selected by the Service Selector module. This selection can be performed during service composition deployment or at runtime. In both cases, the costs of the candidate services are computed by $Cost Analyser$ for a period of time (e.g., one month), the services are ranked according their costs and one service is then selected according to the ranking.

### 3.4 Execution Engine

In the Execution phase (see $Figure 2$), the Execution Engine invokes the services with lowest costs provided by the Service Selector module. Monitoring and adaptation of the service composition can be performed at runtime. Furthermore, while the service composition is being executed, candidate services can be rescheduled to ensure the lowest cost of the service composition.

Additionally, at runtime the estimated consumption of the cost drivers of the cost behaviour must be updated in order to ensure that the cost of services can be calculated more precisely.

In order to reuse existent execution engines, this module is monitored by $Listeners$ provided by our solution, which are responsible for following the service execution and sending the data to the Service and Cost Monitors module.

### 3.5 Service and Cost Monitors

While the service composition is being executed by the Execution Engine, the Service and Cost Monitors monitor the execution and the cost of the service composition and its individual services. When monitoring, this component logs all necessary information to compute the cost of the services as specified by the cost/service experts in the planning phase (see $Figure 2$). Additionally, this component is responsible for asking the Cost Analyser to compute the cost of the candidate services of each abstract service, for instance, to identify which one has the lowest cost for each abstract service.

### 3.6 Adapter

The Adapter component acts on both Execution Engine and Service Selector. Concerning the Execution Engine, the Adapter can act on the entire execution of the service composition in order to manage the cost. For instance, if the cost of the composition gets higher than a certain threshold, this component can ask the Execution Engine to stop the entire service composition in order to avoid that the cost increases even more. Concerning the Service Selector, the Adapter can ask the Service Selector to re-order the list of candidate services according the cost of each service. Additionally, this component can send information to the Modeller in order to inform the cost/service experts about some possible unexpected problem with the cost of the service composition.
3.7 Knowledge Repositories

The Knowledge Repositories are responsible for storing all data produced and used in the architecture. With this component, our architecture can manage the cost of a service composition throughout its life-cycle. Besides other registries, these repositories store cost behaviours descriptions, service composition execution logs, and service composition definitions. To register services, we use a Java implementation of the Universal Description, Discovery, and Integration (UDDI) standard (OASIS, n.d.).

4. EXPERIMENTAL EVALUATION

The goal of this evaluation is to compare the costs of service compositions calculated using our approach with the costs calculated manually or using cost average to compute costs. In order to execute the experimental evaluation, we worked with an implementation of the architecture shown in Figure 7. The architecture was implemented using Activiti (Rademakers, 2012) in the Modeller and Execution Engine and jUDDI as the implementation of UDDI specification (“Apache jUDDI,” n.d.). Moreover, the service-oriented pizza delivery application (Baresi et al., 2004) was fully implemented, where each service was also annotated with cost behaviour.

4.1 Application

The pizza delivery application (see Figure 9) starts when the client calls the Pizza Company to order a pizza. After that, the system tries to authenticate the client by using an authentication service (Client Authentication). In case the client is properly authenticated, the system invokes a profile Web service to obtain the preferences of the client (Get Client Profile). Then, the system gets the client’s preferences that have been registered earlier in a Pizza Catalog service (Get Preferred Pizzas From Catalog Service) and sends them to the client to choose the desired pizza (Ask Client to Choose the Desired Pizza).

When the client chooses a pizza and agrees with the price, his/her credit card number is sent to a Credit Card Validation Web Service to be validated (Validate Client Credit Card). If the client’s credit card is valid, it is debited (Credit Card Transaction) and the pizza baker is notified to start cooking the selected pizza (Communicate Desired Pizza to Baker). After that, the system uses the client’s telephone number to obtain the client’s address from a Phone Company Web Service (Get Client Address From Phone Company Service).

With the client’s address, a GPS Web Service is invoked to get the coordinates of this address (Get Address GPS Coordinates) and then these coordinates are sent to a Map Web Service responsible for generating the map and the best route from the pizza company to the client’s address (Get Map and Route to GPS Waypoint). After that, the system asks both a Multimedia Messaging Service (MMS) to send the map and route to the personal digital assistant (PDA) of

Figure 9. Test case of service composition
the pizza delivery boy (Send Map to Delivery Boy PDA) and a Short Message Service (SMS) to send text message to the client's phone to inform that the pizza will be delivery within $n$ minutes (Send SMS to Client).

Each task of the service composition is performed by a Web service whose cost behaviours and reliability are shown in Table I. The cost behaviours adopted in this paper were created or adapted from cost models of actual services, such as CDYNE (‘CDYNE,” n.d.), Twilio (“Twilio,” n.d.), and Shopify (‘Shopify,” n.d.). In contrast, the reliability of these services was generated randomly from the range of values between 0.95 and 0.999, a range that is suitable for commercial services.

4.2 Simulation

All services were implemented in Java, deployed in a Tomcat server, and registered in a jUDDI server. Moreover, all service data were persisted in a MySQL (Oracle, n.d.) database installed in the same machine.

The service composition was modelled and executed using Activiti (Rademakers, 2012). The AuthService service was used to perform task Client Authentication with a fixed cost of $59.00 per month. In order to simulate different scenarios, we defined that 95% of the authentication requests were valid and 5% of them were invalid. Tasks Get Client Profile and Get Preferred Pizzas From Catalog Service were performed by Web service Database Web with a fixed cost of $9.90 plus a variable cost that depends on the number of requests. Task Ask Client to Choose the Desired Pizza has no cost and the price of the pizza chosen by the client costs on average $10.77. For the credit card validation by the CreditCard Validator, we assumed that 95% of the credit cards are valid and 5% are invalid. Service Credit Card Transaction performs task ShopCredit and has a fixed cost plus a variable cost that depends on both the value and number of transactions executed. Task Get Client Address From Phone Company Service is performed by PhoneABC, which has a step cost. The GeoCoding service has also a step cost and is performed by both Get Address GPS Coordinates and Get Map and Route to GPS Waypoint. Finally, the PhoneMessage service performs both Send Map

<table>
<thead>
<tr>
<th>Task</th>
<th>Service</th>
<th>Cost behaviour</th>
<th>Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client Authentication</td>
<td>AuthService</td>
<td>$59.00 / Month</td>
<td>0.988</td>
</tr>
<tr>
<td>Get Client Profile</td>
<td>Database / Web</td>
<td>$9.90 / Month + $0.05 per 100 requests</td>
<td>0.954</td>
</tr>
<tr>
<td>Get Preferred Pizzas From Catalog Service</td>
<td>Database / Web</td>
<td>$9.90 / Month + $0.05 per 100 requests</td>
<td>0.954</td>
</tr>
<tr>
<td>Validate Client Credit Card</td>
<td>CreditCard Validator</td>
<td>$0.05 per requests</td>
<td>0.980</td>
</tr>
<tr>
<td>Credit Card Transaction</td>
<td>ShopCredit</td>
<td>$29.00 / Month + 2.9% + $0.30 per transaction</td>
<td>0.956</td>
</tr>
<tr>
<td>Get Client Address From Phone Company Service</td>
<td>PhoneABC</td>
<td>$9.99 / Month + 0 - 1,000 = $0.03 1,001 - 3,000 = $0.02 3,001+ = $0.01</td>
<td>0.987</td>
</tr>
<tr>
<td>Get Address GPS Coordinates</td>
<td>GeoCoding</td>
<td>0-1,000 Queries / Month = $0.60 /100 Queries 1,001-3,500 Queries / Month = $0.40 /100 Queries 3,500+ Queries / Month = $0.30 /100 Queries</td>
<td>0.989</td>
</tr>
<tr>
<td>Get Map and Route to GPS Waypoint</td>
<td>GeoCoding</td>
<td>0-1,000 Queries / Month = $0.60 /100 Queries 1,001-3,500 Queries / Month = $0.40 /100 Queries 3,500+ Queries / Month = $0.30 /100 Queries</td>
<td>0.989</td>
</tr>
<tr>
<td>Send Map to Delivery Boy PDA</td>
<td>PhoneMessage</td>
<td>$0.0075 / Transaction</td>
<td>0.956</td>
</tr>
<tr>
<td>Send SMS to Client</td>
<td>PhoneMessage</td>
<td>$0.0075 / Transaction</td>
<td>0.956</td>
</tr>
</tbody>
</table>
to Delivery boy PDA and Send SMS to Client and has a variable cost that depends on the number of transactions executed.

4.3 Predicting Cost

We simulated 30 days of service composition executions in three different average daily executions. The independent variable of our experiment is the average of daily service composition executions \( n_i \) and the dependent variable is the cost of the service composition. In order to obtain the average of daily service composition executions, we decided to generate three random numbers \( n_1, n_2, \) and \( n_3 \), which we arbitrarily decided that should not be smaller than 20 neither bigger than 80, in order to enforce a minimum and maximum number of daily invocations, respectively. Additionally, we decided that the distances \(|n_1 - n_3|, |n_2 - n_3|\) and \(|n_2 - n_1|\) should not be smaller than 10 to prevent that these numbers were too close to each other. We executed the simulations with the values \( n_1 = 29, n_2 = 47 \) and \( n_3 = 62 \), and compared the cost computed by the services providers taking into account the cost behaviours shown in Table I with the cost computed with the following approaches:

- Our approach using models;
- Our approach using event logs; and
- The approach proposed in (Sampathkumaran, 2013).

Following our approach to predict the costs of a service composition using models (see Figure 8), we first modelled the service composition (Model Service Composition) and the cost behaviour of all services (Model Cost Behaviours). After that, we annotated the business process with the task reliability (Annotate Reliability), assigned probabilities to all alternative paths of the business process (Annotate Conditional Statement Probabilities), defined cost behaviours (Annotate Cost Behaviour Identification) and cost driver values (Annotate Cost Driver Values). Algorithm 1 was then executed to compute the probability of correct execution (PCE) of each service and the average cost driver values (CDV) (Compute Cost Driver Values). These cost driver values are obtained by multiplying PCE (average value per invocation of each cost driver of the respective service) by the number of invocations in the period. Finally, Algorithm 2 was executed to compute the cost of each service according to its respective cost behaviour (Compute Cost of Service Composition).

Following our approach to predict the cost of a service composition using event logs, we first got the event log of the simulations. After that, Algorithm 4 was executed to compute the average cost driver values (CDV). Finally, Algorithm 2 was executed to compute the cost of each service according to its respective cost behaviour.

To compute the cost of the service composition by using the approach of average cost of service, we first computed the average cost of each service followed by the computation of the cost for \( n_1 = 29, n_2 = 47, \) and \( n_3 = 62 \). To compute the average cost of the services, we used their cost behaviour and reliability (see Table I) and the average of values \( n_1, n_2, \) and \( n_3 \), that is, 46 invocations per day. Table II

![Figure 10. Total cost of the service composition computed in different approaches](image)

Table II. Average Cost per Invoke for 46 Invocations per Day.

<table>
<thead>
<tr>
<th>Service</th>
<th>Average of Invoices per Day</th>
<th>Average of Invoices per Month</th>
<th>Average Cost per Invoke</th>
</tr>
</thead>
<tbody>
<tr>
<td>AuthService</td>
<td>45.448</td>
<td>1,363.44</td>
<td>$0.04327</td>
</tr>
<tr>
<td>Database Web</td>
<td>80.483</td>
<td>2,414.49</td>
<td>$0.00460</td>
</tr>
<tr>
<td>CreditCard Validator</td>
<td>38.509</td>
<td>1,155.27</td>
<td>$0.05000</td>
</tr>
<tr>
<td>ShopCredit</td>
<td>34.974</td>
<td>1,049.21</td>
<td>$0.04327</td>
</tr>
<tr>
<td>PhoneABC</td>
<td>34.519</td>
<td>1,035.57</td>
<td>$0.03974</td>
</tr>
<tr>
<td>GeoCoding</td>
<td>67.903</td>
<td>2,037.10</td>
<td>$0.00498</td>
</tr>
<tr>
<td>PhoneMessage</td>
<td>63.136</td>
<td>1,894.09</td>
<td>$0.00750</td>
</tr>
</tbody>
</table>
shows the average costs per invocation we obtained as result.

Our experiment shows that the costs computed by our approaches are similar to the costs computed by the service providers (see Figure 10). In contrast, the cost computed by the approach using average costs is similar only in the case of 47 average invocations per day. This is because the average costs of the services were computed considering 46 invocations per day. This shows that the approach that uses average cost is not effective when the demand of the service varies, in contrast of our approaches that compute the cost correctly in all cases.

Considering the cost of each service individually as in Figure 11, we can see that the costs computed by both our approaches are similar to the ones computed by each provider. However, the costs computed with the approach that uses average costs are similar to the costs computed by each provider only when the service cost has a variable behaviour, such as the CreditCardValidator and

![Figure 11. Cost of the services computed in different approaches](image-url)
PhoneMessage services. In the other cases (i.e., fixed, mixed and step cost), the approach that uses average cost does not yield the expected results, such as in AuthService, DBService, ShopCredit, PhoneABC, and GeoCoding. As a consequence of this inaccuracy, a service with a higher cost can be chosen to replace an abstract service instead of another service with lower cost.

5. RELATED WORK

Magnani et al. (Magnani & Montesi, 2007) proposes an approach to analyse the cost of business processes by adding cost properties to BPMN elements. Similarly to our solution, they also add probabilities to all alternative branches in the process. However, only the average cost is annotated in each element in BPMN, and neither reliability nor cost behaviours are considered.

Saeedi at al. (Saeedi et al., 2010) proposes a BPMN extension to predict cost. Unlike (Magnani & Montesi, 2007), Saeedi’s approach is also able to predict performance and reliability of service composition. However, in contrast with our approach, they do not take into account all cost behaviours, and the reliability and performance of the services are not used to compute the cost of the service composition.

Sampathkumaran (Sampathkumaran, 2013) also proposes a BPMN extension to predict the cost of business processes. Additionally, this approach takes into account the reliability of each task when computing the cost of the service compositions. Like other approaches, only the average cost is annotated in the BPMN task, and the cost of the services is computed by only considering the number of invocations. In our approach, instead of annotating the average cost, we annotate the cost behaviour of each task.

Van Hee, Sidorova and Stahl (Van Hee, Sidorova, & Stahl, 2007) computes the expected cost of a service composition by considering the four basic structural workflow models (i.e., sequential, parallel, conditional and loop) and the probability of successful termination for each task. Likewise, Giersd and Sürmeli (Giersd & Sürmeli, 2010) propose a static analysis of the service behaviour by approximating the service cost on a limited set of interaction partners. In contrast with our approach, they use maximum and minimum cost to predict the cost of service compositions.

Wynn at al. (Wynn, Low, Hofstede, & Nauta, 2014) present a framework for reporting and predicting the cost of business processes with cost behaviour annotations. Moreover, their approach uses ProM (Aalst, 2011) in their framework for accounting business process costs. To compute process costs, ProM receives cost-annotated event logs enriched with detailed cost information, and a business process cost behaviour. In the absence of a log, the framework generates logs that can be used to predict cost. In their work, only variable costs are supported, since they treat costs per invocation as fixed costs. Our approach does not use process mining, but rather considers that business event logs should not be annotated with explicit cost values, since in the case of step cost behaviour the cost can vary during the accounting period, for example, due to discounts and execution failures.

Several approaches concentrate on the selection of services to minimise the workflow execution cost in a Grid, while meeting the deadline for delivering results (Yu, Buyya, & Tham, 2005), (Yuan, Li, Wang, & Zhu, 2007), (Ma, Gong, & Zou, 2009), (Abrishami, Naghibzadeh, & Epema, 2012). In (Yu et al., 2005), a workflow management system is provided for discovering appropriate services for processing the workflow tasks and schedule the tasks as services. In this approach, if a task is delayed, the scheduler adjusts the reservation schedule for the next child tasks to compensate the delay at runtime. Aiming to solve a similar problem, Yuan et al. (Yuan et al., 2007) provide a heuristic method, called deadline early tree (DET), to minimize the workflow execution costs with the deadline constraints during the allocation of services to tasks. Ma et al. (Ma et al., 2009) introduce a cost-gradient metric to provide criteria for service selection, and a cost minimisation scheduling approach with a specified deadline. The aforementioned approaches consider only variable costs of each service in a composition. However, services have usually more complex cost behaviours, as we have shown in this paper. Ramacher and Monch (Ramacher & Monch, 2012) propose a mixed integer programming formulation to select services. This minimises the expected cost by taking into account complex cost behaviours, such as fixed, variable, mixed and step cost, and execution time. However, they ignore that variable costs can be associated to many business attributes (cost drivers), such as message size and credit card transaction value.

6. FINAL REMARKS

In this paper, we propose an architecture and the necessary algorithms to manage the cost of service compositions in their life-cycle. Unlike related work, our approach takes into consideration all classes of cost behaviours, i.e., variable, fixed, mixed and step cost. Moreover, since reliability can affect the cost of the entire composition, we also take into account reliability in the computation of the service composition cost.

In our approach, a service composition is not annotated with cost information as a whole, but we prescribe the annotation of each service of the composition with its cost behaviour, including the average values of the cost drivers that play a role in the service invocations. Since many services can have their cost behaviour modified over time, our approach calculates the cost of the composition by taking into account the average number of service composition executions during an accounting period, and using event logs.

In order to evaluate our approach, we developed the proposed architecture shown in Figure 7, a service-oriented
application shown in Figure 9 and carried out experiments to simulate the cost of service composition executions. In these experiments, we evaluated the effectiveness of our approach by comparing the costs computed by the service providers with the costs computed with our approach. From this comparison we concluded that the costs computed with our approach are similar to ones computed directly by the service providers, which demonstrated the suitability of our approach. Additionally, we compared the cost computed by the service providers with the cost computed using average costs. From this comparison we concluded that the cost using average costs only provides reliable results when the cost has a variable behaviour.

As future work, we intend to apply our approach to a more realistic and complex case study. Moreover, we intend to work on simulation techniques to predict the cost of service compositions that consider all classes of cost behaviour. We believe that simulation can improve even more the accuracy of the cost prediction. Moreover, we intend to investigate algorithms to select optimal sets of services in a service composition by taking into account these classes of cost behaviour. Furthermore, we also intend to work on dynamic runtime service composition adaptation, taking not only cost but also other non-functional requirements into consideration.

7. References
Oracle. (n.d.). MySQL.
Siguencia Guzman, L., Van den Abbeele, A., Vandewalle, J., Verhaaren, H,
Luís Ferreira Pires is an Associate Professor at the University of Twente, the Netherlands. He obtained his B.Sc. degree in Electronics from the Technological Institute of Aeronautics (ITA, São José dos Campos, Brazil) in 1983, and his M.Sc. degree in Electrical Engineering from the University of São Paulo (Poli-USP, São Paulo, Brazil), in the Computer Networks specialisation, in 1989. He obtained his PhD degree from the Faculty of Electrical Engineering of the University of Twente, Enschede, the Netherlands, in September 1994. He contributed to various national and international research projects, and co-authored more than 150 research papers in international conferences, workshops and journals. His research interests include design methodologies for distributed systems, architecture of distributed systems, modelling and specification techniques, middleware platforms and telematics applications. He is currently Programme Director of the Business Information Technology B.Sc. and M.Sc. Education Programmes of the University of Twente.

Robson W. A. de Medeiros is an Assistant Professor at Rural Federal University of Pernambuco, Brazil. He received his B.Sc. degree in Computer Science from the Catholic University of Pernambuco (UNICAP, Recife, Brazil), and his M.Sc. degree in Computer Science from the Federal University of Pernambuco (UFPE, Recife, Brazil). Currently, he is a PhD candidate in the field of Service-Oriented Computing at the Federal University of Pernambuco (UFPE, Recife, Brazil) and University of Twente (Enschede, the Netherlands). His main research interests include service computing and quality of service.

Nelson Rosa received the BS, MS and PhD degrees in computer science from Federal University of Pernambuco, Brazil. He is an Associate Professor in the Centre of Informatics of the Federal University of Pernambuco, Brazil. His current research interests include formalization, design and implementation of middleware systems, service oriented computing and cloud computing. He has published more than 60 research papers in the area of middleware, was the Brazilian leader on a 3-years collaboration project with HP Labs (Palo Alto). While in HP’s collaboration, he worked on the treatment of quality attributes of service compositions. He was the recipient of the IEEE International Services Cup award in 2011. He has served as committee member in several conferences on network and distributed systems.