**Noname manuscript No.**
(will be inserted by the editor)

# Genetic Programming for QoS-Aware Web Service Composition and Selection

**Alexandre Sawczuk da Silva · Hui Ma · Mengjie Zhang**

Received: date / Accepted: date

**Abstract** Web services, which can be described as functionality modules invoked over a network as part of a larger application, are often used in software development. Instead of occasionally incorporating some of these services in an application, they can be thought of as fundamental building blocks that are combined in a process known as Web service composition. Manually creating compositions from a large number of candidate services is very time-consuming, and developing techniques for achieving this objective in an automated manner becomes an active research field. One promising group of techniques encompasses *evolutionary computing*, which can effectively tackle the large search spaces characteristic of the composition problem. Therefore, this paper proposes the use of *genetic programming* for Web service composition, investigating three variations to ensure the creation of functionally correct solutions that are also optimised according to their *quality of service*. A variety of comparisons are carried out between these variations and two *particle swarm optimisation* approaches, with results showing that there is likely a trade-off between execution time and the quality of solutions when employing genetic programming and particle swarm optimisation. Even though genetic programming has a higher execution time for most datasets, the results indicate that it scales better than particle swarm optimisation.

**Keywords** Web service composition · Quality of Service · Genetic Programming · Conditional Constraints

A. Sawczuk da Silva, H. Ma, and M. Zhang
School of Engineering and Computer Science,
Victoria University of Wellington,
P.O. Box 600, Wellington 6140, New Zealand
E-mail: {sawczualex,Hui.Ma,Mengjie.Zhang}@ecs.vuw.ac.nz

## 1 Introduction

Web services are popular building blocks for applications, since they are independent and self-contained modules whose functionality can be accessed over a network using standard communication protocols. *Web service composition* is when these blocks are combined into a larger application with minimal custom code writing (Milanovic and Malek, 2004), thus preventing developers from recreating already existing components and consequently accelerating the development process. The combination of services can be performed manually, though the large number of candidates available would cause this to be a very time-consuming process, especially when it is necessary to *select* the most suitable option amongst many services that offer the same functionality. This fundamental problem remains open, and it has been the focus of a growing body of research that aims to propose suitable techniques for performing automated Web service selection and composition.

Evolutionary computing (EC) techniques are a promising research direction to address the Web service composition problem, because they can effectively handle large search spaces by making use of non-deterministic search strategies inspired by biological concepts such as populations, generations, and fitness of individuals (Rao and Su, 2005). In particular, Genetic programming (GP) has been shown to be suitable to fully automated Web service composition (Aversano et al., 2006), which is when the relevant services for the composition are selected and their configuration is determined simultaneously. Despite these advantages, enforcing composition constraints while also optimising the quality of solutions is still an open problem in GP (Rodriguez-Mier et al., 2010; Wang et al., 2013). Thus, the goal of this work is to investigate different service composition approaches in GP in order to determine which is the most promising.

The first approach investigated enforces the correctness of the connections between services by penalising candidates via the fitness function, but candidate initialisation and genetic operations are performed at random. This approach is simple to implement, and may result in good composition results. The second approach initialises candidates by using an algorithm that ensures that the connections between services are correct, and maintains that correctness by employing restricted genetic operators throughout the evolutionary process. Finally, the third approach extends the second approach include conditional constraints, providing the additional benefit of allowing compositions with multiple execution paths to be created. The first and second approaches are compared with two previously proposed Particle Swarm Optimisation (PSO) solutions (da Silva et al., 2014) to ascertain whether there is a difference in overall performance, and if so which method is better suited to addressing the composition problem. Experiments are also conducted to evaluate the performance of the third approach.

This paper assumes that the reader is familiar with the concepts of genetic programming (Koza, 1992) and particle swarm optimisation techniques (Kennedy et al., 2001), and the remainder of the paper is organised as follows. Section 2 provides the necessary background and an overview of the research conducted in this area; Section 3 presents a GP composition approach that relies primarily on the fitness function; Section 4 presents a GP approach that constraints the initialisation and operations on candidates; Section 5 extends the previous section by allowing candidates to incorporate the choice construct; Sections 6 and 7 discuss the experiments conducted using these three approaches; Section 8 draws conclusions and points out future work alternatives.

## 2 Background

### 2.1 Problem Description

Fundamentally, the objective of a Web service composition is to combine services in a way that meets the task requirements while at the same time ensuring that services are executable, that is, ensuring that the composition is *functionally correct.* A functionally correct composition may be defined as a composition where all the inputs of each included service are satisfied with compatible values, either from a preceding service's set of outputs or from the overall set of composition inputs. A value is considered to be *compatible* or *matched* to a given input if it can be conceptually classified under that input, e.g. a *City* value matches a *Location* value, since these two are conceptually suitable. A classic example of automated Web service composition is the travel planning scenario (Srivastava and Koehler, 2003), where the objective is to create a system capable of automatically reserving hotels and flights according to customer preferences. In this scenario the customer preference types, such as departure date and destination city, are the composition *inputs*, and the reservation outcomes, such as issued tickets and receipts, are the composition *outputs.* The relevant composition candidates are a set of hotel and flight booking services that are to be combined into a cohesive task workflow by the composition system. Figure 1 shows a simple composition solution that performs flight and hotel reservations according to a customer's information. More specifically, when using this composite service the customer provides her/his personal information and travel details, such as the departure date, the destination city and the duration of the stay. This information is then used to book return flight tickets, and to determine the customer's arrival date at the destination city.

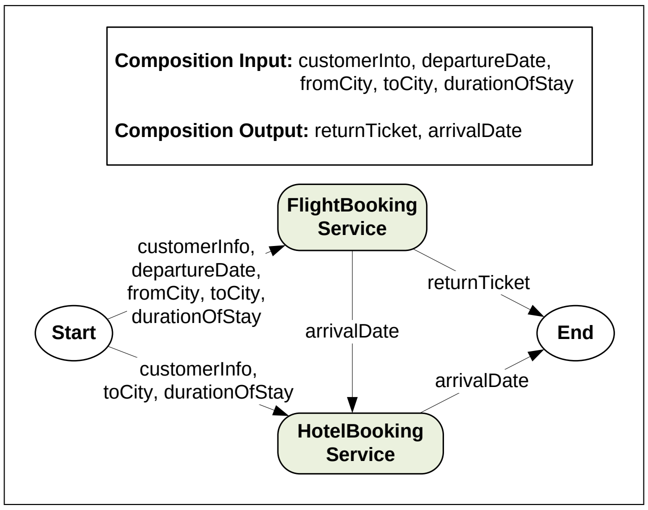

Composition Input: customerInto, departureDate, fromCity, toCity, durationOfStay
Composition Output: returnTicket, arrivalDate
FlightBooking Service
customerInfo, departureDate, fromCity, toCity, durationOfStay
returnTicket
Start
arrivalDate
End
customerInfo, toCity, durationOfStay
arrivalDate
HotelBooking Service

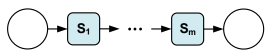
**Fig. 1** Example of a solution to a Web service composition task.

The problem with only focusing on the functionality of services is that their non-functional characteristics are overlooked, failing to consider *Quality of Service* (QoS) measures (Menascé, 2002) such as service availability and response time. To overcome this issue, it is necessary to employ QoS-aware Web service composition techniques which simultaneously cater for functional and non-functional needs, as is done in this work. Four popularly considered QoS attributes (Jaeger and Mühl, 2007; Yu et al., 2013) are included: availability

($A$), which is the probability that a service will be available for execution when a request is sent; reliability ($R$), which is the probability of a service returning an appropriate response within the estimated time; cost ($C$), which is the financial cost associated with executing a service; time ($T$), which is the length of time required for a service to return a response. In the case of $A$ and $R$, the highest values correspond to the highest quality, whereas the inverse is true of $C$ and $T$.

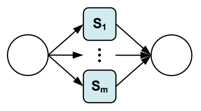
There are four basic constructs supported by popular composition languages (such as BPEL4WS (Van der Aalst et al., 2003)) that describe how services interact in a composition, as described below:

- **Sequence construct:** Services are connected sequentially in this construct, so that the outputs of an earlier service are used to fulfil the inputs of a subsequent one. As shown in Figure 2, the total availability ($A$) and reliability ($R$) can be calculated by multiplying the values of all individual services in the composition (since they are probabilities), and the total time ($T$) and cost ($C$) can be calculated by adding up the values of the individual services.
- **Parallel construct:** Services are independently executed in parallel, meaning that the inputs and outputs of each service are fulfilled and produced independently. As shown in Figure 3, the QoS attributes for this construct are calculated the same way they are in the sequence construct, except for the total time ($T$). In this case, $T$ is obtained by selecting the service that has the highest individual execution time (as the other services can be fully executed within that amount of time).
- **Choice construct:** One of the services organised using the choice construct is executed at runtime, depending on whether the choice condition is met. For example, the choice condition may specify that service $X$ is executed if a *Location* provided is a *US Location*, and service $Y$ is executed otherwise. As shown in Figure 4, the cost, time, reliability and availability of this construct are calculated as a weighted sum of the QoS values from each service options, where the weights ($P_n(n \in \{1..m\})$) correspond to the probability of that service being selected (i.e. the probability of the choice condition being met).
- **Loop construct:** The services placed within a loop construct are executed repeatedly, provided the loop condition is met. For example, the loop condition may specify that another iteration is to be carried out if a pre-specified number of executions has not yet been reached. The total quality attributes of this construct are calculated according to the topology of the service(s) included within the loop (resulting in $t$, $c$, $a$, $r$), adjusting each of these values according to the number of iterations completed ($k$) as shown in Figure 5.



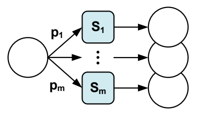
$$T = \frac{\sum_{n=1}^{m} t_n}{m} \qquad C = \frac{\sum_{n=1}^{m} c_n}{m} \qquad A = \prod_{n=1}^{m} a_n$$
$$R = \prod_{n=1}^{m} r_n$$

**Fig. 2** Sequence construct and calculation of its QoS properties (da Silva et al., 2015).



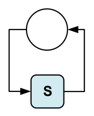
$$T = MAX\{t_n | n \in \{1, \ldots, m\}\}$$
$$C = \frac{\sum_{n=1}^{m} c_n}{m} \qquad A = \prod_{n=1}^{m} a_n \qquad R = \prod_{n=1}^{m} r_n$$

**Fig. 3** Parallel construct and calculation of its QoS properties (da Silva et al., 2015).



$$T = \sum_{n=1}^{m} p_n t_n \qquad C = \sum_{n=1}^{m} p_n c_n \qquad A = \sum_{n=1}^{m} p_n a_n$$
$$R = \sum_{n=1}^{m} p_n r_n$$

**Fig. 4** Choice construct and calculation of its QoS properties (da Silva et al., 2015).



$$T = k * t \qquad C = k * c \qquad A = a^k \qquad R = r^k$$

**Fig. 5** Loop construct and calculation of its QoS properties (Yu et al., 2013).

### 2.2 Objective Function

A key component of the optimisation process is the objective function, which is used to optimise the overall quality of candidates produced using QoS-aware composition techniques. This function works by maximising the QoS scores of a candidate solution $i$ (Alrifai and Risse, 2009) according to the four QoS attributes discussed above:

$$\begin{aligned} obj &= Max(f_i) \\ f_i &= w_1 A'_i + w_2 R'_i + w_3(1 - T'_i) + w_4(1 - C'_i) \end{aligned} \quad (1)$$

where $\sum_{j=1}^{4} w_j = 1$.

The calculation of each total QoS attribute is performed using the formulae described for each construct. When calculating these QoS values in a tree structure, the attributes of each tree node can be computed by using its immediate children as the service nodes; these children may be either atomic (i.e. terminal nodes) or composite services (i.e. non-terminal nodes). The function produces values within the range [0, 1], with 1 corresponding to the highest possible quality and 0 corresponding to the worst. In order to achieve scores that are within the desired range, overall QoS values are normalised between 0 and 1 (denoted as $A'$, $R'$, $C'$, and $T'$) using the lowest and highest values of the relevant services (which are identified as shown in Algorithm 1) as lower and upper bounds. In the case of $T$ and $C$, the upper bounds are the sum of the values from all relevant services. Finally, $C$ and $T$ scores are offset as $(1 - C)$ and $(1 - T)$ to allow the function to perform maximisation.

## 2.3 Existing GP-based Composition Approaches

One popular group of approaches relies on GP to perform Web service composition, since GP is suitable to solving problems where the changes in the topology configuration of the solution must be explored (Dupuis et al., 2012). Mucientes et al. (2009) and Rodriguez-Mier et al. (2010) rely on a context-free grammar to ensure that the initial population of candidates is functionally correct. In these works, compositions are represented as trees where the terminal nodes are atomic services while the non-terminal nodes refer to topological constructs of the composition. During the evolution process, the correctness of candidates is maintained by ensuring that any genetic operations check input-output matches. These approaches were tested using a variety of composition problems and service repositories, with results indicating that GP is a robust technique when applied to the composition problem. However, an important limitation is that these approaches estimate the quality of solutions using topological measurements such as execution path lengths (obtained by measuring the tree's depth Mucientes et al. (2009)) as opposed to considering QoS measurements.

A similar technique is proposed by Wang et al. (2013), ensuring that candidates are functionally correct when initialised and that future generations remain correct. In this work, compositions are represented as trees where the root node is the composition output, the terminal nodes are the composition inputs, and the internal nodes are atomic services forming data flows from the leaves towards the root of the tree. This approach never produces candidates that do not fulfil the composition task provided, as can happen in the work of Rodriguez-Mier et al. (2010), since in this case a greedy search algorithm is used for generating candidates and performing mutation (as opposed to using a grammar). Despite these advantages, the fitness function used in this work does not consider QoS measurements, instead estimating quality by counting the number of services including in the composition (the smaller the composition, the better the quality).

A GP Web service composition framework is proposed by Xiao et al. (2012). Similarly to the work of Mucientes et al. (2009), compositions are represented as trees with atomic service terminals and composition construct non-terminals. The key feature of this approach is that the fitness function employed uses black-box testing with automatically generated scenarios, in addition to checking for functional correctness. The use of black-box testing adds a semantic dimension to the composition process, ensuring that compositions are not only correct but also behave as expected. Despite this, QoS measurements are once again neglected as the way to measure the quality of solutions, a similar pattern that is used in Aversano et al. (2006).

The approach of Yu et al. (2013) relies on a single fitness function that penalises solutions that are not fully functionally correct by lowering their overall score, while at the same time rewarding the selection of services that lead to good overall QoS. Once again, compositions are represented by placing atomic services in the leaf nodes, organised according to topological constructs in the inner nodes. The main advantage of this method is that it is capable of taking global QoS measures into account during the evolutionary process. However, its penalisation strategy is not guaranteed to produce final solutions that are entirely functionally correct.

## 2.4 Existing PSO-based Composition Approaches

Another popular group of approaches for Web service composition relies on PSO, and some of these works are discussed here. The fitness function designed by Amiri and Serajzadeh (2012) considers a number of QoS attributes when calculating the quality of a solution, including the reputation of services and their rate of successful execution. This function is then used during the PSO iterations, with each particle investigating one location of the solution space at each iteration. The assumption this work makes is that before the optimisation, an abstract workflow where the steps needed in the composition have already been defined in as abstract services, which means that the particles are only tasked with finding the best possible concrete service to fill each abstract service slot (without any concerns for the topology of the solution). Because of this, solutions are represented as vectors where each cell contains an atomic service chosen to fulfil a given slot. The preselection of an abstract workflow may simplify the composition process, but it has the limitation of requiring either a domain expert to make this choice or some automated selection strategy.

The concept of an abstract workflow is also explored by the PSO approach proposed by Xia et al. (2009), consequently causing particles to have the same solution representation as the work described above. Unlike the previous approach, however, this work employs a multi-objective strategy for identifying the solutions with promising QoS attributes. Instead of producing a single composition solution at the end of a run, this approach produces a set of solutions containing the best trade-off identified between multiple conflicting quality attributes. Cost and availability are examples of two conflicting QoS attributes: the ideal composition would have the lowest possible cost while simultaneously offering the highest possible availability, but in reality a trade-off between these two factors is more likely. This approach has the advantage of considering the perspective of all quality measures individually (Rezaie et al., 2010), though it also suffers from the limitations caused by preselecting a workflow.

The approach of Ludwig (2012) once again requires an abstract workflow to be provided, though this time changing the way in which the position of particles is updated. As with the previously discussed works, solutions are represented as vectors of services containing a candidate for each workflow slot. The position update strategy applies a list of changes to each particle, with each change replacing certain service candidates. Additionally, a local search technique is incorporated into the process to prevent the stagnation of particles in an attempt to prevent convergence to local optima. While the PSO search strategy is different from the one discussed in previous works, the issues associated with the workflow preselection step are still present.

## 3 First Approach: GP Composition based on Fitness Function

The first approach investigated in this work uses penalisations in the fitness function to encourage the correctness of solutions. It bears some similarity to the work of Yu et al. (2013), however in our approach the fitness function separates the functional correctness penalisation and the QoS optimisation into two distinct value ranges, whereas that work does not make such a distinction. The chosen representation for candidates is that of a tree, with inner nodes representing the topological constructs in the composition and the leaf nodes representing the atomic Web services included in the solution, as shown in Figure 6 (note that the particular composition shown in this figure does not make use of a parallel construct, though in this representation such a construct can also be included as an inner node). The inputs and outputs of each composition construct node are determined by analysing the I/O characteristics of its subtrees. One advantage of this representation is that a simple tree traversal can be used to determine the inputs, outputs, and overall QoS values of each node in the candidate, simplifying fitness calculations. Another advantage is that the use of composition construct nodes and service nodes allows for the exploration of promising workflow topologies in tandem with the improvement of the overall QoS attributes. However, candidates that are not fully functionally correct may be produced during the evolutionary process, making it necessary for the fitness function to penalise such individuals.

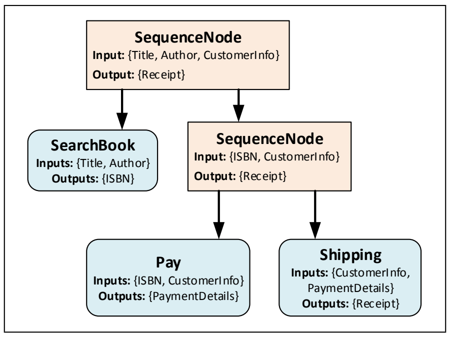

SequenceNode
Input: {Title, Author, CustomerInfo}
Output: {Receipt}
SearchBook
Inputs: {Title, Author}
Outputs: {ISBN}
SequenceNode
Input: {ISBN, CustomerInfo}
Output: {Receipt}
Pay
Inputs: {ISBN, CustomerInfo}
Outputs: {PaymentDetails}
Shipping
Inputs: {CustomerInfo, PaymentDetails}
Outputs: {Receipt}

**Fig. 6** Example of tree representation for Web service composition.

### 3.1 Fitness Function

The fitness function employed by this approach is its main contribution, since it simultaneously improves the overall QoS of candidates and penalises those compositions that are not fully functionally correct. This is similar to the work of Yu et al. (2013), however in our case two separate value ranges are produced by the function: values in the lower range indicate that the composition is not entirely functionally correct; values in the upper range indicate the QoS score of a fully functionally correct solution. This distinction was created to encourage the prioritisation of functional correctness over non-functional optimisation.

The values produced by the fitness function vary between -1 and 1, with the range $[-1, 0]$ indicating the degree of functional correctness of a solution (where 0 is full correctness), and the range $[0, 1]$ indicating the is QoS score (where 1 is the best possible quality). As mentioned before, a candidate is traversed using a depth-first algorithm to determine the functional and non-functional characteristics of each node in the tree. This information is then used for the fitness calculation.

The leaf nodes of the tree represent atomic Web services, so in this case their QoS values (i.e. their functional characteristics) are already known and do not need to be calculated. With regards to functionality, they are always given a functional score of 0 (i.e. not considered). Inner nodes, on the other hand, must have their QoS values calculated according to their corresponding formulae (shown in Subsection 2.1), and for the purposes of this calculation each of their children is handled as an atomic service.

Since all the nodes in a parallel construct are executed independently, the parallel cannot enforce any constraints between services. Therefore, just as with the atomic Web services parallel nodes do not figure in the calculation of the functional correctness score of a composition. On the contrary, sequence nodes denote a relationship between the outputs of a service and the inputs of its successor, therefore a functional correctness score can be calculated for sequences. This can be done by averaging the percentage of successfully matched inputs for each node in the sequence, as shown below (note that this results in a value between 0 and 1):

$$average = \frac{\sum_{i=2}^{n} \frac{|output_{i-1} \cap input_i|}{|input_i|}}{n} \tag{2}$$

where $n$ denotes the amount of sequence node's children.

All sequence nodes in the tree are visited, each time calculating their score and adding this value to an overall running average for the entire tree. After they have all been visited, the matching score for the overall task inputs ($in_{req}$) and outputs ($out_{req}$) is computed and added to the running average. As a final step, the average is offset with -1 in order to produce a final value within the range $[-1, 0]$. If a fully functional solution has been achieved (i.e. the value is 0), then the objective function $f_i$ discussed earlier (Equation 1) is used to calculate the solution fitness; otherwise, the functional score becomes the final fitness. This fitness function is shown below:

$$fitness(i) = \begin{cases} f_i, & \text{if } func(i) = 0 \\ func(i) & \text{otherwise} \end{cases} \tag{3}$$

where

$$func(i) = -1 + \frac{w_5(\frac{|in_i \cap in_{req}|}{|in_{req}|}) + w_6(\frac{|out_i \cap out_{req}|}{|out_{req}|}) + treeScore(root)}{2} \tag{4}$$

$w_5 + w_6 = 1$ and $treeScore(root)$ is a procedure that recursively traverses the candidate and calculates the running average for the sequence constructs in the composition using Equation 2.

### 3.2 Generation of Initial Candidates, Mutation and Crossover

The initial population is randomly generated. The genetic operations employed are *crossover*, in which the randomly selected subtrees of two candidates are swapped, and *mutation*, in which a subtree is randomly replaced with a new arbitrarily generated substitute.

## 4 Second Approach: Constrained GP Composition

The second approach investigated in this work ensures that solutions also retain their functional correctness throughout the evolutionary process. This is done by using a population initialisation algorithm that creates functionally correct solutions, and then by employing restricted evolutionary operators that maintain the functional correctness. In comparison to the first approach, the advantage of this approach is twofold: firstly, any of the solutions produced by the technique can be used, regardless of their quality; secondly, the restriction to functionally correct solutions significantly reduces the problem's search space, likely yielding better solutions. Similarly to the first approach, candidates are represented using trees where inner nodes consist of parallel and sequence constructs that direct the flow of the composition, and leaf nodes consist of the Web services used as basic components. Each parallel and sequence construct requires a set of inputs and produces a set of outputs according to the nodes that compose its subtree. Functional correctness is ensured by generating the initial composition candidates and mutating subtrees through the use of a greedy algorithm, as well as restricting crossover operations to the exchange of functionally equivalent subtrees. This approach is different from that discussed in Wang et al. (2013), as the candidate representation in that work has the inner nodes of the tree being the atomic services included in the composition and the parent-child relationships being the edges of the corresponding composition workflow. In this paper, the inner nodes of the tree are composition constructs, while the leaf nodes are atomic services. Consequently, the population initialisation and genetic operators also work in a different fashion. More detailed explanations of these techniques are presented later in this Section.

This particular composition representation was chosen because it allows for more flexibility when adding new constructs to the composition tree. A choice (if-else) construct, for example, is commonly supported by composition languages such as BPEL4WS and OWL-S (Milanovic and Malek, 2004), and can be added as an inner tree node with relative simplicity. On the other hand, it is not so simple to add a choice construct to composition representations that are based on the concept of a Directed Acyclic Graph (DAG), since conventional DAGs inherently represent deterministic flows of execution (Potthof et al., 1994). In fact, the representation adopted by Wang et al. (2013) was initially considered for this work, but was found to have limited extensibility in terms of constructs exactly for this reason.

### 4.1 Fitness Function

The fitness function employed in this technique is the objective function $f_i$ presented in Subsection 2.2 (Equation 1):

$$fitness(i) = f_i \tag{5}$$

Note that the fitness function only considers QoS values.

### 4.2 Generation of Initial Candidates

Instead of randomly initiating the first generation of candidates, a greedy algorithm is employed to create a fully functionally correct candidates for the initial population. This algorithm creates compositions represented as a DAG, so conversion to a tree representation must be performed subsequently (discussed in Subsection 4.3). A similar conversion is performed in Wang et al. (2013), though in that case the tree representation is entirely different. Consequently, a new algorithm had to be designed for this work. Before creating the compositions, however, it is necessary to discover all services that could possibly be used for that purpose. These are referred to as *relevant* services, and their discovery is performed using Algorithm 1. The discovery algorithm requires a set of composition task inputs ($I$), outputs ($O$), and a service repository ($R$). It begins by creating a set of outputs that can be used for discovering relevant services, and initialising it with the input values from the composition task ($I$). It then searches through the service repository, discovering all services whose input can be satisfied by values in the output set. The output of each discovered service is added to the output set, meaning that it can be used to discover further compatible services. This process continues as long as new services are discovered and returns the service list $L$, provided that the set of discovered services can be assembled into a composition that satisfies the expected output for the composition task ($O$).

```
Procedure discover()
    //Input  : $I$, $O$, $R$
    //Output: services $L$
    Initialise output set with $I$;
    Find services satisfied by output set;
    while at least one service found do
        Add services to $L$;
        Add the outputs of these services to the
        output set;
        Find services satisfied by the updated
        output set;
    end
    if Output set satisfies $O$ then
        return $L$;
    else
        Report no solution;
    end
```

**Algorithm 1:** Discovering relevant services for composition (adapted from Wang et al. (2013) and da Silva et al. (2014)).

After having discovered the relevant services for the given composition task, Algorithm 2 is used to generate a functionally correct composition candidate, expressed as a DAG. The algorithm requires a list of relevant services $L$, as well as a node *start* representing the starting point of the graph (*start* produces the composition task's) and a node *end* representing the ending point of a graph (*end* requires the composition's output as its input). It builds the composition graph $W$ backwards, starting from the *end* node (added to the queue of nodes to be processed) and working towards the *start* node. While the queue is not empty, its head is polled, the node is marked as visited, and a series of predecessor nodes whose outputs completely fulfil this node's inputs are selected (the predecessors may include the *start* node). Directed edges are added to $W$ to connect each of the predecessors to the node, and the predecessors are added to the queue (excluding the start node and nodes already visited). This process is repeated until the queue is empty, at which point the composition has been finished and $W$ is returned.

```
Procedure generate()
    //Input  : $L, start, end$
    //Output: workflow graph $W$
1:  Add $end$ node to the queue;
2:  while queue is not empty do
3:      Remove next node from queue;
4:      Add node to visited set;
5:      while not all node inputs satisfied do
6:          Select predecessor from $L$ whose
            outputs fulfil some/all node inputs;
7:          Add node, predecessor, and their
            edge to solution $W$;
8:          if predecessor not start and not
            visited then
9:              Add predecessor to queue;
10:         end
11:     end
12: end
13: return $W$;
```

**Algorithm 2:** Generating a functionally correct service composition (adapted from Wang et al. (2013) and da Silva et al. (2014)).

## 4.3 Graph-to-Tree Translation

After creating a composition candidate as a DAG, it is necessary to convert it to the tree representation that will be used for the GP evolutionary process, as illustrated by Figure 7. A notable outcome of this conversion process is that any service nodes with multiple predecessors in the DAG are replicated in the tree, appearing the same number of times as their number of predecessors. This is the case because trees, by definition, can encode at most one predecessor (parent) per node, meaning that multiple subtrees are necessary to accurately translate a service with multiple predecessors. The conversion is accomplished by employing Algorithm 3, a recursive function that requires a graph $W$ as input and produces a tree $T$ as output. The function $graphToTree$ traverses the graph in a breadth-first manner, starting from the *start* node and working its way to the *end* node. The currently selected node is $from$, and since the algorithm should begin at the *start* node, the initial value of $from$ is *start*. The algorithm then splits into three cases, which are individually covered below:

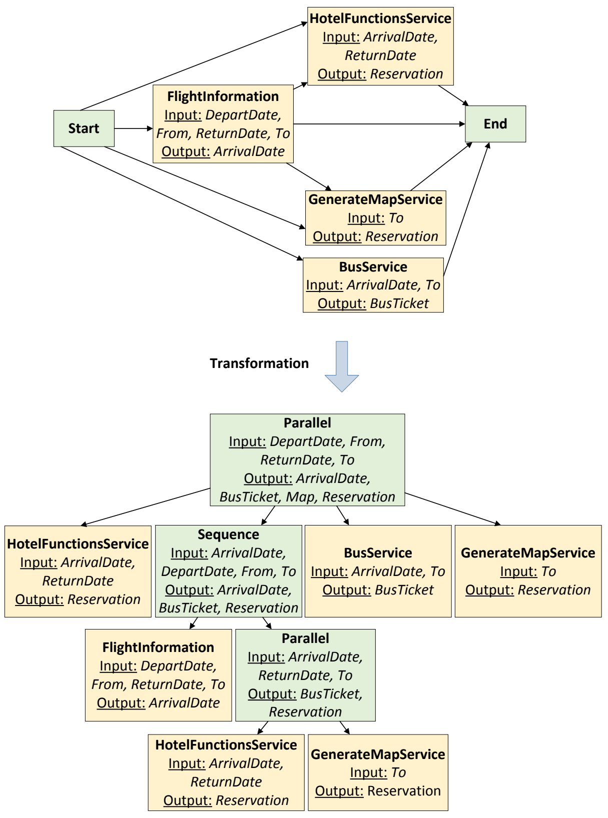


**Fig. 7** Example of a graph composition transformed into the corresponding tree composition.

1. $from$ **is a leaf:** A node is considered a leaf if its only outgoing edge points to the *end* node. If this is true, then the resulting tree $T$ will simply return the node itself, since all graph nodes represent Web services.
2. $from$ **is equal to the** *start* **node:** It is important to make a distinction between this case and the others, because the *start* node requires a slightly different structure when a parallel node is required. If the *start* node has a single child (i.e. feeds a single node with its output), then $graphToTree$ is called on this child and the result is returned as $T$. Otherwise, the *start* has multiple children, so a parallel node should be created to contain these children.

```
Procedure graphToTree()
    //Input  : $from, pInput, start, end$
    //Output: composition tree $T$
1:  Initialise tree $T$;
2:  if $from$ has no service successor then
3:      $T \leftarrow from$;
4:  else if $from = start$ then
5:      if $from$ has one successor then
6:          $T \leftarrow$ graphToTree($successor(from)$, $pInput, start, end$);
7:      else
8:          $T \leftarrow$ createParallelNode($from$, $successors(from), pInput, start, end$);
9:      end
10: else
11:     $rightChild$;
12:     $nodeChildren \leftarrow children(from)$;
13:     Create empty set of outputs $o$;
14:     if $end$ is a successor of $from$ then
15:         Remove $end$ from $from$'s successors;
16:         Add $end$ inputs to $o$;
17:     end
18:     if $from$ has one successor then
19:         $rightChild \leftarrow$ graphToTree($sucessor(from)$, $pInput, start, end$);
20:         $T \leftarrow$ createSequenceNode($from$, $rightChild, pInput, o$);
21:     else
22:         $rightChild \rightarrow$ createParallelNode($from$, $successors(from), pInput, start, end$);
23:         $T \rightarrow$ createSequenceNode($from$, $rightChild, pInput, o$);
24:     end
25: end
26: return $T$;
```

**Algorithm 3:** An algorithm for converting a DAG composition representation into a tree.

Note that this parallel node is not wrapped by a sequence node, as in the next case.

3. **Default case:** The children of $from$ are retrieved, and the algorithm checks whether the *end* node is among them. If so, *end* is removed from the list of children, since it should not be included as a leaf in the tree. Subsequently, the number of remaining children is checked. If there is only one child, a sequence node should be created using $from$ as the left node and the child tree root as the right node. Otherwise, a parallel node wrapped in a sequence node is created. More specifically, $from$ is assigned as the left child of the sequence node, and its children are placed in a parallel node which is assigned as the right child of the sequence node.

Functions for creating sequence and parallel nodes, $createSequenceNode$ and $createParallelNode$, are presented separately as Algorithms 4 and 5, respectively. These are described below:

1. $createSequenceNode$: a sequence node is created using the left child and right child provided. The inputs of this node are the combination of those required by the left child and those inherited from parent nodes (*pInput*); the outputs are the combination of those produced by the right child and optionally those produced by the left child (which should be provided in the *o* argument).
2. $createParallelNode$: a parallel node is created by generating the subtrees for each child provided, assigning these subtrees as children for the parallel node, and combining their individual inputs and outputs into overall parallel inputs and outputs.

```
Function createSequenceNode()
    //Input  : $leftChild, rightChild, pInput, o$
    //Output: sequence node $N$
1:  $N \leftarrow SequenceNode(leftChild, rightChild)$;
2:  $N$'s inputs are $leftChild$'s inputs plus $pInput$;
3:  $N$'s outputs are $rightChild$'s outputs plus $o$;
4:  return $N$;
```

**Algorithm 4:** An auxiliary function for creating a sequence node.

```
Function createParallelNode()
    //Input  : $node, children, pInput, start, end$
    //Output: parallel node $N$
1:  $N \leftarrow ParallelNode()$;
2:  forall the $child$ in $children$ do
3:      $childT \leftarrow$ toTree($child, pInput, start, end$);
4:      $child$'s inputs are added to $N$'s inputs;
5:      $child$'s outputs are added to $N$'s outputs;
6:      $childT$ is added as a subtree of $N$;
7:  end
8:  return $N$;
```

**Algorithm 5:** An auxiliary function for creating a parallel node.

## 4.4 Mutation and Crossover

The mutation operation ensures that the functional correctness of each candidate is maintained. It begins by

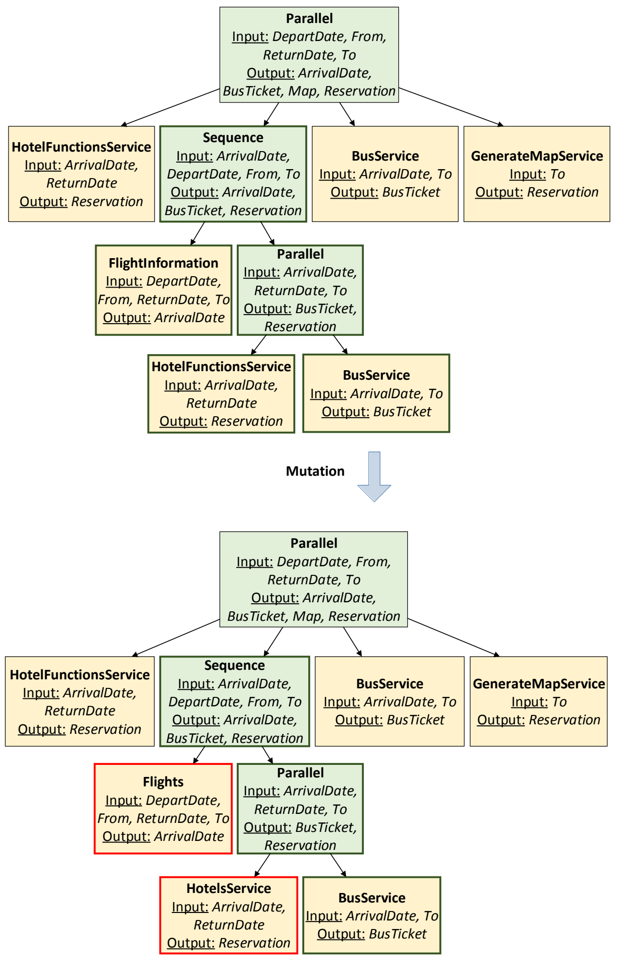

Parallel
Input: DepartDate, From, ReturnDate, To
Output: ArrivalDate, BusTicket, Map, Reservation
HotelFunctionsService
Input: ArrivalDate, ReturnDate
Output: Reservation
Sequence
Input: ArrivalDate, DepartDate, From, To
Output: ArrivalDate, BusTicket, Reservation
BusService
Input: ArrivalDate, To
Output: BusTicket
GenerateMapService
Input: To
Output: Reservation
FlightInformation
Input: DepartDate, From, ReturnDate, To
Output: ArrivalDate
Parallel
Input: ArrivalDate, ReturnDate, To
Output: BusTicket, Reservation
Mutation
Flights
HotelsService

**Fig. 8** Example of a candidate's mutation.

selecting a random node from the candidate tree, which is the root of a given subtree. This subtree is then replaced by a new subtree, generated using Algorithms 2 and 3 presented earlier. Since each node carries information about its required inputs and outputs, it is possible to create a substitute tree with a different structure but with equivalent functionality. If the selected node is the root for the candidate tree, an entirely new composition will be generated; if it is a leaf node, either a new service will be selected or a composition with equivalent functionality will be used. Alternatively, if for a given functionality requirement the services available in the repository can only produce a single suitable composition, then the currently selected subtree will be replaced with a copy of itself, effectively leaving the candidate unchanged. Figure 8 provides a visual representation of the mutation process. In this figure, the sequence node in the original candidate is chosen as the root for the subtree to be mutated. A new subtree is then generated, under the constraint that it must provide the same functionality as the original subtree. The newly generated subtree turns out to have the same overall structure as the original one, with a sequence root and a parallel inner node; however, two of its leaf nodes (services) are different from those included in the original subtree. This likely changes the overall QoS of the solution, while also ensuring that the modified candidate remains functionally correct.

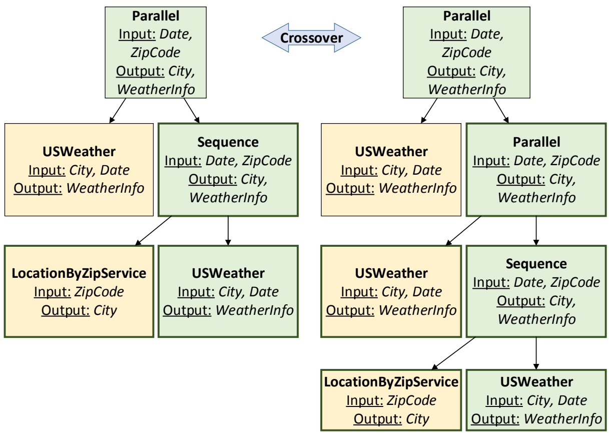


**Fig. 9** Example of crossover between two candidates.

The crossover operation also ensures functional correctness, this time by verifying that the components exchanged by two candidates are functionally equivalent. Given candidates $A$ and $B$, the first step is to select a depth $d$ within the bounds of both $A$ and $B$ trees. A node $n_A$ is then randomly selected and retrieved from depth $d$ of tree $A$, while the set of all nodes at depth $d$ of tree $B$ are retrieved as $nodes_B$. If there is a node $n_B$ in $nodes_B$ whose inputs and outputs are exactly the same as those of $n_A$, the subtrees represented by $n_A$ and $n_B$ are are swapped in candidates $A$ and $B$. On the other hand, if there is no node in $nodes_B$ that is functionally equivalent to $n_A$, then no crossover occurs. The choice of considering nodes at the same depth of the two parental trees was made in order to reduce the overall computation cost of performing a full crossover operation — where each node from tree $A$ is compared to each node from tree $B$ looking for a functional match, regardless of the tree depth, as done by Rodriguez-Mier et al. (2010). Figure 9 provides a visual representation of the crossover process between the right-hand side children of both root nodes. The figure shows two parents and their selected subtrees before the crossover has happened. In this case, the right-hand child of the root of each parent has been selected for the crossover, since these two children are functionally equivalent to each other. Once crossover occurs, two children will be generated by swapping the right-hand children of the two parents. Once again, this operation maintains the functional correctness of the children while likely modifying their QoS.

## 5 Third Approach: Constrained GP Composition with Choice Construct

As an extension to the second approach, the third approach discussed in this work also includes the choice construct as one of the composition possibilities, meaning that a candidate may have multiple execution paths. The inclusion of this construct is important because allowing the creation of multiple independent execution paths is a standard feature of service composition frameworks (Van der Aalst et al., 2003). In order to support the choice construct, the population initialisation algorithm presented in Subsection 4.2 has been modified to allow for the encoding of multiple execution paths. Whenever using a choice construct, the algorithm includes a service with multiple output possibilities in the composition. This way, the construct can choose a given execution path according to the output produced by the service with multiple outputs. For example, supposing that the service can produce two different subtypes of *BalanceCheck* — *PositiveBalanceCheck* and *NegativeBalanceCheck* —, then the choice construct could choose a certain execution path depending on whether the subtype of *BalanceCheck* is a *PositiveBalanceCheck*. An example of a candidate tree including a choice construct is shown in Figure 10.

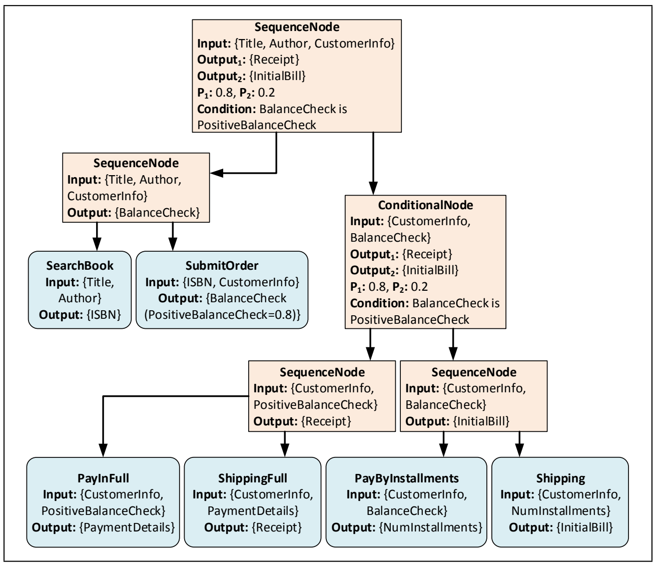

SequenceNode
Input: {Title, Author, CustomerInfo}
Output1: {Receipt}
Output2: {InitialBill}
P1: 0.8, P2: 0.2
Condition: BalanceCheck is PositiveBalanceCheck
SequenceNode
Input: {Title, Author, CustomerInfo}
Output: {BalanceCheck}
SearchBook
Input: {Title, Author}
Output: {ISBN}
SubmitOrder
Input: {ISBN, CustomerInfo}
Output: {BalanceCheck (PositiveBalanceCheck=0.8)}
ConditionalNode
Input: {CustomerInfo, BalanceCheck}
Output1: {Receipt}
Output2: {InitialBill}
P1: 0.8, P2: 0.2
Condition: BalanceCheck is PositiveBalanceCheck
SequenceNode
Input: {CustomerInfo, PositiveBalanceCheck}
Output: {Receipt}
SequenceNode
Input: {CustomerInfo, BalanceCheck}
Output: {InitialBill}
PayInFull
Input: {CustomerInfo, PositiveBalanceCheck}
Output: {PaymentDetails}
ShippingFull
Input: {CustomerInfo, PaymentDetails}
Output: {Receipt}
PayByInstallments
Input: {CustomerInfo, BalanceCheck}
Output: {NumInstallments}
Shipping
Input: {CustomerInfo, NumInstallments}
Output: {InitialBill}

**Fig. 10** Example of tree representation with conditional node for Web service composition.

### 5.1 Fitness Function

The fitness function used in this approach is exactly the same as the one used in the second approach, which measures the goodness of an individual as the weighted sum of normalised QoS values:

$$fitness(i) = f_i \tag{6}$$

### 5.2 Generation of Initial Candidates

When creating compositions with multiple execution paths, a different composition task needs to be used. This extended composition task contains a set of available inputs, a condition used for choosing the execution path (e.g. *BalanceCheck* is a *PositiveBalanceCheck*), and one set of outputs for each of the execution paths. This composition task is used by the initialisation algorithm to produce a candidate encoded as a graph, which is then transformed into a tree. The steps used when creating a composition candidate with multiple execution paths are shown in Algorithm 6, which uses the $condGraphToTree()$ function described in Algorithm 7. The input required by the algorithm consists of a set of required composition inputs $I$, two sets of outputs $O_1$ and $O_2$ for each execution path, a condition $C$ used for determining the execution path to follow, and a set of probabilities $P$ reflecting the likelihood of each path being executed. This algorithm's basic idea is to create a candidate by connecting independently created tree parts. Initially two sub-compositions in graph form, $G_1$ and $G_2$, are created to provide the functionality of the two distinct execution paths selected by the choice construct. These graphs are transformed into a tree format by employing Algorithm 3, yielding $T_1$ and $T_2$, which are then connected to a $ConditionalNode$ representing the choice construct with condition $C$. Next, another sub-composition $G_4$ is generated to connect from the provided composition inputs $I$ to the $ConditionalNode$ created earlier, and $G_4$ is translated into $T_4$. The probability $P$ is set in the $ConditionalNode$ using some of the outputs produced by $T_4$ to satisfy the check in condition $C$. A $SequenceNode$ is then set as the composition tree root and used to connect $T_4$, the initial part of the composition, to the part of the tree with multiple paths ($T_3$).

### 5.3 Mutation and Crossover

The mutation operator is applied to a randomly chosen subtree within the candidate tree (i.e. a tree node and

```
Procedure condGenerate()
    //Input  : $I$, $O_1$, $O_2$, $C$, $P$
    //Output: candidate tree $T$
    if $O_2$ is not empty then
        $G_1 \leftarrow$ createGraph($I \cup C.if, O_1$);
        $G_2 \leftarrow$ createGraph($I \cup C.else, O_2$);
        $T_1 \leftarrow$ condGraphToTree($G_1.start$);
        $T_2 \leftarrow$ condGraphToTree($G_2.start$);
        $T_3 \leftarrow$ new $ConditionalNode(C)$;
        $T_3.leftChild \leftarrow T_1$;
        $T_3.rightChild \leftarrow T_2$;
        if $C$ fulfilled by $I$ then
            $T_3.probability \leftarrow P$;
            return $T_3$;
        else
            $G_4 \leftarrow$ createGraph($I, C.else$);
            $T_4 \leftarrow$ condGraphToTree($G_4.start$);
            $T_3.probability \leftarrow T_4.end.P$;
            $T \leftarrow$ new $SequenceNode()$;
            $T.leftChild \leftarrow T_4$;
            $T.rightChild \leftarrow T_3$;
            return $T$;
        end
    else
        $G \leftarrow$ createGraph($I, O_1$);
        $T \leftarrow$ condGraphToTree($G.start$);
        return $T$;
    end
```

**Algorithm 6:** Generating a new candidate tree or a mutated subtree with conditional (choice) construct.

```
Procedure condGraphToTree()
    //Input  : $N$
    //Output: tree $T$
    if $N$ has no service successor then
        return $N$;
    else if $N = start$ then
        if |$N$ has one successor then
            $T \leftarrow$
            condGraphToTree($successor(N)$);
        else
            $T \leftarrow$
            createParallelNode($successors(N)$);
        end
    else
        Remove $end$ from $N$'s successors (if
        present);
        if $N$ has one successor then
            $rightChild \leftarrow$
            condGraphToTree($successor(N)$);
        else
            $rightChild \leftarrow$
            createParallelNode($successors(N)$);
        end
        $T \leftarrow$ new $SequenceNode()$;
        $T.leftChild \leftarrow N$;
        $T.rightChild \leftarrow rightChild$;
    return $T$;
Procedure createParallelNode()
    //Input  : $children$
    //Output: tree $T$
    $T \leftarrow$ new $ParallelNode()$;
    foreach $c$ in $children$ do
        $subtree \leftarrow$ condGraphToTree($c$);
        Add $subtree$ to $T$'s children;
    end
    return $T$;
```

**Algorithm 7:** Converting a graph into tree representation with conditional (choice) construct.

its children), and it replaces the original subtree with a new one with the same functionality but a different structure. This is done by executing Algorithm 6, ensuring that the newly created subtree has the same inputs and outputs as the original subtree (node) selected. The crossover operator exchanges two service nodes between the two chosen individuals, provided that these two swapped leaves have equivalent functionality (i.e. produce equivalent outputs given equivalent inputs). This exploration of functionally equivalent alternatives for a task models the process of Web service selection during the composition process.

## 6 Experiment Design

Two sets of experiments have been conducted in this work in order to measure the performance of the different GP approaches discussed in previous sections. The first set of experiments compared the best fitness and execution times of the first and second GP approaches (referred to as GP-I and GP-II, respectively, in this section) against two PSO benchmark algorithms, further discussed below. The objective was to determine whether the proposed GP approaches present some advantages in comparison to the benchmark algorithms. GP-III was not included in this first comparison because it requires composition tasks with multiple execution paths, whereas the other algorithms cannot handle multiple paths (a second set of experiments for GP-III is discussed later on). The datasets used for the first set of experiments were generated using the QWS dataset (Al-Masri and Mahmoud, 2007) as its basis, since currently no benchmark datasets are available for evaluating QoS-aware web service composition (Yu et al., 2013). The six datasets contain information that has been collected online detailing the inputs, outputs, time, cost, reliability, and availability of real Web services. Four different composition tasks were used throughout this set of experiments, requiring the creation of composition solutions of various sizes and complexities. Their details are displayed in Table 1.

**Table 1** Experiment tasks.

| Task | Inputs | Outputs | Dataset (No. Services) |
|---|---|---|---|
| 1 | PhoneNumber | Address | 1(20) |
| 2 | ZipCode, Date | City, WeatherInfo | 2(30) |
| 3 | From, To, DepartDate, ReturnDate | ArrivalDate, Reservation | 3(60) |
| 4 | From, To, DepartDate, ReturnDate | ArrivalDate, Reservation, BusTicket, Map | 4(150), 5(450), 6(4500) |

The first set of experiments was conducted on a personal computer with a 3.4 GHz CPU and 8 GB RAM. For all approaches, 50 independent runs were executed per dataset. The PSO parameter settings used in this set of experiments were the same as those outlined in the original work (da Silva et al., 2014), which in turn were based on the existing PSO literature (Shi and Eberhart, 1998; Kennedy et al., 2001). Likewise, the choice of GP parameters was generally based on the work of Koza (1992), though with varying population sizes. Larger populations were used when previous attempts with smaller sets of candidates failed to converge; smaller populations were chosen when previous attempts with larger sets of candidates did not yield significant optimisation improvements, thus only increasing the optimisation time. It must be noted that the successful use of smaller populations for evolution-based techniques is not uncommon in the Web service composition literature (Cao et al., 2007; Gao et al., 2007).

For GP-I, a population size of 1000 was used — smaller populations were previously attempted with unsatisfactory convergence rates, and a population size of 30 was used for GP-II — larger populations were previously attempted without significant gains to the best solution fitness. For GP-I, each run was required to continue until a fully functional result was achieved, at which point 50 more iterations would occur and the run would finish. For GP-II, runs lasted 100 generations. For both GP-I and GP-II, the crossover and mutation probabilities were set to 0.9 and 0.1, respectively, the maximum depth for solution trees was set to 3, and the single best solution in one generation was copied to the next. For GP-I, the fitness function was configured with weights of 0.25 for all QoS properties, and of 0.5 for both $w_5$ and $w_6$. For GP-II, the fitness function was configured with weights of 0.25 for all QoS properties. Since all candidates are required to represent functionally correct compositions, the size of the trees is not constrained or monitored during the run. As the fitness function rewards smaller candidates, bloating has not been found to be an issue during execution. For both PSO approaches, a swarm of 30 particles was used and runs were allowed to execute for a maximum of 100 iterations, but were terminated earlier if the best fitness remained the same throughout 10 iterations. The fitness function was configured with weights of 0.25 for all QoS properties, the PSO inertia weight $w$ was set to 1, and acceleration constants $c_1$ and $c_2$ were both set to 1. The greedy-based PSO approach was configured to choose the initial composition workflow from 50 randomly generated candidates.

The second set of experiments measured the performance of GP-III (the third GP variation proposed), an approach that allows the creation of compositions with multiple execution paths. The objective was to determine whether the creation of compositions with multiple execution paths is feasible using an extension of GP-II. To the best of our knowledge, no other fully automated composition approaches that allow for the optimisation of solutions containing multiple execution paths have been proposed, therefore GP-III was tested by comparing it to GP-II. More specifically, each solution with two paths — if and else — produced by GP-III was compared to two solutions produced independently by GP-II — one solution providing the functionality of the if-path, and the other of the else-path. A direct comparison and significance analysis between these two approaches is not possible, but an indirect comparison will hopefully provide interesting patterns that lead to insights. Because of this, comparisons to the PSO approaches were not performed in this set of experiments, and instead the focus was on comparing the two more similar GP techniques. GP-III was tested using new datasets that allow for the creation of compositions with multiple execution paths, which were generated based on those presented in Wang et al. (2013). However, the new datasets contain composition tasks that require multiple execution paths, as well as services that are capable of producing multiple output subtypes at runtime given a certain (randomly generated) probability. The multiple subtypes produced by these services can then be used by a choice construct that decides which path to be executed. Once again, experiments were conducted on a personal computer with a 3.4 GHz CPU and 8 GB RAM. 50 independent runs were executed per dataset, both for the conditional tasks (i.e. those using the third approach) and the non-conditional tasks (i.e. those using the second approach, creating independent if and else execution paths). Each run was executed with a population of size 20, and runs lasted for 50 generations. The crossover and mutation probabilities were set to 0.9 and 0.1, respectively, and the single best solution in one generation was copied to the next. Limits for the minimum and maximum solution trees were not set, since candidates are already constrained by functional correctness checks during the evolutionary process. The fitness function was configured with weights of 0.25 for all QoS properties.

### 6.1 Two PSO Benchmark Algorithms

Before discussing experiment results it is necessary to introduce two PSO benchmark algorithms that were employed when carrying out comparisons against the proposed GP approaches. These algorithms were chosen because they are representatives of an alternative EC-based strategy to fully automated Web service composition. For reasons of brevity, only the key characteristics of each PSO approach will be described here, however their full explanation can be found in the original work from which they were reproduced (da Silva et al., 2014). For both of these approaches, the fitness function employed in the evolutionary process is the unchanged objective function presented in Subsection 2.2 (Equation 1) — exactly the same as what was employed in the GP approaches.

#### *6.1.1 Greedy-Based PSO Approach*

The greedy-based PSO approach (da Silva et al., 2014) uses a greedy algorithm to generate an initial Web service composition workflow where services can be executed sequentially, in parallel, or in a combination thereof. This workflow contains abstract slots for placing services, each slot presenting a different set of available inputs and required outputs. For each slot, a list of compatible services is compiled. PSO is then employed to select the best possible service for each slot in order to arrive at a solution with the best possible QoS attributes overall. Each particle is represented as having $n$ dimensions, where $n$ corresponds to the number of abstract slots in the workflow, and each dimension points to a Web service from its list of compatible services. In summary, in greedy-based PSO the structure of the composition is determined first, and the services to populate that structure are selected afterwards.

#### *6.1.2 Graph-Based PSO Approach*

The graph-based PSO approach (da Silva et al., 2014) also employs the greedy composition algorithm, but this time during the evolutionary process. Initially, the discovery of all services from the repository that could possibly be used for the requested composition task is performed using a basic algorithm. Once the discovery is finished, a directed graph showing all the input-output relationships between these services is created — this is referred to as the *master graph*. The services in the master graph are represented as nodes, and the relationships between them as edges. Each particle has $k$ dimensions, where $k$ corresponds to the number of edges in the master graph. Each dimension holds a value between 0 and 1, which represents a weight associated with that edge. Since each particle only contains a series of weights, during PSO it is necessary to extract the candidate composition workflow from the master graph using the greedy algorithm in order to interpret the solution it represents. The extraction algorithm is run aided by the weights in the particle, meaning that edges with the highest weights are selected to be in the candidate composition. After the workflow has been extracted its fitness can finally be calculated. In summary, in graph-based PSO both the structure of the composition and the services that populate it are selected simultaneously.

## 7 Results and Discussions

### 7.1 Results of First Set of Experiments

The results of the first set of experiments are shown in tables 2 and 3; table 2 shows the mean fitness values achieved by all approaches, and table 3 shows their average execution times. In both tables, the first column records the dataset used and its total number of services, the second column contains the composition task employed, and the third column shows the minimum number of services from that dataset which had to be used in order to create a fully functional solution for the composition task. The fourth, fifth, sixth and seventh columns present the fitness/time of the greedy-based, graph-based, GP-I, and GP-II approaches, respectively; the eighth, ninth, tenth, and eleventh columns show the results of Wilcoxon signed-rank tests at .95 confidence interval, carried out to verify whether there is any statistically significant time/fitness differences. These differences are indicated by + for significant larger values, and − for significant smaller values. $T_1$ represents the comparison between GP-I and greedy-based, $T_2$ between GP-I and graph-based, $T_3$ between GP-II and greedy-based, and $T_4$ between GP-II and graph-based. Note that time results include setup times associated with service discovery, creation of the master graph, etc.

The results show that the graph-based PSO approach has a clearly better execution time than GP-I, and clearly worse than that of the greedy-based PSO approach. The average fitness, on the other hand, suggests that overall performance of the greedy-based and graph-based approaches is equivalent, and the fitness of the GP-I approach becomes progressively inferior with

**Table 2** Mean fitness results for each approach, using first and second GP approaches.

| Dataset (No. Servs.) | Task | Min. Comp. Size | Fitness | | | | Significance Testing | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Greedy PSO | Graph PSO | GP-I | GP-II | $T_1$ | $T_2$ | $T_3$ | $T_4$ |
| 1(20) | 1 | 1 | 0.808±0 | 0.808±0 | 0.808±0 | 0.808±0 | | | | |
| 2(30) | 2 | 2 | 0.713±0 | 0.713±0 | 0.639±0.04 | 0.713±0 | - | - | | |
| 3(60) | 3 | 2 | 0.634±0 | 0.631±0.011 | 0.634±0 | 0.634±0 | | + | | |
| 4(150) | 4 | 4 | 0.532±0 | 0.524±0.01 | 0.413±0.06 | 0.527±0.01 | - | - | - | |
| 5(450) | 4 | 4 | 0.532±0 | 0.525±0.01 | – | 0.526±0.01 | | | - | |
| 6(4500) | 4 | 4 | 0.568±0.010 | 0.637±0.022 | – | 0.617±0.02 | | | + | |

**Table 3** Mean time results for each approach, using first and second GP approaches.

| Dataset (No. Servs.) | Task | Min. Comp. Size | Time (ms) | | | | Significance Testing | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | Greedy PSO | Graph PSO | GP-I | GP-II | $T_1$ | $T_2$ | $T_3$ | $T_4$ |
| 1(20) | 1 | 1 | 22.9±1.2 | 41.3±10 | 149.6±58.3 | 62.2±81.4 | + | + | + | + |
| 2(30) | 2 | 2 | 9±0.1 | 13.8±2.8 | 346±282 | 193.5±13.5 | + | + | + | + |
| 3(60) | 3 | 2 | 11±0 | 87.2±18 | 180.6±68.6 | 187.1±11.2 | + | + | + | + |
| 4(150) | 4 | 4 | 21.7±0.5 | 116.1±24.5 | 67689.7± 109320.9 | 340.8±36.5 | + | + | + | + |
| 5(450) | 4 | 4 | 33.6±1 | 60.4±2.3 | – | 351.3±32.5 | | | + | + |
| 6(4500) | 4 | 4 | 462.4±61.2 | 752.3±78.6 | – | 634.8±51.4 | | | + | - |

the growth of dataset sizes. As it can be observed, the fitness and time values for the execution of GP-I using datasets 5 and 6 are missing from the table. This is because the runs using those two datasets failed to converge after a significant amount of time. In fact, the efficiency of GP-I is severely reduced for dataset 4, as seen by the sudden spike in the execution time and drop in the fitness value. These results indicate that GP-I is not a scalable approach to service composition.

For GP-II, the results show that the fitness for all datasets is mostly equivalent to those of the PSO approaches, with small variations for datasets 4 and 5, but differences are more pronounced in dataset 6. Likewise, the execution time for each approach mostly follows a consistent pattern as the number of services in the dataset increases, with GP-II taking the longest time to execute, but this pattern is once again broken in dataset 6. Interestingly, the execution time required by the graph-based PSO approach undergoes roughly a fifteen-fold increase as the number of services handled grows from 450 to 4500 (dataset 5 to dataset 6), meanwhile the GP-II approach undergoes less than a twofold increase in the same scenario. Despite requiring a longer execution time for dataset 6, the graph-based PSO approach also produces results with a significantly higher fitness value. This seems to indicate that there is a trade-off between fitness and execution time for larger datasets, an observation that was also made the original paper presenting the graph-based PSO approach (da Silva et al., 2014).

These results reveal two interesting trends. Firstly, it becomes apparent that restricting the search space to only include fully functionally correct solutions, as done by the graph-based PSO approach and the GP-II approach, leads to composition solutions that have a better overall quality than those from a non-restricted search space. Secondly, results indicate that representing solutions directly as trees, where the final composition workflow is easily deducible from a candidate's structure, allows techniques to scale better than when using the indirect representation employed by the graph-based PSO. That is because the graph-based PSO approach begins by creating a large graph that shows all possible service dependencies, so the number of edges in this graph quickly grows as the service repository increases, which translates to larger particles and consequently larger processing times. Another interesting pattern is in the standard deviations associated with the fitness means shown: many of these standard deviations have a value of 0, indicating that the solutions found for each run had the exact same quality values. Further investigation shows that this was the case in situations where the search space is restricted, either when using smaller datasets (those with 60 services or less) or when using the greedy PSO approach (which restricts the topology allowed for a solution). However, even in those cases the mean fitness increases until eventual convergence throughout the run, indicating that the evolutionary optimisation approaches are indeed performing as expected.

## 7.2 Results of Second Set of Experiments

The results of the second set of experiments are shown in Table 4, which shows figures produced when generating compositions with multiple execution paths, and Table 5, which shows figures produced when generating two composition results that independently represent if-paths and else-paths. In Table 4, the first column in-

**Table 4** Mean time and fitness results when creating solutions with multiple execution paths, using GP-III.

| | Conditional | |
|---|---|---|
| **Set (size)** | **Avg. fitness** | **Avg. time (s)** |
| 1 (158) | 0.601 ± 0.013 | 1.290 ± 0.100 |
| 2 (558) | 0.712 ± 0.009 | 2.829 ± 0.250 |
| 3 (604) | 0.631 ± 0.008 | 13.285 ± 1.229 |
| 4 (1041) | 0.718 ± 0.048 | 6.146 ± 0.574 |
| 5 (1090) | 0.698 ± 0.005 | 11.759 ± 0.948 |
| 6 (2198) | 0.662 ± 0.017 | 92.392 ± 11.353 |
| 7 (4113) | 0.578 ± 0.010 | 97.344 ± 13.705 |
| 8 (8119) | 0.656 ± 0.005 | 326.387 ± 37.659 |

**Table 5** Mean time and fitness results when creating each execution path separately, using GP-II.

| | Non-conditional | | | |
|---|---|---|---|---|
| | If branch | | Else branch | |
| **Set (size)** | **Avg. fitness** | **Avg. time (s)** | **Avg. fitness** | **Avg. time (s)** |
| 1 (158) | 0.508 ± 0.000 | 0.563 ± 0.144 | 0.588 ± 0.037 | 0.718 ± 0.079 |
| 2 (558) | 0.588 ± 0.084 | 1.490 ± 0.526 | 0.694 ± 0.016 | 1.527 ± 0.194 |
| 3 (604) | 0.365 ± 0.000 | 4.387 ± 0.768 | 0.788 ± 0.000 | 7.099 ± 0.906 |
| 4 (1041) | 0.689 ± 0.064 | 4.510 ± 1.177 | 0.741 ± 0.429 | 3.568 ± 0.429 |
| 5 (1090) | 0.446 ± 0.000 | 5.726 ± 0.755 | 0.688 ± 0.011 | 6.491 ± 0.743 |
| 6 (2198) | 0.412 ± 0.063 | 58.295 ± 12.765 | 0.645 ± 0.024 | 52.308 ± 5.785 |
| 7 (4113) | 0.363 ± 0.000 | 44.838 ± 5.926 | 0.688 ± 0.032 | 51.725 ± 4.260 |
| 8 (8119) | 0.474 ± 0.000 | 106.119 ± 7.152 | 0.766 ± 0.002 | 186.896 ± 20.008 |

dicates the dataset used, the second column shows the mean fitness of the best solution obtained in each run, and the third column provides the mean execution time for the third approach. Table 5 follows the same layout as Table 4, but it shows results for the if-branches and else-branches separately.

Two patterns emerge by analysing the results. Firstly, the fitness of the solution with multiple execution paths is approximately equal to the average of the fitness of the if and else branches independently produced. Even though the fitness of the two approaches cannot be directly compared, this pattern potentially indicates that the structure of the solutions produced using GP-III is equivalent to those produced using GP-II. Secondly, the execution time of the GP-III is approximately equivalent to the sum of the execution times for the individual paths created using GP-II, which shows that creating a composition representation that contains a choice construct and multiple execution paths is equally as efficient as generating each execution path separately. These two patterns demonstrate that compositions requiring multiple execution paths can be created by using a GP approach with a choice construct, producing results of comparable quality and in an equivalent amount of time to approaches that do not include such construct.

In addition to ensuring that the connections between services are correct, the inclusion of the choice construct into the compositions generated by GP-III means that it is also necessary to prevent a candidate from being modified in a way that violates the overall flow of the candidate. Namely, if a composition is expected to generate two alternative groups of outputs, then exactly one choice construct should be present, as this results in the creation of two alternative execution paths. Likewise, if three alternative groups of outputs were requested, then two choice constructs would need to be present, one as a child of the other to create a total of three paths. These path constraints add a new layer of complexity to the composition problem, which must be handled independently from the service input/output constraints. This experiment shows that it is possible to handle these restrictions for a simple case with two sets of outputs. In principle we can decompose the tasks so that our proposed GP-III can be used to handle more complex scenarios (i.e. composition tasks that require many alternative sets of outputs).

## 7.3 Summary of Evaluation

Results show that the use of a random population initialisation and of random mutation and crossover operators did not yield the best results, as evidenced by the first set of experiments. However, by changing the population initialisation and operators to enforce functional correctness, and by focusing the fitness function on quality optimisation only, definite improvements are observed. The use of constrained initialisation and operators allows GP to reach equivalent quality levels to PSO solutions, and scale better as the datasets employed in the comparison grow. Most importantly, the constrained initialisation and genetic operations are robust enough to allow the inclusion of conditional constructs into compositions, supporting the creation of solutions with multiple execution paths.

The evaluation results discussed in the previous subsections highlight the differences in behaviour between the GP techniques when addressing the tension between effective searching and constraint handling. On the one hand, GP-I has a very flexible behaviour that allows candidates to be randomly modified into a vast array of configurations; on the other hand, GP-II and GP-III's behaviour can be quite restrictive, as the mutation and crossover operators must can be performed if their modifications preserve the functional correctness of candidates. As it has been shown, the flexibility afforded by GP-I leads to difficulties in identifying feasible solutions (since the unconstrained space it searches is so large), whereas the behaviour of GP-II and GP-III eliminates the issue of infeasibility. Despite this advantage, the restrictiveness of GP-II and GP-III has the drawback that certain operation attempts may fail to produce offspring that is different from its parents: in the case of mutation, the chosen subtree may be the only structure that provides the required functionality; in the case of crossover, the two parents may not have any interchangeable nodes. Thus, the composition problem remains open to the investigation of new techniques.

## 8 Conclusions

This paper proposed three different GP approaches for QoS-aware Web service composition: the first approach (GP-I) uses fitness function penalisations to encourage the correctness of composition solutions; the second approach (GP-II) enforces the correctness of solutions by restricting the population initialisation and genetic operators used; and the third approach (GP-III) extends the second approach by enabling the creation of solutions that include choice constructs. Experiment results show that the second approach produces compositions with better quality than those produced by the first approach. When compared to the PSO techniques, the second approach seems to present a trade-off between execution time and fitness: while the fitness of the GP approach is lower than those of the graph-based PSO, its execution time is lower as well. Finally, it is shown that the third approach manages to include the choice construct as a composition possibility without diminishing the quality of compositions or requiring longer execution times. This leads to the conclusion that GP approaches can be scalable to the problem of automated Web service composition, as well as flexible enough to allow more complex composition constructs to be considered. The initialisation algorithm utilised by the second and third approaches creates solutions by translating compositions in graph format to trees, which incurs additional costs. Therefore, future works should investigate the possibility of evolving compositions directly as graphs, which would likely further improve the performance of this composition approach.

## References

Van der Aalst WM, Dumas M, ter Hofstede AH (2003) Web service composition languages: Old wine in new bottles? In: Proceedings of the 29th Euromicro Conference, IEEE, pp 298–305

Al-Masri E, Mahmoud QH (2007) QoS-based discovery and ranking of web services. In: Proceedings of 16th International Conference on Computer Communications and Networks, IEEE, pp 529–534

Alrifai M, Risse T (2009) Combining global optimization with local selection for efficient QoS-aware service composition. In: Proceedings of the 18th International Conference on World Wide Web, ACM, pp 881–890

Amiri MA, Serajzadeh H (2012) Effective web service composition using particle swarm optimization algorithm. In: Proceedings of the 6th International Symposium on Telecommunications (IST), IEEE, pp 1190–1194

Aversano L, Di Penta M, Taneja K (2006) A genetic programming approach to support the design of service compositions. International Journal of Computer Systems Science & Engineering 21(4):247–254

Cao L, Li M, Cao J (2007) Using genetic algorithm to implement cost-driven web service selection. Multiagent and Grid Systems 3(1):9–17

Dupuis JF, Fan Z, Goodman ED (2012) Evolutionary design of both topologies and parameters of a hybrid dynamical system. Evolutionary Computation, IEEE Transactions on 16(3):391–405

Gao C, Cai M, Chen H (2007) QoS-aware service composition based on tree-coded genetic algorithm. In: Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International, IEEE, vol 1, pp 361–367

Jaeger MC, Mühl G (2007) QoS-based selection of services: The implementation of a genetic algorithm. In: ITG-GI Conference on Communication in Distributed Systems (KiVS), VDE, pp 1–12

Kennedy J, Kennedy JF, Eberhart RC, Shi Y (2001) Swarm intelligence. Morgan Kaufmann

Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection, vol 1. MIT press

Ludwig SA (2012) Applying particle swarm optimization to quality-of-service-driven web service composition. In: Proceedings of the 26th International Conference on Advanced Information Networking and Applications (AINA), IEEE, pp 613–620

Menascé DA (2002) QoS issues in web services. Internet Computing 6(6):72–75

Milanovic N, Malek M (2004) Current solutions for web service composition. Internet Computing 8(6):51–59

Mucientes M, Lama M, Couto MI (2009) A genetic programming-based algorithm for composing web services. In: Proceedings of the 9th International Conference on Intelligent Systems Design and Applications, IEEE, pp 379–384

Potthof A, Seibert S, Thomas W (1994) Nondeterminism versus determinism of finite automata over directed acyclic graphs. Bulletin of the Belgian Mathematical Society Simon Stevin 1(2):285

Rao J, Su X (2005) A survey of automated web service composition methods. In: Semantic Web Services and Web Process Composition, Springer, pp 43–54

Rezaie H, NematBaksh N, Mardukhi F (2010) A multi-objective particle swarm optimization for web service composition. In: Networked Digital Technologies, Springer, pp 112–122

Rodriguez-Mier P, Mucientes M, Lama M, Couto MI (2010) Composition of web services through genetic programming. Evolutionary Intelligence 3(3-4):171–186

Shi Y, Eberhart RC (1998) Parameter selection in particle swarm optimization. In: Evolutionary programming VII, Springer, pp 591–600

da Silva AS, Ma H, Zhang M (2014) A graph-based particle swarm optimisation approach to QoS-aware web service composition. In: Congress on Evolutionary Computation (CEC), IEEE

da Silva AS, Ma H, Zhang M (2015) A GP approach to QoS-aware web service composition including conditional constraints. In: Congress on Evolutionary Computation (CEC), IEEE, pp 2113–2120

Srivastava B, Koehler J (2003) Web service composition — current solutions and open problems. In: ICAPS Workshop on Planning for Web Services, vol 35, pp 28–35

Wang A, Ma H, Zhang M (2013) Genetic programming with greedy search for web service composition. In: Database and Expert Systems Applications, Lecture Notes in Computer Science, vol 8056, Springer Berlin Heidelberg, pp 9–17

Xia H, Chen Y, Li Z, Gao H, Chen Y (2009) Web service selection algorithm based on particle swarm optimization. In: Proceedings of the 8th International Conference on Dependable, Autonomic and Secure Computing, IEEE, pp 467–472

Xiao L, Chang CK, Yang HI, Lu KS, Jiang Hy (2012) Automated web service composition using genetic programming. In: Proceedings of the 36th Annual Computer Software and Applications Conference Workshops (COMPSACW), IEEE, pp 7–12

Yu Y, Ma H, Zhang M (2013) An adaptive genetic programming approach to QoS-aware web services composition. In: Congress on Evolutionary Computation (CEC), IEEE, pp 1740–1747