A GP Approach to QoS-Aware Web Service Composition including Conditional Constraints

Alexandre Sawczuk da Silva, Hui Ma, Mengjie Zhang
School of Engineering and Computer Science, Victoria University of Wellington, New Zealand
Email: Alexandre.Sawczuk.da.Silva@ecs.vuw.ac.nz | Hui.Ma@ecs.vuw.ac.nz | Mengjie.Zhang@ecs.vuw.ac.nz

***Abstract*—Automated Web service composition is one of the holy grails of service-oriented computing, since it allows users to create an application simply by specifying the inputs the resulting application should require, the outputs it should produce, and any constraints it should respect. The composition problem has been handled using a variety of techniques, from AI planning to optimisation algorithms, however no approach so far has focused on handling three composition dimensions simultaneously, producing solutions that are: (1) fully functional (i.e. fully executable), (2) respect conditional constraints (e.g. user can specify logical branching), and (3) are optimised according to non-functional Quality of Service (QoS) measurements. This paper presents a genetic programming approach that addresses these three dimensions simultaneously through the fitness function, as well as through the enforcement of constraints to candidate trees during initialisation, mutation, and crossover. The approach is tested using an extended version of the WSC2008 datasets, and results show that fully functional and quality-optimised solutions can be created for all associated tasks, with an execution time that is roughly equivalent to that of a non-conditional approach.**

I. Introduction

As applications increasingly interact with the Web, the concept of Service-Oriented Architecture (SOA) [1] emerges as a popular solution. The key components of SOA are often Web services, which are functionality modules that provide operations accessible over the network via a standard communication protocol [2]. One of the greatest strengths of Web services is their modularity, because it allows the reuse of independent services that provide a desired operation as opposed to having to re-implement that functionality. The combination of multiple modular Web services to achieve a single, more complex task is known as *Web service composition*, and developing a system capable of creating such compositions in a fully automated manner is one of the holy grails of the field [3].

The complexity of Web service composition lies in the number of distinct dimensions it must simultaneously account for. On the first dimension, services must be combined so that their operation inputs and outputs are properly linked, i.e. the output produced by a given service is usable as input by the next services in the composition, eventually leading up to the desired overall output. On the second dimension, the composition must meet any specified user constraint or preference. A constraint is defined as a user restriction that must be met in order for a composition solution to be considered valid, and this mainly concerns execution flow features (e.g. the composition must have multiple execution options – branches – according to a given condition) [4], [5]. A preference, on the other hand, is a user restriction that would be desirable to observe, even though a composition solution is still considered valid if this restriction is not met (e.g. between two services with similar functionality, a user would always prioritise the use of one service over the other in a composition) [4]. It must be noted that constraints relating to Quality of Service (QoS) values are not included in this dimension. On the third dimension, the resulting composition must achieve the best possible overall Quality of Service (QoS) with regards to attributes such as the time required to execute the composite services, the financial cost of utilising the service modules, and the reliability of those modules.

978-1-4799-7492-4/15/$31.00

Several techniques have been proposed to address the composition problem, such as variations of AI planning [6], Evolutionary Computation (EC) techniques [7], and hybrid optimisation algorithms [8]. These approaches produce promising results, however the great majority of them only account for two composition dimensions at once. For example, AI planning techniques for composition focus on guaranteeing functional correctness (first dimension) and fulfilment of constraints (second dimension), while EC techniques such as Genetic Algorithms (GA) and Genetic Programming (GP) focus on QoS (third dimension) in addition to functional correctness (first dimension) but do not include conditional branches (second dimension).

Thus, the objective of this work is to propose a Web service composition approach that simultaneously considers elements from all the three dimensions described above when generating solutions. This approach employs Genetic Programming (GP) for evolving a population of near-optimised solutions, at the same time restricting the structure of candidates according to functional correctness and user constraints. Specifically, each dimension is addressed as follows:

- **First dimension (functional correctness):** The solutions are represented as trees where the way in which services are linked to each other is restricted to preserve functionality (more details in Subsection III-A).
- **Second dimension (user constraints):** A branching conditional constraint is included as one of the possible nodes in the tree representation of solutions, thus also enabling the enforcement of user constraints (see Subsection II-B).
- **Third dimension (Quality of Service):** A fitness function is used to optimise candidate solutions with regards to their overall QoS attributes (see Subsection III-C).

Knowledge of GP [9] is assumed, and the remainder of this paper is organised as follows: section II provides background information on the area of Web service composition; section

III presents the proposed GP approach; section IV discusses experiments performed on the proposed approach; section V discusses experiment results; section VI concludes the paper.

II. Background

A. Problem Description

The objective of Web service composition is to create a new, composite service that accomplishes a given task. For example, consider the scenario of an online book shopping system, adapted from [4]. In this scenario, the objective is to employ existing Web services to accomplish a basic book shopping operation. Preferably, the services to be used, the order in which they are to be invoked, and how they interact to one another should be determined automatically. Therefore, the book and customer details (e.g. title, author, customer information) and the expected purchase outcome (e.g. receipt) act as the composition task inputs and outputs, and the shopping-related services as the atomic composition components.

In certain cases, however, the customer may have specific constraints. For example, the customer's preferred method of payment is likely to depend on his/her current account balance: if the customer has enough money to pay for the book in full, then he/she would like to do so, otherwise the customer would like to pay by installments. In this case, the composition task has one set of inputs (book and customer details), and a condition (balance) that may lead to two different sets of outputs depending on whether it is met (either a receipt for paying full or the initial installment bill). In this type of composition, the runtime value of the type in the condition is used to ultimately decide which set of outputs should be produced. Figure 1 provides a visual representation of the example described above. Therefore, a technique capable of producing compositions with conditional execution branches becomes necessary.

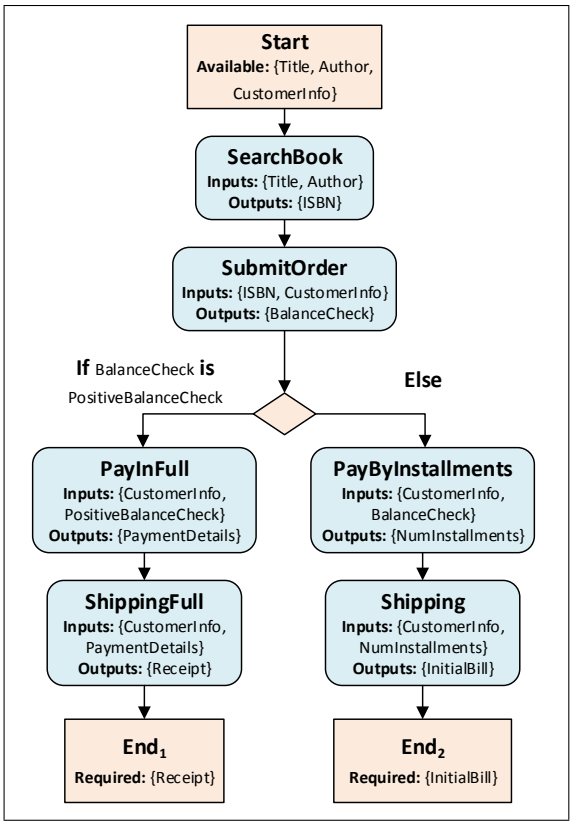

Fig. 1. Example of composition requiring a branching constraint.

B. Languages

Languages for Web service composition, such as BPEL4WS [10], offer constructs (e.g. sequential, parallel, choice) to describe how services interact to each other when assembled into a composite service in terms of input satisfaction and output production. However, in addition to these functional aspects, the non-functional attributes of each service (i.e. their quality) must also be considered when producing a composition. In this work, four QoS measures are taken into account [11]: Time (T), which indicates the execution time of a service between sending a request and receiving a response; Cost (C), which is the financial cost associated with utilising the service; Availability (A), which is the probability of a service being available when requested for execution, and Reliability (R), which is the probability of a service responding appropriately when receiving a request. These attributes are associated with individual services, but can be calculated for a service composition according to the constructs used in it. The following constructs are employed in the proposed representation:

- *Sequence construct:* In a sequence construct services are executed sequentially, with the outputs of a service feeding the inputs of the next. This is depicted in Figure 2. When calculating the overall QoS attributes for a sequence construct, the availability and probability values of all atomic services are multiplied, while the time and cost are added.
- *Parallel construct:* In a parallel construct, services are executed in parallel with their inputs being independently satisfied and their outputs independently produced. This construct is shown in Figure 3. The QoS attributes are calculated just as they are for the sequence construct, except for the time, which is considered to be that of the atomic service with the longest execution time.
- *Choice construct:* In a choice (conditional) construct, only one of m services will be executed depending on whether a given value condition is met. In this construct, the produced output is one of m possible sets, as shown in Figure 4. The likelihood of producing each of the m sets is represented as a probability value, with the probabilities for the sets adding up to 1. Therefore, the QoS attributes for the conditional construct are calculated using the weighted sum of each quality attribute over all possible services, using the probabilities as the weights.

C. Related Work

1) AI Planning: AI planning approaches to Web service composition ensure functional correctness by building a solution graph step by step. This solution graph may either be built to enforce a set of user constraints, or it may be used to find an optimal solution in terms of QoS. However, no planning approaches in the literature have attempted to accomplish both of these objectives simultaneously, which is a key difference from the work proposed in this paper. The approaches discussed below are representative of these two

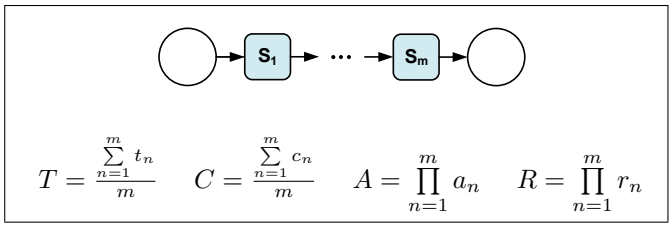

Fig. 2. Sequence construct and calculation of its QoS properties [12].

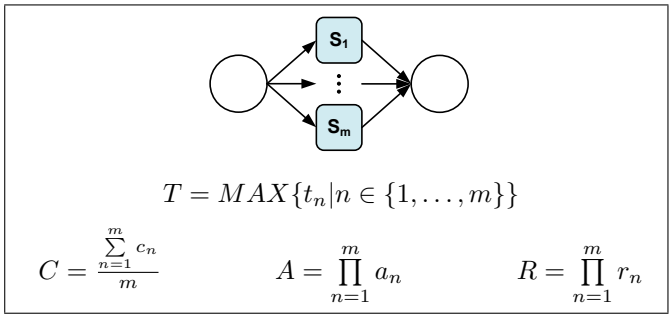

Fig. 3. Parallel construct and calculation of its QoS properties [12].

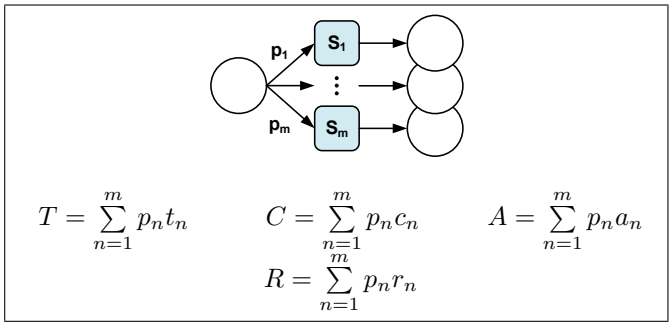

Fig. 4. Choice construct and calculation of its QoS properties.

distinct research paths. For the constraints direction, it soon becomes evident that benchmarks to measure the effectiveness of solutions have not yet been established by researchers. This means that experiments in this area focus on demonstrating that approaches are capable of achieving compositions with that fulfil the required constraints, while comparisons between distinct approaches that consider branching are lacking.

The work in [5] presents an approach to include user preferences into the process of Web service composition. This is accomplished by relying on a framework written in Golog, a language created for agent programming. Golog is used to specify the particular attributes of generic workflows that represent commonly requested composition procedures (an example of a generic workflow would be one that is dedicated to booking inter-city transportation). The syntax of a logic-based language used to specify user preferences is described, allowing for branching according to conditions, and for expressing preferences over alternative services. Despite supporting branching, only one set of final outputs is allowed, meaning that the branches must be merged before reaching the end node of the composition workflow.

In [6], authors combine a planning algorithm and a graph search algorithm to achieve both QoS optimisation and functional correctness in Web service compositions. The Generic Graphplan algorithm first builds a representation of the search space as a planning graph, then finds a solution within this graph by traversing it backwards. This standard planning approach is modified to use Dijkstra's algorithm [13] when performing the backwards traversal, thus finding an optimised solution. The planning graph is extended to include labels associated with each proposition (i.e. each intermediate action between two vertices), where each label contains a layer number and associated execution costs. Dijkstra's algorithm is used to calculate the upcoming costs of each node in the graph. Then, a backtracking algorithm uses this information to determine the optimal solution.

2) EC Techniques: EC techniques applied to Web service composition are primarily concerned with optimising the QoS of candidates, as well as producing functionally correct solutions. However, despite listing constructs with conditional constraints as a possibility, approaches in this area do not support tasks that include the specification of conditions, a gap that this work seeks to fill. The works discussed below represent merely one of the many facets in the vast area of EC-based service composition.

In [7], a survey of the application of Genetic Algorithms (GA) to Web service composition is presented. GAs are a popular choice for tackling combinatorial optimisation problems, and thus have been widely applied to the problem of Web service composition. A common representation for atomic Web services uses integers, meaning that each service is represented as an integer. The encoding scheme for a composition is commonly done as an array of integers, but some authors have attempted to use a matrix that includes semantic information. Researchers also commonly investigate QoS representations, operators and fitness function variations to be applied to GA. An observed problem with the GA technique is that it tends to prematurely converge to locally optimal solutions, thus preventing further exploration of possibilities.

The approach described in [14] employs GP to generate composition solutions that are functionally correct at all stages of execution, and that are also optimised according to QoS attributes. A greedy algorithm is used to generate an initial population of functional candidates as graphs, which are then translated to a tree representation. Particular genetic operators are employed throughout the evolutionary process: the crossover operation ensures that the subtrees swapped between two candidates are functionally equivalent, thus preserving the correctness of the overall solution; the mutation operation replaces a randomly selected solution subtree with another one that is generated using the same algorithm employed for initialisation. Since the initialisation and genetic operators maintain functional correctness, the fitness function is only concerned with optimising solutions from a QoS perspective, which is done by calculating a weighted sum of the overall QoS attributes for the composition.

3) Hybrid approaches: Hybrid approaches combine elements of AI planning and optimisation techniques for solving the composition problem with functionally correct, optimised solutions [8], [15]. These hybrid approaches are quite similar to each other, relying on a directed acyclic graph as the base representation for a candidate solution, and then applying the optimisation techniques to this structure. However, despite incorporating the use of planning techniques, they do not include any discussion on the issue of producing solutions that satisfy user constraints or preferences. Another commonality between these works is that they require the use of SAWSDL-annotated datasets for testing, but these are not widely avail-

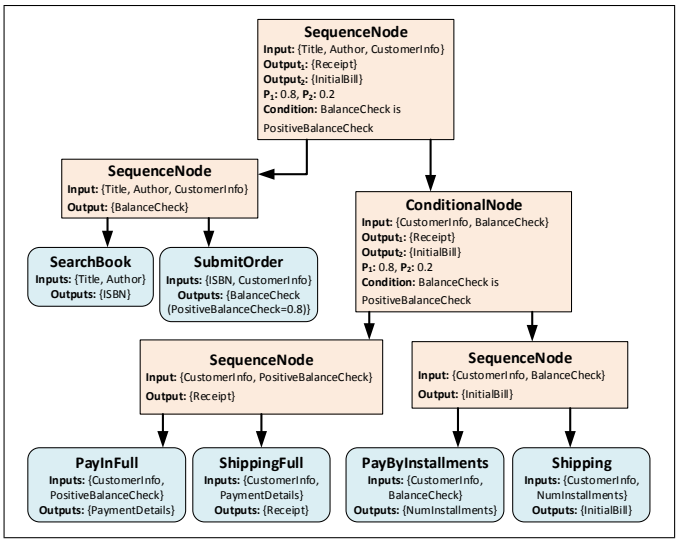

Fig. 5. Example of tree representation for Web service composition.

able to the research community. Therefore, authors developed their own datasets, and utilised each dataset's optimal task solution as the benchmark with which to evaluate the success of their implementation. More specifically, authors calculated the percentage of runs that culminated in the identification of the global optimum as the recommended solution.

In [8], an approach that combines AI planning and an immune-inspired algorithm is used to perform fully automated QoS-aware Web service composition, also considering semantic properties. One significant contribution of this work is the proposal of an Enhanced Planning Graph (EPG), which extends the traditional planning graph structure by incorporating semantic information such as ontology concepts. Given this data structure, the composition algorithm selects the best solution configuration from a set of candidates. A fitness function considering QoS values and semantic quality is used to judge the best solution, and a clonal selection approach is employed to perform the optimisation. Candidates cells (solutions) are cloned, matured (mutated by replacing services with others from the same cluster in the EPG) and the cell most suited to combating the invading organism (i.e. the best solution) is discovered.

III. Proposed Approach

Our proposed approach employs Genetic Programming to evolve solutions according to their overall Quality of Service, meanwhile maintaining their functional correctness. A candidate solution for a composition is represented as a tree, where the non-terminal nodes represent the composition flow constructs (sequence, parallel, and conditional), and the terminal (leaf) nodes represent the atomic Web services included in the composition. An example of such a tree is shown in Figure 5. To calculate the overall QoS of a tree, the formulae in Figures 2, 3 and 4 are employed, where the children of a current node act as the services used in the calculations. For Figure 5, for example, the QoS is first calculated for the sequence nodes directly above the leaves, then for the conditional node, and finally for the root node. This is discussed further in Subsection III-C.

A population initialisation algorithm similar to the one in [14] is employed, and so are constrained mutation and crossover operations. However, the difference of the current approach in comparison to [14] is that it also considers the case where a user requires a branching constraint to be met, like the one illustrated in Figure 1. The presence of branching means that different values are produced by a service at runtime, and these concrete values are subtypes of a statically defined concept. For example, a service that statically outputs a *fruit* concept may output the subtype *banana* at runtime, and thus branching conditions may be defined based on the different kinds of fruits. The relationship between types is held by a taxonomy that encodes the inputs and outputs pertinent to the candidate atomic services employed in the composition. For simplicity, the implementation presented in this work handles branching with two options (if and else), so this scenario will be the focus throughout the following sections.

A. Population Initialisation

As opposed to generating a composition candidate purely based on a set of available inputs and another of desired outputs, when handling branching constraints it is also necessary to consider a branching condition. Thus, a composition task with branching must include a set of available inputs, a condition of the is-a kind (e.g. if *banana* $\succeq$ *fruit*), and two sets of outputs, one expected in case the condition is met and the other in case it is not. With this information it is possible to run the initialisation algorithm and create a composition candidate in a graph format, later translated into a tree representation.

Input : I, O_1, O_2, C, P
Output: candidate tree T
1: **if** $O_2 \neq \emptyset$ **then**
2: $G_1 \leftarrow \texttt{createGraph}(I \cup C.if, O_1)$;
3: $G_2 \leftarrow \texttt{createGraph}(I \cup C.else, O_2)$;
4: $T_1 \leftarrow \texttt{toTree}(G_1.input)$;
5: $T_2 \leftarrow \texttt{toTree}(G_2.input)$;
6: $T_3 \leftarrow$ new $ConditionalNode(C)$;
7: $T_3.leftChild \leftarrow T_1$;
8: $T_3.rightChild \leftarrow T_2$;
9: **if** $C \sqsubseteq I$ **then**
10: $T_3.prob \leftarrow P$;
11: **return** T_3;
12: **else**
13: $G_4 \leftarrow \texttt{createGraph}(I, C.else)$;
14: $T_4 \leftarrow \texttt{toTree}(G_4.input)$;
15: $T_3.prob \leftarrow T_4.final.P$;
16: $T \leftarrow$ new $SequenceNode()$;
17: $T.leftChild \leftarrow T_4$;
18: $T.rightChild \leftarrow T_3$;
19: **return** T;
20: **end**
21: **else**
22: $G \leftarrow \texttt{createGraph}(I, O_1)$;
23: $T \leftarrow \texttt{toTree}(G.input)$;
24: **return** T;
25: **end**

Algorithm 1. Generating a new candidate tree or a mutated subtree.

Algorithm 1 is used to create a candidate which may incorporate a branching constraint, though it is also capable of handling unconditional tasks. It requires a set of inputs I, an if-else condition C, two sets of outputs O_1 (for if-branch)

and O_2 (for else-branch). The set of probabilities P is only required for a specific case, explained below, and C as well as O_2 are not required for tasks without branching. The intuition behind this algorithm is to assemble a candidate tree in parts. Firstly, if the task has in fact two sets of outputs and a condition C, then two sub-composition graphs G_1 and G_2 are generated to represent the if and the else branches, respectively. This generation is performed using the $createGraph$ algorithm proposed in [14]. Subsequently, these graphs are translated to trees T_1 and T_2 using the $toTree$ procedure proposed in Algorithm 2, and included as children of a $ConditionalNode$ created for condition C (the left child represent the if-branch, and the right child the else-branch).

```
   Procedure toTree()
   |  Input  : N
   |  Output: tree T
1: |  if N is leaf then
2: |  |  return N;
3: |  else if N = input then
4: |  |  if |N.to| = 1 then
5: |  |  |  T ← toTree(next.to[0]);
6: |  |  else
7: |  |  |  T ← createParallelNode(N.to);
8: |  |  end
9: |  else
10: |  |  r;
11: |  |  children ← N.to − output;
12: |  |  if |children| = 1 then
13: |  |  |  r ← toTree(children[0]);
14: |  |  else
15: |  |  |  r ← createParallelNode(children);
16: |  |  end
17: |  |  T ← new SequenceNode();
18: |  |  T.leftChild ← N;
19: |  |  T.rightChild ← r;
20: |  return T;
   Procedure createParallelNode()
   |  Input  : children
   |  Output: tree T
21: |  T ← new ParallelNode();
22: |  subtrees ← {};
23: |  foreach c in children do
24: |  |  S ← toTree(c);
25: |  |  subtrees ← subtrees ∪ {S};
26: |  end
27: |  T.children ← subtrees;
28: |  return T;
```

ALGORITHM 2. Converting graph into tree representation.

Secondly, the algorithm verifies whether the value used for condition C can be met by using the set of inputs I. If it can, then the set of probabilities P of each branch being executed is associated to the $ConditionalNode$ and the tree is returned (this assumes that the probabilities for obtaining a specific value for the if condition in C are already known – typically during mutation). If it cannot, then another sub-composition graph G_4 is generated and translated to T_4, creating the part of the composition that leads from the available inputs I to the value used in condition C. In this case, the probability associated with each branch of the condition is extracted from the last service in T_4 (i.e. the service that produces the overall sub-composition output that satisfies the value used in condition C), and a $SequenceNode$ is created as the tree root to link this deterministic part of the composition (T_4) to the conditional part (T_3). Finally, if no condition C and output set O_2 are provided, then the candidate is generated purely by using the $createGraph$ and $toTree$ algorithms.

For reasons of brevity, the $toTree$ procedure shown in Algorithm 2 for converting a graph representation of a composition to a tree representation is not explained in detail. Its general idea is the same as the one discussed in [16], which is to recursively traverse a graph, starting from the start ($input$) node and working towards the end ($output$) node. At each node, the number of outgoing edges determines whether to create a sequential or a parallel construct, and $toTree$ recurses on each of the destination nodes of these outgoing edges. It is important to note that after creating the candidate tree, it must be traversed to determine the input values required to execute each node of the tree, and the output values produced by each node. This information is recorded within each node.

B. Mutation and Crossover

The crossover operator employed in the evolutionary process swaps any two leaf nodes (i.e. Web services), one from each candidate, provided that these two leaves are functionally equivalent in terms of input and output values but represent different Web services. This particular crossover operation implementation was chosen because it provides an effective mechanism for performing Web service selection, whose objective is to identify the best service to perform a task out of a set of candidates providing the same functionality. The mutation operator selects a node of the candidate tree at random, and using its input and output information produces a new subtree that may be structurally different yet provides the same functionality. The subtree is generated using Algorithm 1, and is used to replace the originally selected node in the candidate.

C. Fitness Function

The fitness function is employed to measure the overall quality of a composition candidate. It does so by calculating the weighted sum of the four overall composition QoS attributes [12], using the following function:

$$fitness_i = w_1(1 - A_i) + w_2(1 - R_i) + w_3 T_i + w_4 C_i \quad (1)$$

where $\sum_{i=1}^{4} w_i = 1$

This function produces values in the range [0, 1], with 0 representing the best possible quality and 1 the worst. The A, R, T, and C values for each composition are calculated according to the formulae displayed in Figures 2, 3, and 4. More specifically, the overall values are calculated starting from the tree leaves and working upwards towards the tree root. Since the resulting fitness value must be between 0 and 1, the weights used in the function must add up to 1, and the QoS values must be normalised to the [0, 1] range. As availability (A) and reliability (R) are probability values, they already meet this requirement. The time and cost values for each Web service, on the other hand, are normalised by using the smallest (Min) and largest (Max) respective values in the

dataset, according to the following functions:

$$T_i = \begin{cases} \frac{T_{orig} - T_{min}}{T_{max} - T_{min}}, & \text{if } T_{max} - T_{min} \neq 0. \\ 1 & \text{otherwise.} \end{cases} \quad (2)$$

$$C_i = \begin{cases} \frac{C_{orig} - C_{min}}{C_{max} - C_{min}}, & \text{if } C_{max} - C_{min} \neq 0. \\ 1 & \text{otherwise.} \end{cases} \quad (3)$$

Finally, A and R are offset by 1 in the fitness function. This is because A and R indicate the best qualities (i.e. the highest probabilities) with the highest possible values, whereas the fitness function indicates the best possible candidate quality with the lowest possible value (it is a minimising function).

D. GP Algorithm for Composition

When applying GP to the service composition problem, each candidate in the population represents a possible candidate solution. The initial population is created as described above, and the fitness of each candidate is evaluated using the fitness function. The fittest individuals are reproduced to the next generation, while other candidates are created in the next generation by applying the mutation and crossover operations to currently promising individuals. This new generation is then evaluated according to the fitness of candidates, and the process is repeated for a fixed number of generations. Lastly, the candidate with the highest fitness in the final population is chosen as the composition solution.

IV. Experiment Design

Experiments were conducted to evaluate the performance of the proposed GP approach to service composition. The greatest hurdles in this process were the lack of datasets that supported the creation of compositions with branching, and the lack of comparable approaches that produce solutions with multiple output possibilities. In order to enable the specification of branching conditions it is necessary to utilise a dataset that includes information about the different subtypes of outputs Web services may produce at runtime. For example, a Web service that outputs the result of a balance check to a customer's account (shown in Figure 1) may return a positive balance check if the customer has enough funds to pay in full, or a negative balance check otherwise. Therefore, the service outputs the general *balance check* type, but at runtime it may produce either the *positive balance check* or the *negative balance check* subtypes. By having this information, it is possible to create a composition whose execution is conditional on the runtime value of *balance check*.

A consequence of considering the output possibilities of a service is that each possibility has an associated probability of being produced at runtime. For example, the probability of a balance check being positive could be 0.8, meaning that the probability of it being negative would be merely 0.2. These probabilities have been shown to be calculable by tracking the logged activities of a Web service, and to be a valuable tool in predicting execution patterns in a composition workflow [17]. In our work, output probabilities are used to calculate the quality of compositions that include a branch construct (as shown in Figure 5), where the branching condition is based on the possible runtime output values for a service. This means that the probability of a branch being chosen is dependent on the probabilities of a certain output being produced by the service whose value is used to satisfy the branching condition.

A. Datasets

The datasets to be used when testing this approach had to contain input and output information, including multiple output possibilities when appropriate, probability values for each of those possibilities, and QoS values for each Web service. As no datasets were found to provide all of these items, existing datasets were extended to include all of the necessary information. The datasets in question are those used in [14], which are an augmented version of the 2008 Web Service Challenge (WSC2008) [18] that includes QoS attributes. Those datasets were chosen because they already provided an ontology of input and output value types that could be used to generate multiple output possibilities for a service, as well as quality values for optimisation.

The datasets were extended in two steps. In the first step, new composition tasks were created for each dataset. As opposed to requiring the production of a single output set, the new tasks demand one of two sets to be produced depending on whether a branching condition is met, as exemplified by Figure 6. Each of the tasks was manually ensured to be achievable. Additionally, new tasks that do not require branching were also created based on these branching tasks, for comparison purposes. These non-conditional tasks are similar to the branching ones, but they do not require conditions or multiple output sets (either the if-branch or else-branch output set was kept). In the second step, to ensure that branching was in fact possible, all services in the dataset which produce the output types used in the branching condition were extended to produce two sets of outputs, one producing an instance of the specific concept (the *if* condition) and another producing an instance of the general concept (the *else* option). Finally, probabilities were randomly assigned to each output possibility manually, ensuring that the probabilities for all output sets of a same service added up to 1 as they would in a real situation.

B. Parameters

The approach was implemented using *Epochx* [19], a Java framework for GP applications. Tests were carried out using a personal computer with an Intel Core i7-4770 CPU (3.4 GHz), and 8 GB RAM. 50 independent runs were conducted for each dataset, using conditional (branching) tasks as well as the non-conditional (non-branching, separate if and else) tasks. For each run, a population of 20 candidates was used, and the evolutionary process produced exactly 50 generations. While a population of size 20 is generally considered small for GP, it was nevertheless used because larger populations led to computation times that were inadmissible for this problem domain. Crossover and mutation probabilities were set to 0.9 and 0.1, and elitism was set to 1 candidate. Tournament selection was used, with a tournament size of 7. As the candidate trees were already constrained by functional correctness mechanisms when initialising and evolving the population, no limit to the candidate tree depths was applied. All fitness function weights were set to 0.25, indicating that all QoS attributes are considered equally important by the composition requestor.

V. Results

A. Overall Results

Experiment results are shown in Tables I and II. Table I displays the results obtained when running the conditional tasks, with the first column showing the dataset used for

```
<task>
      <provided>
            <instance name="inst507612613"/>
            <instance name="inst211649568"/>
            <instance name="inst1610541870"/>
            <instance name="inst1318470020"/>
            <instance name="inst511297711"/>
            <instance name="inst1695942861"/>
            <instance name="inst120304438"/>
            <instance name="inst1319288246"/>
      </provided>
      <options>
            <condition>
                  <general concept="con1404081368"/>
                  <specific concept="con2027332959"/>
            </condition>
            <if>
                  <instance name="inst688870502"/>
                  <instance name="inst1012480226"/>
                  <instance name="inst432650641"/>
                  <instance name="inst1832758643"/>
            </if>
            <else>
                  <instance name="inst907818669"/>
                  <instance name="inst441457430"/>
                  <instance name="inst1857750899"/>
                  <instance name="inst739462845"/>
                  <instance name="inst1242487909"/>
                  <instance name="inst629067506"/>
            </else>
      </options>
</task>
```

Fig. 6. A task containing multiple output possibilities depending on a branching condition. The condition should be read as "if *con1404081368* is of subtype *con2027332959* at runtime, then the composition should produce *if* outputs".

execution and the number of services in it, the second column showing the mean fitness of the best solution, and the third column showing the mean execution time in seconds for a run (the latter two columns include standard deviations). Table II contains the results for running the non-conditional tasks, both the if branch and the else branch. The first three columns are laid out as in Table I and show the if-branch results, while the fourth and fifth columns show else-branch results. All fitness and time values, including their standard deviations, are rounded to 3 decimal points of precision. An additional test to verify the effectiveness of the evolutionary process in identifying optimal solutions for conditional tasks was also carried out. In this test, an artificial optimal solution with fitness = 0 (the best possible fitness) was added to dataset 1, and the evolutionary process was conducted for this data with the same settings as before. Our approach found the artificial optimum (i.e. fitness = 0) for all runs, demonstrating that it is capable of effectively exploring the search space.

B. Evolved Programs

A real candidate solution to the task shown earlier (Figure 6) is depicted in Figure 7, demonstrating through an example that the proposed approach is capable of producing and optimising fully functional solutions. The root of the solution tree, which contains the input, both sets of outputs, and the branching condition, perfectly matches the information requested in the composition task, meaning that the task is fully satisfied. The root also displays a set of probabilities which shows that the likelihood of meeting the if-condition at

	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 I. MEAN FITNESS VALUES AND EXECUTION TIMES FOR EACH DATASET, USING TASKS REQUIRING THE CHOICE CONSTRUCT.

	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

TABLE II. MEAN FITNESS VALUES AND EXECUTION TIMES FOR EACH DATASET, WITH BRANCHES CREATED SEPARATELY.

runtime is 0.4 for this composition. The left child of the root node (i.e. *serv1252722885*) produces outputs that lead to the branching condition, either producing an instance that matches the if-condition (*inst572533116*) or an instance that matches the else-condition concept (*inst2096765192*). Note that the other two outputs produced by this service do not vary. Finally, the branching condition concept is used as an input to one of the children of the conditional node (either *serv763420204* or *serv2014838942*) depending on its runtime value.

C. Discussions

For both conditional and non-conditional tasks, the small standard deviation values for all fitness means indicates that the algorithm converged to solutions with very similar fitnesses during the runs for each dataset. We hypothesise that these fitness means are quite close to the global optimal solution for each task, though the exact optimal values are not known. The fitness of conditional and non-conditional tasks is calculated using the same function, however the results shown here are not directly comparable because the conditional solutions group sections of the non-conditional branches according to probabilities. However, by averaging the fitness values of the two non-conditional tasks for a dataset we obtain approximately the fitness of the corresponding conditional task, potentially showing that conditional and non-conditional solutions for a given dataset present a similar structure. In terms of execution time, an increase to the mean occurs roughly proportionately to the increase in the total number of services in the dataset. The mean execution time for the conditional tasks is approximately equivalent to the sum of the mean times for each non-conditional branch, showing that there is no performance slowdown arising from the use of the conditional

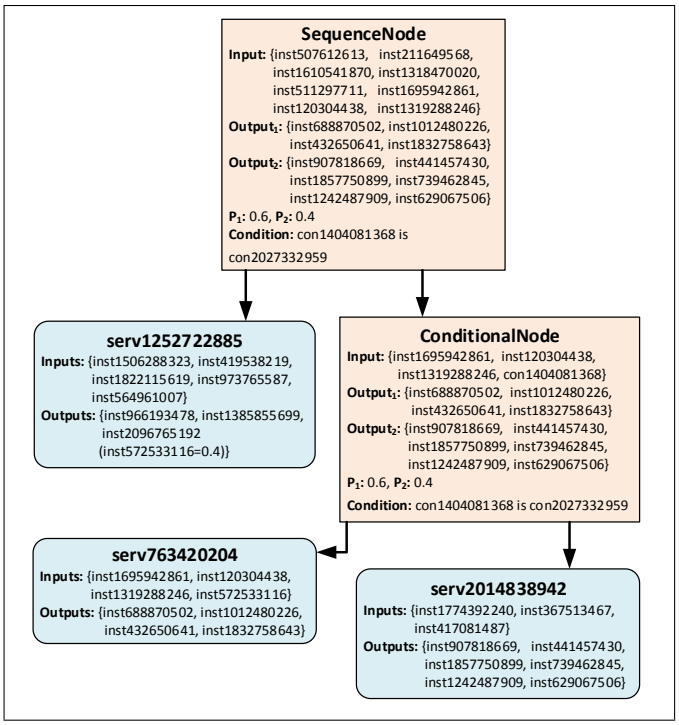

Fig. 7. A possible solution to the composition task shown earlier.

construct. This is a key finding, since it demonstrates that the proposed approach allows a conditional constraint to be met while also being just as efficient as a non-conditional method. Despite this positive outcome, results point out that both conditional and non-conditional tasks for the largest dataset require several minutes to execute, which is not ideal. An informal investigation of the program's performance has revealed that the data structure used to represent the concept ontology presents an execution time bottleneck, leading to the conclusion that improvements to its implementation could be performed.

VI. Conclusions

This paper presented a genetic programming approach to QoS-aware automated Web service composition. The novelty of this approach is that it addresses three composition dimensions simultaneously: the basic functional correctness of solutions (i.e. input-output matches), the use of branching constraints required by the user (i.e. complex functional correctness requirements), and the optimisation of solutions according to QoS attributes. The first two dimensions were handled by the utilisation of an algorithm to initialise and mutate solutions according to the required constraints, and the third dimension was handled by the utilisation of a fitness function that ranks solutions according to their overall QoS. No datasets with conditional tasks were available to test this approach, therefore the WSC2008 datasets were extended and used for this purpose. Results showed that the proposed approach was capable of identifying solutions that are fully functional, contain branches, and are quality-optimised for all datasets. The proposed approach was also shown to take a similar amount of time to that of a method that does not support branching. Only one branching condition is currently handled, so future work could extend this approach to flexibly handle scenarios that require several branches. The project could also be extended to handle more complex conditions, likely leading to the employment of a language to describe the constraints. Finally, the evolutionary process of the proposed technique should be studied in more detail, analysing the population's convergence behaviour through additional experiments.

References

[1] R. Perrey and M. Lycett, "Service-oriented architecture," in *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*. IEEE, 2003, pp. 116–119.

[2] K. Gottschalk, S. Graham, H. Kreger, and J. Snell, "Introduction to web services architecture," *IBM systems Journal*, vol. 41, no. 2, pp. 170–177, 2002.

[3] N. Milanovic and M. Malek, "Current solutions for web service composition," *IEEE Internet Computing*, vol. 8, no. 6, pp. 51–59, 2004.

[4] P. Wang, Z. Ding, C. Jiang, and M. Zhou, "Automated web service composition supporting conditional branch structures," *Enterprise Information Systems*, vol. 8, no. 1, pp. 121–146, 2014.

[5] S. Sohrabi, N. Prokoshyna, and S. A. McIlraith, "Web service composition via the customization of golog programs with user preferences," in *Conceptual Modeling: Foundations and Applications*. Springer, 2009, pp. 319–334.

[6] M. Chen and Y. Yan, "Qos-aware service composition over graphplan through graph reachability," in *Services Computing (SCC), 2014 IEEE International Conference on*. IEEE, 2014, pp. 544–551.

[7] L. Wang, J. Shen, and J. Yong, "A survey on bio-inspired algorithms for web service composition," in *Computer Supported Cooperative Work in Design (CSCWD), 2012 IEEE 16th International Conference on*. IEEE, 2012, pp. 569–574.

[8] C. B. Pop, V. R. Chifu, I. Salomie, and M. Dinsoreanu, "Immune-inspired method for selecting the optimal solution in web service composition," in *Resource Discovery*. Springer, 2010, pp. 1–17.

[9] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone, *Genetic programming: an introduction*. Morgan Kaufmann San Francisco, 1998, vol. 1.

[10] P. Wohed, W. M. van der Aalst, M. Dumas, and A. H. Ter Hofstede, "Analysis of web services composition languages: The case of bpel4ws," in *Conceptual Modeling-ER 2003*. Springer, 2003, pp. 200–215.

[11] Y. Yu, H. Ma, and M. Zhang, "An adaptive genetic programming approach to qos-aware web services composition," in *Evolutionary Computation (CEC), IEEE Congress on*. IEEE, 2013, pp. 1740–1747.

[12] A. da Silva, H. Ma, and M. Zhang, "A graph-based particle swarm optimisation approach to qos-aware web service composition and selection," in *Evolutionary Computation (CEC), 2014 IEEE Congress on*, July 2014, pp. 3127–3134.

[13] S. Skiena, "Dijkstra's algorithm," *Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, Reading, MA: Addison-Wesley*, pp. 225–227, 1990.

[14] A. Wang, H. Ma, and M. Zhang, "Genetic programming with greedy search for web service composition," in *Database and Expert Systems Applications*. Springer, 2013, pp. 9–17.

[15] C. B. Pop, V. Rozina Chifu, I. Salomie, R. B. Baico, M. Dinsoreanu, and G. Copil, "A hybrid firefly-inspired approach for optimal semantic web service composition," *Scalable Computing: Practice and Experience*, vol. 12, no. 3, 2011.

[16] C. D. Nguyen, T. A. Dung, and T. H. Cao, "Text classification for dag-structured categories," in *Advances in Knowledge Discovery and Data Mining*. Springer, 2005, pp. 290–300.

[17] D. Clarke, I. Saleh, and M. B. Blake, "Modelling service workflow outcomes by assessing the underlying message flows," in *WETICE Conference (WETICE), 2014 IEEE 23rd International*. IEEE, 2014, pp. 9–14.

[18] A. Bansal, M. B. Blake, S. Kona, S. Bleul, T. Weise, and M. C. Jaeger, "Wsc-08: continuing the web services challenge," in *E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services, 2008 10th IEEE Conference on*. IEEE, 2008, pp. 351–354.

[19] F. Otero, T. Castle, and C. Johnson, "Epochx: Genetic programming in java with statistics and event monitoring," in *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*. ACM, 2012, pp. 93–100.