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

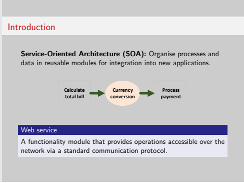
A GP Approach to QoS-Aware Web Service Composition including Conditional Constraints

Alexandre Sawczuk da Silva, Hui Ma, Mengjie Zhang

Evolutionary Computation Research Group
School of Engineering and Computer Science, Victoria University of Wellington

IEEE Congress on Evolutionary Computation, 25-28 May 2015

Good afternoon everyone, I'm Alex and today I will talk about our paper on A GP Approach to Web service composition with conditional constraints.



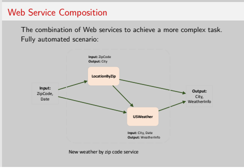
Introduction

Service-Oriented Architecture (SOA): Organise processes and data in reusable modules for integration into new applications.

Web service

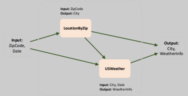
A functionality module that provides operations accessible over the network via a standard communication protocol.

- As the web becomes more and more pervasive, so does the concept of Service-Oriented Architecture.
- The idea in SOA is to organise business processes and data into independent modules which can then be reused in applicactions as needed.
- For example, an e-commerce website may need to perform currency conversion depending on the location of its current customer, and so it uses an already existing currency conversion module.
- The basic components in SOA are services, which are functionality modules accessible over the network. In this example, currency conversion is a Web service.



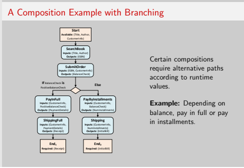
Web Service Composition

The combination of Web services to achieve a more complex task.
Fully automated scenario:



New weather by zip code service

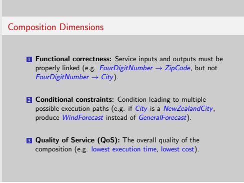
- The combination of multiple modular Web services in order to achieve a single, more complex task is known as Web service composition.
- Developing a system capable of creating such compositions in a fully automated manner is one of the holy grails of the field.
- With full automation, new services can be created simply by specifying the inputs the new service should require and the outputs it should produce, and the resulting composition is treated as a black box.
- For example, imagine we need to create a service that provides the weather forecast of a location based on its ZIP code. An automated composition system could identify and connect two different services, one that identifies a city based on a ZIP code and another that uses the city and date information to produce the forecast.


A Composition Example with Branching
Start
Available: {Title, Author, CustomerInfo}
SearchBook
Inputs: {Title, Author}
Outputs: {ISBN}
SubmitOrder
Inputs: {ISBN, CustomerInfo}
Outputs: {BalanceCheck}
If BalanceCheck is PositiveBalanceCheck
Else
PayInFull
Inputs: {CustomerInfo, PositiveBalanceCheck}
Outputs: {PaymentDetails}
PayByInstallments
Inputs: {CustomerInfo, BalanceCheck}
Outputs: {NumInstallments}
ShippingFull
Inputs: {CustomerInfo, PaymentDetails}
Outputs: {Receipt}
Shipping
Inputs: {CustomerInfo, NumInstallments}
Outputs: {InitialBill}
End$_1$
Required: {Receipt}
End$_2$
Required: {InitialBill}

Certain compositions require alternative paths according to runtime values.

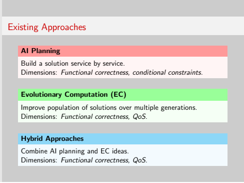
Example: Depending on balance, pay in full or pay in installments.

- In certain cases, we might need alternative paths in a composition, depending on the runtime values produced.
- For example, in the case of a service that models the purchase of a book, the customer might pay for the book in full or in installments, depending on their current balance.
- So, once the balance is check, one of two different branches may be executed, leading to different outputs.


Composition Dimensions

1. **Functional correctness:** Service inputs and outputs must be properly linked (e.g. *FourDigitNumber* → *ZipCode*, but not *FourDigitNumber* → *City*).
2. **Conditional constraints:** Condition leading to multiple possible execution paths (e.g. if *City* is a *NewZealandCity*, produce *WindForecast* instead of *GeneralForecast*).
3. **Quality of Service (QoS):** The overall quality of the composition (e.g. lowest execution time, lowest cost).

- As we can see, the complexity of Web service composition lies in the number of different dimensions it must consider at the same time.
- In the first dimension, we must ensure that services have their inputs and outputs properly linked. For example, a service that produces a four-digit number should be able to feed its output to the input of a service that accepts a zip code, however a four-digit number should not be accepted as a city name.
- In the second dimension, we must consider conditions that lead to different execution paths. For example, if a city is located in New Zealand, we would like a wind-only forecast as opposed to a general one.
- In the third dimension, we must produce compositions with the best possible quality attributes. For example, we would like it to have the lowest possible execution time as well as the lowest possible financial cost.



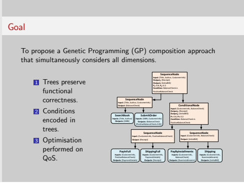
Existing Approaches

AI Planning
Build a solution service by service.
Dimensions: *Functional correctness, conditional constraints.*

Evolutionary Computation (EC)
Improve population of solutions over multiple generations.
Dimensions: *Functional correctness, QoS.*

Hybrid Approaches
Combine AI planning and EC ideas.
Dimensions: *Functional correctness, QoS.*

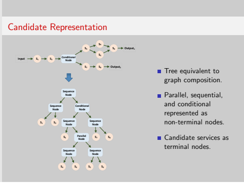
- Several techniques have been proposed to address the composition problem, however these approaches only account for at most two dimensions at once.
- In AI planning, a solution is built step by step, in each step adding a service and verifying the overall state of the composition. This approach typically verifies functional correctness and conditional constraints, but not the overall QoS.
- In evolutionary computation, a population of solutions is created and improved over multiple generations. This approach typically verifies functional correctness and optimises the overall QoS, but does not address conditional constraints.
- Hybrid approaches combine elements of AI planning and EC, making it easier to check the functional correctness and at the same time optimise the overall QoS. However, this approach also fails to address conditional constraints.


Goal

To propose a Genetic Programming (GP) composition approach that simultaneously considers all dimensions.

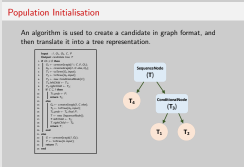
1. Trees preserve functional correctness.
2. Conditions encoded in trees.
3. Optimisation performed on QoS.

- Given these limitations, the objective of the approach proposed in this paper is to address these three dimensions simultaneously by using genetic programming.
- The functional correctness dimension is addressed by restricting the way in which services are linked to each other in each solution tree.
- Conditional constraints are addressed by including a conditional nodes as possible nodes in the tree representation.
- The Quality of Service dimension is addressed by using a fitness function that optimises solutions on quality attributes.
- Let's understand these in more detail.


Candidate Representation

- Tree equivalent to graph composition.
- Parallel, sequential, and conditional represented as non-terminal nodes.
- Candidate services as terminal nodes.

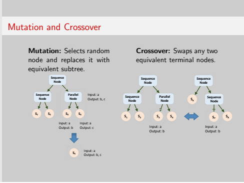
- Web service compositions can be intuitively represented as directed acyclic graphs, as shown by the first structure. We can see that parallel and sequential configurations are encoded by the edges, and conditional constraints by special nodes.
- However, a tree representation is required for GP. Therefore, in our approach parallel, sequential, and conditional configurations are all represented as non-terminal nodes of the tree, while candidate atomic services are the terminal nodes.
- The tree structure shown here is equivalent to the composition shown in graph form.


Population Initialisation

An algorithm is used to create a candidate in graph format, and then translate it into a tree representation.

```
Input   : $I$, $O_1$, $O_2$, $C$, $P$
Output: candidate tree $T$
1: if $O_2 \neq \emptyset$ then
2:     $G_1 \leftarrow$ createGraph($I \cup C.if, O_1$);
3:     $G_2 \leftarrow$ createGraph($I \cup C.else, O_2$);
4:     $T_1 \leftarrow$ toTree($G_1.input$);
5:     $T_2 \leftarrow$ 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:        $G_4 \leftarrow$ createGraph($I, C.else$);
14:        $T_4 \leftarrow$ 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:    $G \leftarrow$ createGraph($I, O_1$);
23:    $T \leftarrow$ toTree($G.input$);
24:    return $T$;
25: end
```

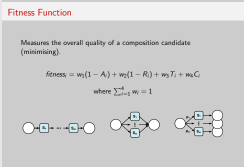
- In order to create candidates with correct input-output connections between services, we employ an initialisation algorithm that creates compositions in graph form and then translates them into tree form.
- The basic idea of this algorithm is to build each part of the tree separately, and then assemble the parts into a candidate. Each part is built ensuring that the connections are correct. We begin by building the parts that go from the conditional node to the overall outputs (T_1 and T_2), and both possibilities are built. Then we move on to the part that leads to the condition from the overall inputs (T_4). These are connected to a sequence and a conditional node.
- For simplicity, in this work we assume that each composition has only one conditional node, so its position is always the same in the tree.


Mutation and Crossover

Mutation: Selects random node and replaces it with equivalent subtree.

Crossover: Swaps any two equivalent terminal nodes.

- During the evolution process, the custom operators used are mutation and crossover. Both of these operators maintain the functional correctness of candidates, that is, they ensure that input-output connections still make sense.
- The mutation operator randomly selects a node in the tree, removes that subtree and replaces it with a newly generated subtree. The new subtree may have a different structure than the old one, but it is functionally equivalent to it, i.e. produces the same output given the same input. New subtrees are generated using the initialisation algorithm.
- The crossover operator finds any functionally equivalent terminal nodes in two candidate trees and swaps them. We chose this crossover operation because it models the process of Web service selection, where the best service is chosen among a group of functionally equivalent candidates.



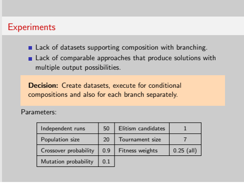
Fitness Function

Measures the overall quality of a composition candidate (minimising).

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

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

- The fitness function is used to optimise the composition candidates according to their overall quality of service attributes. The values produced range from 0, in the best possible case, to 1, in the worst possible.
- These attributes are combined into a single value by using a weighted sum, where the weights add up to 1. The individual QoS values are also normalised to remain between 0 and 1.
- The different overall quality attributes are calculated differently depending on the constructs of the composition. For example, the execution time is the sum of individual service times in a sequential construct; it is the longest individual time in a parallel construct; it is a weighted sum of individual times, where the weights are probabilities, in a branching construct.



Experiments

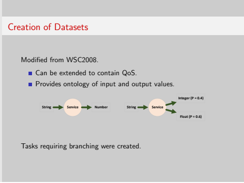
- Lack of datasets supporting composition with branching.
- Lack of comparable approaches that produce solutions with multiple output possibilities.

Decision: Create datasets, execute for conditional compositions and also for each branch separately.

Parameters:

Independent runs	50	Elitism candidates	1
Population size	20	Tournament size	7
Crossover probability	0.9	Fitness weights	0.25 (all)
Mutation probability	0.1		

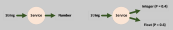
- The performance of the proposed approach was evaluated through a set of experiments.
- There were two big difficulties in conducting these experiments: firstly, there is a lack of existing datasets that support branching in the field; secondly, there are no other approaches that can be used for direct comparison (they have no branching and QoS optimisation simultaneously).
- Having these problems in mind, our decision was to create our own datasets, and to run experiments both for conditional tasks with two branches and for the branches separately.
- While definitive conclusions cannot be made by comparing the results with branching to those without branching, interesting patterns may nevertheless emerge.
- The parameters shown here were used for testing.



Creation of Datasets

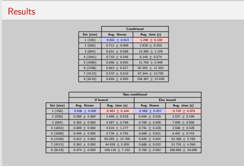
Modified from WSC2008.

- Can be extended to contain QoS.
- Provides ontology of input and output values.



Tasks requiring branching were created.

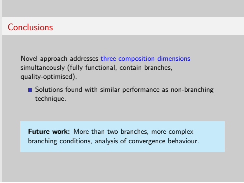
- The datasets were created by extending WSC2008, a well-known dataset in the field. We chose WSC2008 because it is easily extendable to contain QoS values, and it already used an ontology of input and output values, meaning that services which produce a certain output type can be easily extended to produce two different output subtypes of the original.
- For example, if a service originally produces number, it can be changed to produce an integer 40% of the time and a floating point number 60% of the time.
- Services in the datasets were modified this way, with arbitrary probabilities which always add to 1.
- The original dataset tasks were also extended to require the use of branching.


Results

	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

	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

- The results are divided into two tables, one for solutions with branching and the other for when the branches are individually produced.
- By comparing the two tables we see that two patterns arise: we see that the fitness value of the conditional solution is roughly equal to the average of the fitness values of the two separate branches, which potentially indicates that these solutions have a similar structure; we also see that the mean execution time for creating the conditional solution is roughly the sum of the mean times for each branch, showing that there is no performance slowdown for the creation of branched solutions.
- This is key finding which indicates that branched approach can be just as efficient as a non-branched one.



Conclusions

Novel approach addresses three composition dimensions simultaneously (fully functional, contain branches, quality-optimised).

- Solutions found with similar performance as non-branching technique.

Future work: More than two branches, more complex branching conditions, analysis of convergence behaviour.

-

Thank you!

Questions?

Thank you! Are there any questions?