GraphEvol: A Graph Evolution Technique for Web Service Composition

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

Abstract

The *IJCAI–15 Proceedings* will be printed from electronic manuscripts submitted by the authors. The electronic manuscript will also be included in the online version of the proceedings. This paper provides the style instructions.

1 Introduction

A Web service can be defined as a software module that accomplishes a specific task and that is made available for requests over the Internet. The fundamental benefit of such modules is that they can be interwoven with new applications, preventing developers from rewriting functionality that has already been implemented. Service-Oriented Architecture (SOA) is a paradigm that expands on this idea, advocating that the main atomic components of a software system should be Web services, since this maximizes code reuse and information sharing. As services are typically present standard interfaces, the possibility arises to create approaches capable of combining services automatically according to the final desired system, in a process known as *Web service composition*. The objective of these approaches is to produce a workflow, i.e. a directed acyclic graph (DAG), stipulating the sequence in which each atomic service should be executed, as well as the output-input connections between services.

Many approaches to Web service composition have been proposed in the literature, from variations on AI planning techniques to the employment of integer linear programming solvers. In particular, promising results have been achieved with the use of Evolutionary Computation (EC) techniques, because their search methods successfully handle the large search space characteristic of the composition problem. Existing evolutionary techniques for fully automated Web service composition can be divided into two different groups according to the extent of the composition they perform: *semi-automated composition techniques*, assume that a general workflow of abstract tasks has already been provided and that the algorithm must simply find the best candidate for each task slot; *fully automated composition techniques*, on the other hand, make no such assumption, meaning that they construct the task workflow and select the most suitable candidates simultaneously. Because of their greater capabilities, fully automated composition techniques are the focus of this paper.

While semi-automated composition can be accomplished using a variety of techniques such as Genetic Algorithms (GA) and Particle Swarm Optimisation (PSO) [], fully automated composition using EC is currently mostly restricted to techniques employing the traditional Genetic Programming (GP) model [Rodriguez-Mier *et al.*, 2010]. In this paradigm, each each composition candidate is a tree that represents an underlying graph solution that can be obtained through a translation process. The question then arises whether it is possible to represent a candidate directly as a graph, and whether the system's performance is affected from doing so. Thus, the objective of this paper is to present and analyse *GraphEvol*, an evolutionary computation technique for Web service composition where each candidate is represented as a DAG and modified while remaining in that form. The remainder of this paper is organised as follows: Section 2 presents background information concerning Web service composition; Section 3 presents the proposed technique; Section 4 discusses the experiments conducted to compare the performance GraphEvol to that of a GP composition approach; Section 5 describes the experiment results; Section 6 concludes the paper.

2 Background

2.1 The Web Service Composition Problem

In a standard composition problem, a user would like to obtain a composite Web service with a particular functionality. The user makes a request to a Web service composition system specifying the *inputs* the composite service should accept, the *outputs* it should produce, the *constraints* it should consider (e.g. the resulting composite service should have the lowest possible execution time), and the *repository* from which it should select the atomic services to include in the composition. Given this information, the composition system *discovers* relevant services from the repository, and uses these services as potential candidates to be included in the final composition. A composition algorithm is run to *select* appropriate services at the same time it *builds* a workflow denoting how each service in the composition interacts with the others with regards to their required input and produced output.

A classic example of automated Web service composition

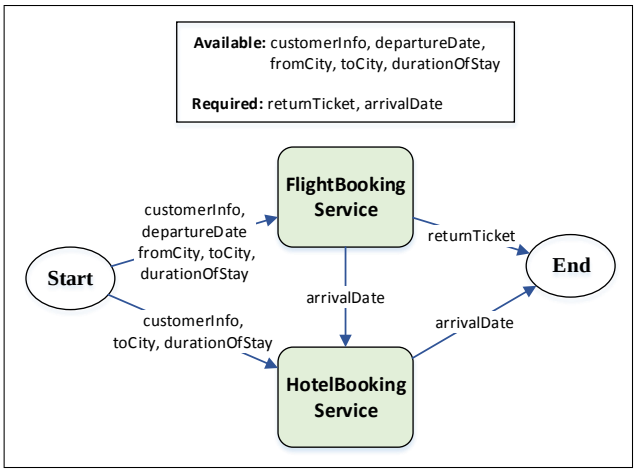

Figure 1: Example of a solution to a Web service composition task.

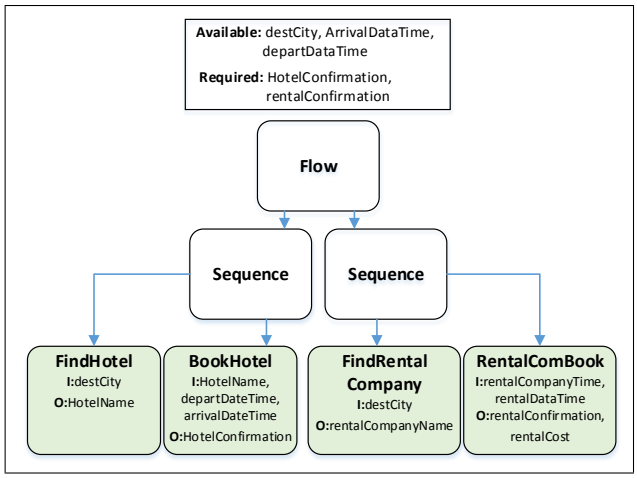

Figure 2: Example of a typical GP candidate composition tree [Aversano *et al.*, 2006].

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.

2.2 Related Work

There are many publications on the subject of EC applied to Web service composition, but this Subsection will focus on those which perform fully automated composition, since their composition outcomes are analogous to those of GraphEvol. One of the pioneering GP composition approaches [Aversano *et al.*, 2006] uses workflow constructs as the non-terminal tree nodes and Web service candidates as the terminal nodes, where workflow constructs represent the output-input connections between two services. For example, if two services are sequentially connected, so that output of service *A* is used as the input of service *B*, this would be represented by a *sequence* workflow construct having *A* as the left child and *B* as the right one. Likewise, if two services *A* and *B* have independent inputs and outputs and can be executed in parallel, this can be encoded as a *flow* construct having *A* and *B* as its children. An example of a tree generated by this technique is shown in Figure 2, adapted from an illustration in the referenced work. The initial population is initialised randomly, which means that the initial compositions represented in that generation are very unlikely to be executable, since their inputs and outputs are mismatched. The aim of the technique is to improve these output-input matches by employing a fitness function that calculates their correspondence and awards higher fitness scores to those candidates with larger overlaps. The genetic operators employed for this evolutionary process are crossover, where two subtrees from two individuals are randomly selected and swapped, and mutation, where a subtree for an individual is replaced with a randomly generated substitute.

Another GP approach [Rodriguez-Mier *et al.*, 2010] follows a similar tree representation to the one discussed above, as well as a similar implementation of mutation and crossover operators. The greatest distinction between this approach and others is in its use of a context-free grammar to generate the initial population of candidates and to create the subtrees used during mutation. The grammar restricts the tree structure configurations allowed in the tree, thus reducing the search space considered when searching for a solution. Another key difference of this approach is in its fitness function, which not only minimises the output-input mismatches in candidates, but also minimises the overall number of atomic services in the composition and the size of the largest sequential chain of such services. These two additions to the fitness function are aimed at reducing the overall execution time and cost of the resulting compositions. This technique was tested against two well-known semantically-annotated datasets, OWL-S TC [Kuster *et al.*, 2008] and WSC2008 [Bansal *et al.*, 2008], showing that it yields valid solutions to all composition tasks while requiring a relatively low amount of execution time.

In addition to GP, an approach using PSO has also been shown to be a suitable method for fully automated Web service composition [da Silva *et al.*, 2014]. In this technique, candidates encoded as a vectors of weights (particles), each ranging from 0 to 1. These weights are mapped to the edges of a *master graph*, which is a data structure created at initialisation time to represent all the possible output-intput relationships between relevant candidate services. The central idea of this technique is to utilise a greedy algorithm to extract non-redundant functional solutions from the master

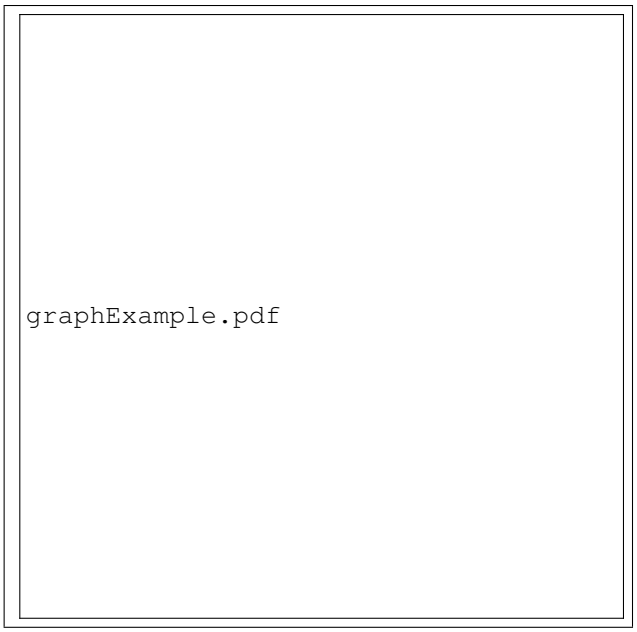

Figure 3: Example of a candidate composition graph using GraphEvol.

graph, using the weights as guides that prioritise the choice of certain edges and nodes of the structure. The weights are evolved using the classical PSO algorithm, and at each generation the fitness of all solutions is calculated by extracting the underlying structure from the master graph. Since the master graph structure is built ensuring full matching of services inputs and outputs, the fitness function employed for evolution optimises the population according to non-functional quality (QoS) measures such as response time, availability, and reliability.

3 GraphEvol

GraphEvol, the evolutionary computation approach proposed in this work, bears many similarities with the GP approaches discussed above. Namely, it initialises and a population of candidates that are encoded using non-linear data structures, evolves this population using crossover and mutation operators, and evaluates the quality of each candidate based on the nodes included in its structure. However, as opposed to representing candidate compositions as trees that correspond to underlying graph structures, GraphEvol represents them directly as graphs. Figure 3 shows a graph representation example that is equivalent to the candidate tree shown in Figure 2. As a consequence of this direct graph representation, the mutation and crossover operators must be implemented differently, and so must the fitness function. Another important aspect of GraphEvol is that is uses a graph-building algorithm for creating new solutions, and for performing mutation and crossover. The general procedure for GraphEvol is shown in Algorithm 1, however each fundamental aspect of the proposed technique is explored in more detail in the following subsections.

ALGORITHM 1. Steps in the GraphEvol search approach.

1. Initialise the population using the graph building algorithm.
while *max. generations not met* **do**
  2. Select the fittest graph candidates for reproduction.
  3. Apply mutation, crossover, and elitism to the selected candidates, generating offspring.
  4. Evaluate the fitness of the new graph individuals.
  5. Replace the lowest-fitness individuals in the population with the new graph individuals.

3.1 Graph-building algorithm

3.2 Mutation and crossover

3.3 Fitness function

4 Experiment Design

5 Results

6 Conclusions

References

[Aversano *et al.*, 2006] Lerina Aversano, Massimiliano Di Penta, and Kunal Taneja. A genetic programming approach to support the design of service compositions. *International Journal of Computer Systems Science & Engineering*, 21(4):247–254, 2006.

[Bansal *et al.*, 2008] Ajay Bansal, M Brian Blake, Srividya Kona, Steffen Bleul, Thomas Weise, and Michael 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*, pages 351–354. IEEE, 2008.

[da Silva *et al.*, 2014] A.S. da Silva, Hui Ma, and Mengjie Zhang. A graph-based particle swarm optimisation approach to qos-aware web service composition and selection. In *Evolutionary Computation (CEC), 2014 IEEE Congress on*, pages 3127–3134, July 2014.

[Kuster *et al.*, 2008] U Kuster, Birgitta Konig-Ries, and Andreas Krug. Opossum-an online portal to collect and share sws descriptions. In *Semantic Computing, 2008 IEEE International Conference on*, pages 480–481. IEEE, 2008.

[Rodriguez-Mier *et al.*, 2010] Pablo Rodriguez-Mier, Manuel Mucientes, Manuel Lama, and Miguel I Couto. Composition of web services through genetic programming. *Evolutionary Intelligence*, 3(3-4):171–186, 2010.

[Srivastava and Koehler, 2003] Biplav Srivastava and Jana Koehler. Web service composition-current solutions and open problems. In *ICAPS 2003 workshop on Planning for Web Services*, volume 35, pages 28–35, 2003.

ALGORITHM 2. Generating a new candidate graph.

Input : $I, O, relevant$
Output: candidate graph G

$start.outputs \leftarrow \{I\}$;
$end.inputs \leftarrow \{O\}$;
$G.edges \leftarrow \{\}$;
$G.nodes \leftarrow \{start\}$;
$seenNodes \leftarrow \{start\}$;
$currEndInputs \leftarrow \{start.outputs\}$;
$candidateList \leftarrow$ `findCands(`$seenNodes, relevant$`)`;
while $end.inputs \not\sqsubseteq currentEndInputs$ **do**
  $cand \leftarrow candidateList.next()$;
  if $cand \notin seenNodes \wedge cand.inputs \sqsubseteq currEndInputs$ **then**
    `connectNode(`$cand, G$`)`;
    $currEndInputs \leftarrow currEndInputs \cup \{cand.outputs\}$;
    $seenNodes \leftarrow seenNodes \cup \{cand\}$;
    $candidateList \leftarrow candidateList \cup$ `findCands(`$seenNodes, relevant$`)`;
`connectNode(`end, G`)`;
return G;

Procedure `connectNode(`n, G`)`
  $G.nodes \leftarrow G.nodes \cup \{n\}$;
  $inputsToFulfil \leftarrow \{cand.inputs\}$;
  while $|inputsToFulfil| > 0$ **do**
    $graphN \leftarrow G.nodes.next()$;
    if $|n.inputs \sqcap graphN.outputs| > 0$ **then**
      $inputsToFulfil \leftarrow inputsToFulfil - (n.inputs \sqcap graphN.outputs)$;
      $G.edges \leftarrow G.edges \cup \{graphN \rightarrow n\}$;