Automated web service composition using genetic programming

by

Liyuan Xiao

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Iowa State University

Ames, Iowa

2011

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Automated web service composition is a popular research topic because it can largely reduce human efforts as the business increases. This thesis presents a search-based approach to fully automate web service composition which has a high possibility to satisfy user's functional requirements given certain assumptions. The experiment results show that the accuracy of our composition method using Genetic Programming (GP), in terms of the number of times an expected composition can be derived versus the total number of runs can be over 90%. System designers are users of our method. The system designer begins with a set of available atomic services, creates an initial population containing individuals (i.e. solutions) of candidate service compositions, then repeatedly evaluates those individuals by a fitness function and selects better individuals to generate the next population until a satisfactory solution is found or a termination condition is met. In the context of web service composition, our algorithm of genetic programming is highly improved compared to the traditional genetic programming used in web service composition in three ways: 1. We comply with services knowledge rules such as service dependency graph when generating individuals of web service composition in each population, so we can expect that the convergence process and population quality can be improved. 2. We evaluate the generated individuals in each population through black-box testing. The proportion of successful tests is taken into account by evaluating the fitness function value of genetic programming, so that the convergence rate can be more effective. 3. We take cross-over or mutation operation based on the parent individuals' input and output analysis instead of just choosing by probability as typically done in related work. In this way, better children can be generated even under the same parents. The main contributions of this approach include three aspects. First, less information is needed for service composition. That is, we do not need the composition work-flow and the semantic meaning of each atomic web service. Second, we generate web service composition with full automation. Third, we generate the composition with high accuracy owing to the effect of carefully preparing test cases.

Keywords: genetic programming, black-box testing, functional requirements, test cases, services composition

CHAPTER 1. INTRODUCTION

Web service is considered as an independent modular application which can be invoked through internet to complete a specific task. In this thesis, the user can regard web service as a black-box which can produce desired outputs if required inputs can be given. However, for some complex applications, it is impossible to satisfy user's requirements through a single web service. So it became popular to investigate services composition methods to select proper web services from an available web services repository, organize those selected services in a correct way, and test the composed web service based on the user's requirements.

Most of the work on this problem focuses on the QoS optimization goal by generating the web services composition semi-automatically based on work-flow. A small number of researchers try to solve this problem through AI planning. Considering that organizing work-flow is a time-consuming manual process, we proposes a genetic programming based method which can do three steps of selecting, organizing and testing automatically without relying on work-flow. In our approach, a web service composition will be represented as a syntax tree. The leaves of the tree represent atomic services selected from service repository, and the non-terminal nodes in the tree represent the composition-structure patterns which express how services can be composed together. The services repository set is regarded as the terminal set and the composition-structure pattern set is regarded as the function set. We generate the candidate service compositions based on those two sets and update the candidates until we find the best one which has an acceptable fitness value or a predefined number of generations is reached.

Compared to the other related work, our approach performs better as in the following sense. First, we improved the fitness function used in the work of Aversano et al. (2006) and

Mucientes et al. (2009). We do black-box testing to evaluate each individual in each generation using test cases. The proportion of successful test cases is considered as a very important factor in the fitness function. This is because the success rate is a very intuitive and strong factor to evaluate the functionality of a candidate composition. By adding this parameter, the fitness function would be more reliable. The test cases used in the black-box testing can be generated from existing mechanism automatically. Actually some researchers have achieved much on the test cases generation for web service composition based on functional specification. For example, in Lu (1994), a tool called CAT is proposed to produce the executable test cases based on given functional specification. Other than generating test cases automatically for the black-box testing, test cases may be obtained from the historic data or observed data as explained in the Situ environment (Chang et al., 2009). In this thesis, we assume that we have existing test cases to use. Second, when we choose individuals as parents to generate children, we give to weight to the probability for each individual to be chosen by comparing SDG (Gu et al., 2008) and individuals. The individuals which are more consistent with SDG will have higher chance to be chosen. In this way, those candidates which are far away from the optimal ones will have a reduced probability to be chosen. So we have more chance to get better children since better parents are selected. This also guarantees that the black-box testing can be run on the composition candidates and the convergence rate and the population quality can be increased. Third, we iteratively generate and update the candidates automatically. During the candidate generation process, we only consider the input/output of the desired web services composition. The web service composition is regarded as a black-box, so there is no need to consider what work flow should be inside. Fourth, when manipulating the parent candidates, instead of choosing the crossover or mutation operation by probability, we choose the genetic operator by the result of analyzing the input/output of the parents. For the same parent candidates, we can get better children candidates if we take proper genetic operator based on the analysis of the input/output of the parent candidates. Fifth, comparing to Rodriguez-Mier et al. (2010) which is too much rely on the semantic meaning of input/output parameter such as the variable name, our method is more stable when the available atomic services have the same input/output parameter name and type but have very different functionality actually.

Users' requirements can be divided into hard goal and soft goal from the perspective of users and the degree of concern about the software. The hard goals are the issues in the requirements specification which are clearly necessary to be satisfied and have a specific standard to judge whether satisfied or not. The soft goals are those issues short of clear standard to judge and often described as constraints for optimization. From the software itself, the users' requirements can be divided into functional requirements and non-functional requirements. The functional requirements are what the software has to do and the non-functional requirements are how well the software does. From our definition, we can say that the functional requirements are a subset of the hard goals.

The objective is to choose some atomic services from the available service set to compose in a way to fulfill complex requirements. A lot of research in this field has made progress on the non-functional part such as response time, availability, reliability and so on and many can be complementary to our approach. In this thesis, we focus on the user's functional requirements.

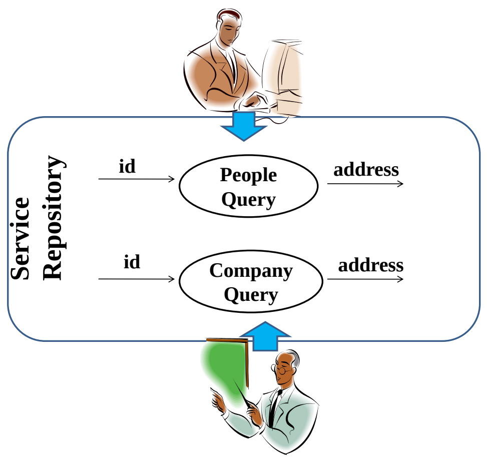
The rest of my thesis is organized as follows. Chapter 2 reviews some related work in this field. Chapter 3 introduces some background about genetic programming and service composition. Chapter 4 gives the details of our technical approach. Chapter 5 describes an analysis about the complexity about this algorithm. Chapter 6 show the results of experiments using our method in the service composition problem and compare our method with the existing method in Rodriguez-Mier et al. (2010). Chapter 7 concludes this thesis and presents some future work.

CHAPTER 2. RELATED WORK

Web service composition can decrease cost by reusing existing web services. This part gives an overview for the existing work in this field.

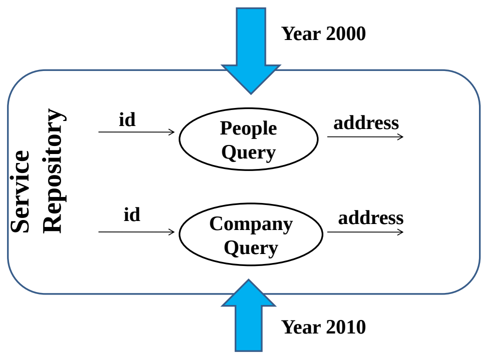
Most research on this topic is limited to the scope of QoS of service composition. Only few researchers proposed some approaches to deal with the functional requirements of the web service composition. Rao and Su (2004) did a survey of automated web service composition methods. In Aversano et al. (2006), Lerina Aversano proposed a method to use genetic programming to support the design of service composition. The approach is to generate work-flow of a business goal using genetic programming to support the service integration activities in the process of finalizing the work-flow. In this work, the work-flow generated can not be used by the user directly. In Mucientes et al. (2009), an approach is proposed to generate the valid structures of composite services through the genetic programming algorithm. In Majithia et al. (2004), a framework is presented to facilitate automated service composition in Service-Oriented Architecture by using semantically-described heterogeneous services. In Rodriguez-Mier et al. (2010), genetic programming is also used to compose web services and several public service repository are used in the experiments. However, the fitness function of the genetic programming depends too much on atomic services' input/output variable name and the composition they generated is short of a reliable accurate. It is very possible that there exists services which have the same input/output name but have totally different functions. Suppose we have two services named PeopleQuery service and CompanyQuery service. The two services may be developed by different programmers as shown in Figure 2.1 , and they used the same input/output name. These two services may be developed by the same programmer but one programmer may have used the same input/output name for both of the two services as shown in Figure 2.2. Service A is used to query people's information and service B is used to query company's information. Both service A and service B use 'id' as input name and use 'address' as output name. In this case, only considering the input/output variable's name and type can not differentiate service compositions sometimes as shown in Figure 2.3. Pablo's method will regard those two compositions as the same one but actually those two service composition are totally doing different tasks.

Figure 2.1 Two Services with Same Input/Output Name but Different Function(1)



In this thesis, we focus on generating web services composition automatically to fulfill user's functional requirements using genetic programming. We also use the black-box testing to evaluate those candidates we generated with automatically generated test cases. Some researchers proposed their mechanism to generate test cases automatically. In Kaschner and Lohmann (2009), the paper proposed an approach to automatically generate test cases for black-box testing to test the web service composition. Those test cases are derived from an operating guideline of a specification and can be used to test conformance of an implementation without inner knowledge of the web services composition. Other references could be found in Lallali et al. (2008). In this modified genetic programming, the result of the black-box testing is taken into account in the fitness function. In this thesis, we assume that we have available test

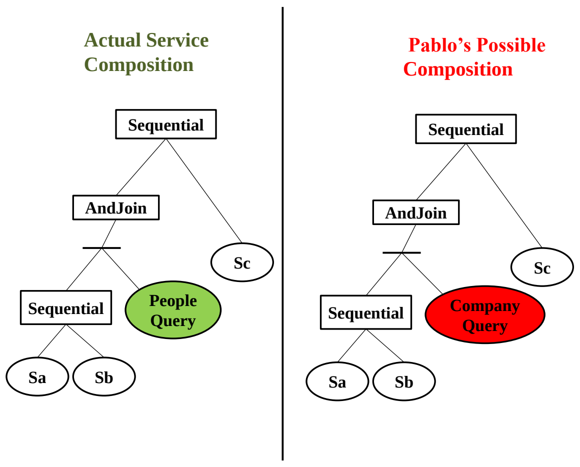
Figure 2.2 Two Services with Same Input/Output Name but Different Function(2)



cases. From the experiment results, based on effective test cases, the composition generated by our method can be above 90% accuracy in terms of meeting the user's goal. During the process of genetic programming, we modify the traditional algorithms by using the knowledge such as service dependency graph between services.

Web service composition is needed in various field and situation. The genetic programming we modified in this thesis is especially useful when we have a lot of data but do not have any clue about the work-flow. For example, in Physics, Biology and Chemistry field, we have a lot of experiment data and available services but we do not know what is the inside logic among those data and services. In this case, genetic programming has advantages to compose the web services because we can compose the web services without knowing the work-flow.

Figure 2.3 Two Compositions which Can Not Be Differentiated by Input/Output Name



CHAPTER 3. BACKGROUND

We modified the genetic programming and applied it to solve service composition problem. In this part, we introduce some basic idea about genetic programming and web service composition.

3.1 Genetic Programming

Genetic programming provides an approach to solve problems by induction (Koza, 1992). The goal of genetic programming is to search the fittest individual computer program in the space of all possible computer programs consisting of functions and terminals (i.e. operations on functions) appropriate to the problem domain. During the process of searching, each individual can be regarded as a syntax tree which is composed of terminal nodes and non-terminal nodes. The terminal nodes are from the function set and the non-terminal nodes are from the operand set, which are defined on the problem domain. A simple example is to find a proper mathematical expression to solve a mathematical problem. We may define the function set as arithmetical operators such as $\{+, -, *, /, sqrt\ \}$ and the terminal sets as some numbers and variables such as $\{1, 2, 3, 6, a, b, c\ \}$ based on the problem domain.

The genetic programming algorithm attempts to simulate natural evolution and selection of a population of possible individuals to search for an optimal or near-optimal solution. Instead of searching the entire search space, it starts with a random initial population of a defined number of individuals. It then uses the fitness function as a guidance to evolve the population towards a better population. Between each update from former population to later population, we apply operations on the individuals such as crossover, mutation and reproduction to generate new individuals which may be added to the new population. The algorithm stops when the optimal solution is found or a defined number of generations has been reached.

With respect to a problem, if we can determine the set of terminals, the set of functions, the fitness measure, the parameters and variables such as the size of the population and the criterion to terminate the processing, then we are able to use the genetic programming to solve this problem.

Genetic programming is used in many kinds of problems. In Pedro et al. (2010), it surveys existing literature about the application of genetic programming to classification like clustering and association discovery and so on. In Koza and Rice (1992), it proposed that using genetic programming to automatically generate a computer program to enable an autonomous mobile robot to perform the task of moving box from the middle of an irregular shaped room to the wall. In Weimer et al. (2009) and Forrest et al. (2009), genetic programming is successfully applied to the problem of software repair.

3.2 Service Composition Structure

To fulfill complex requirements, atomic services may need to be connected by different structures. In Yu et al. (2007), 6 composition structures are identified together with their algorithm to select web services. The 6 composition structures are: Sequential, AND split (fork), XOR split (conditional), Loop, AND join (Merge) and XOR join (Trigger).

In the sequential pattern, one service output is used as input of another service. In the AND split pattern, one service output is used as input of all other services. In the XOR split pattern, one service output is used by one of other possible services depending on the probability of selecting each such service. In the loop pattern, one service output is the input of itself. In the AND join, several service outputs are combined together as the input to another service. In the XOR join, one of several service outputs is used as input of a service depending on its probability of being selected.

CHAPTER 4. GENETIC PROGRAMMING BASED AUTOMATED SERVICE COMPOSITION FRAMEWORK

This section presents our technique for composing services. In this section, our project framework, the principle of using genetic programming to compose services, the detailed algorithm and the fitness function of genetic programming are discussed respectively. Currently, our research is based on two assumptions:

Assumption 1: We have existing test cases which can be used in black-box testing.

Assumption 2: The expected web services composition can be composed from two or more atomic web services among the available services repository. In other words, within the problem domain, we know the available services set for the expected composition.

Based on assumption 1 and assumption 2, our goal is to use genetic programming to compose the desired web service composition with high accuracy meeting user's goal based on the effective test cases.

Our method is specially effective in the problem with a very complicated system but we do not know the inside structure of the system, such as the problem of generating mathematical web service composition which is composed of atomic services with mathematical functions. This kind of problem is also very common in different field, such as Chemistry, Biology and so on. People may get a lot of experiment data for different parameters, and those parameters must have some unknown relationship each other. In this case, our method can take the data as test cases and help them find the regular pattern between those parameters with high accuracy. Aside from the mathematical web services composition, our method also can be used in all kinds of web service composition problem if we have appropriate test cases.

To get test cases for black-box testing in the genetic-programming based composition method, several ways are possible:

1. There are some existing mechanisms to generate test cases automatically from the requirements specification if the user's requirement document is very formal.

2. Historic data and observed data can be used as test cases. For example, from the Situ environment (Chang et al., 2009), we can obtain test cases from observing human's behavior pattern.

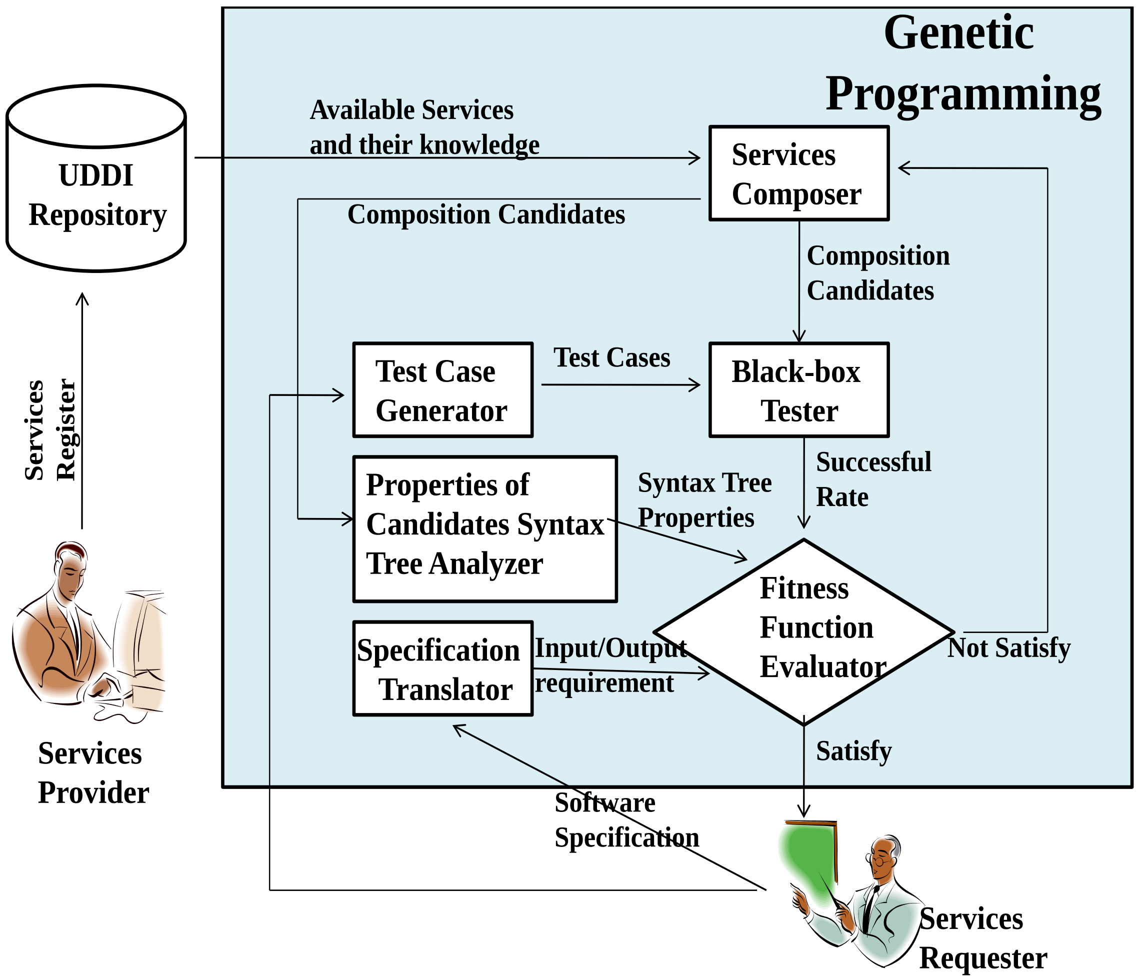
3. We are going to research how to generate effective test cases automatically for web service composition. Some researchers have achieved good results in this field.

About assumption 2, from observing the user behavior patterns and analyzing the problem domain, related services can be inferred . Those services can be added to the set of available services mentioned in this thesis.

4.1 Overview of Framework

The whole framework including both our current work of using genetic programming to compose web services and future work such as test cases generation is given in Figure 4.1. The main part of the framework is the genetic programming used to search web services composition which can satisfy user's input/output requirements. As shown in the figure, the service provider provides available atomic services to UDDI which are registries used in an enterprise to share Web services. The service knowledge can provide service dependency constrains between atomic services such as the service dependency graph (SDG). The services requester gives software specification which contains the input/output requirements. The genetic programming then do all the following steps automatically: select the atomic services, generate initial service composition population, evaluate service composition candidates in each population, and update service composition population. During the process of evaluation, we evaluate the generated service composition candidates by analyzing the following factors: black-box testing results of candidates, the consistency between web service composition candidates' input/output and user's required input/output, and properties of the syntax tree of candidates such as the depth of the tree. In order to automate the whole framework, a mechanism is needed to generate test cases automatically to run black-box testing against the service composition candidates. Some researchers have worked this out, refer to Kaschner and Lohmann (2009). To compare the input/output of composition candidates and input/output provided by users, a translator is needed to transfer the software specification to input/output requirements.

Figure 4.1 Web Service Composition Framework



Following the assumption described above, this thesis focuses on the services selection, testing and fitness evaluation. We assume that we have available test cases and the users can clarify the input they provide and the output they desire. Our test case generation mechanism and requirement specification translator will be built in the future to make the whole automatic framework complete. The main task of this thesis is to enhance the traditional genetic programming algorithm and use fitness function specifically designed by us to generate an acceptable web service composition.

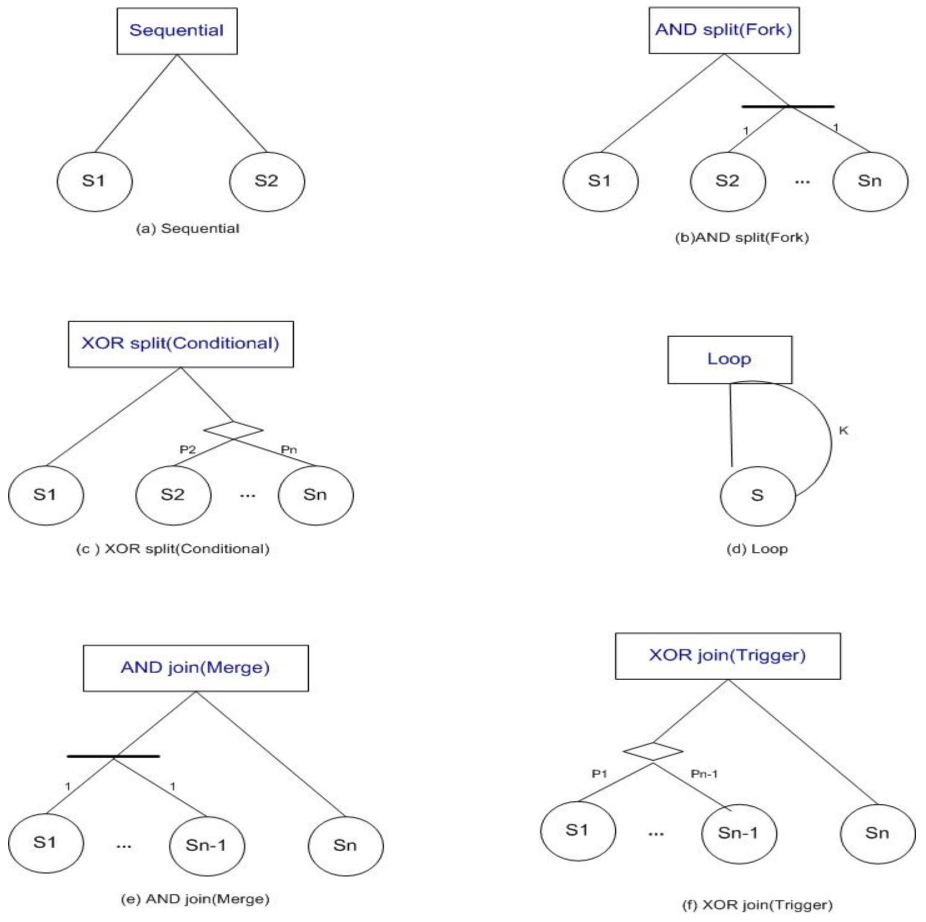
4.2 Research Methodology

The primary focus of our current research allows us to define web service composition as a problem of searching a computer program in search space (all possible service compositions) to satisfy users' input and output requirements. To apply genetic programming method to our problem, we need to figure out the mapping from service composition problem to genetic programming method.

Step 1: Determine the set of terminals in the syntax tree. The genetic programming is operated on all the available services. Suppose we have n available services which are named from S_1 to S_n, the terminal set can be defined as $T = \{S_1, S_2, S_3, \ldots, S_n\}$. Each leaf in the syntax tree is an element from set T.

Step 2: Determine the set of functions. Based on the 6 kinds of composition structure patterns mentioned in section 3.2, we define the function set as F = {sequential, AND split, XOR split, Loop, AND join, XOR join}. Those 6 elements in F are shown in Figure 4.2. The circles represent services and the rectangles represent the functions among those atomic services. Here, two more structure patterns are added from the work of Aversano et al. (2006). Each web service composition candidate is represented as a syntax tree in the genetic programming, where the nonterminal nodes in the tree are from the composition structure patterns. In (a), S_2 is executed after S_1 and S_2 take the S_1's output as input. In (b), all the services from S1 to Sn are executed after S_1 and all of them take S_1's output as input. In (c), only one service will be executed after S_1 and the probability of S_2 to Sn decides which service will be chosen to be executed. In (d), service S takes its own output as the input. The iteration times should not be more than K. In (e), all the services from S_1 to S_{n-1} are executed before Sn. Their output is taken as the input of S_n. In (f), based on the probability, only one service from S_1 to S_{n-1} is taken to be executed before S_n and its output is taken as input of S_n.

Figure 4.2 Web Service Composition Function Set



Step 3: Determine the fitness measure. The fitness measure evaluates the quality of the service composition candidates. Our fitness function is built by considering several aspects: (1) The ratio of the number of success test cases to the number of total test cases which can be a convincing parameter to say a candidate is good or not. (2) The input and output of the candidate syntax tree which can be obtained by analyzing the input and output of each atomic service. (3) The input and output of user's requirements. (4)

The depth and number of leaves of the syntax tree to indicate the complexity of a tree. Detailed explanation of fitness function will be described in section 4.6.

Step 4: Determine the parameters and variables for controlling the algorithm run such as population size M, the number of evolving generations, the run time, and so on. Definitions of all these parameters will be given in section 5.

Step 5: Determine the criteria for terminating genetic programming run. We can stop the genetic programming when a predetermined number of generation is reached or when we find a candidate whose fitness function value reaches a goal value we set.

4.3 Genetic Programming Algorithm

We applied genetic programming by modifying the traditional one. The detailed algorithm is as follows:

1. Randomly choose M individuals as the initial population obeying the SDG constraints.

2. Repeat the following steps until an individual whose fitness value is very close to value 2 is found or a predefined number of generation is reached.

    (a) Conduct the black-box testing for each individual in the current population.

    (b) Calculate the fitness Fit(S_i) of each individual S_i in the current population.

    (c) Do while number of individuals in the temporary population is less than $\lfloor M \rfloor/2$.

        i. Randomly choose a pair of individuals from the current population.

        ii. Remove the pair of individuals from the current population.

        iii. Choose the individual which has a higher value of Fit(S_i) and add it to a temporary generation.

        End While

    (d) Do while the number of the individuals in the temporary population is more than 1

i. Randomly choose a pair of individuals as parents from the temporary population, check the input and output of the individuals, if the two individual's input do not overlap with the user's input then

   Do mutation on the parents to get two children obeying SDG constraints

   Else do crossover operation on them to get two children obeying SDG constraints

ii. Add both parents and generated children into new population.

   End while

(e) If the size of new population m is less than M, then generate $(M - m)$ individuals randomly and add them to the new population

(f) Use the new population to replace the current population.

3. Output the optimal individual or near optimal individual.

Comparing to traditional genetic programming, we improved the algorithm in step 1, step 2(a) and step 2(d). The following sections will show how the algorithm is improved in those steps.

4.4 SDG Matching Check

The Service Dependency Graph (Gu et al., 2008) we used in step 2(a) in section 4.3 is an AND/OR graph showing input/output dependencies among atomic services operations. We utilize it to assign weights to the individuals before choosing them as parents to generate new individuals for next generation. An individual should have high chance to be chosen as parent if no structure is in conflict with the SDG. Otherwise, the individual should have lower chance to be chosen as parents. We use Ci as the weight of the probability of individuals to be chosen before calculating fitness function because the SDG is a very important constraint in services composition. Doing this can help us choose high quality parents in each generation during the process of running genetic programming. In this process, weak parents can be pruned because genetic programming prefers to choose those candidates which obey the SDG constraints better. With better parents in each generation, it is believed that the convergence rate can be improved.

The SDG constraints could be obtained from the service knowledge base provided by the service providers. In addition, based on the atomic services input and output variable type, we also can make some constraints of the service repository and add them to the SDG. For example, if there are n services in our service repository $\{S_1, S_2......S_n\}$, then for each service S_i, only the services which have at least one output type in S_i input type set can be invoked before service S_i. Similarly, only the services which have at least one input type in S_i output type can be invoked after service S_i.

4.5 Selection of Genetic Operators

In algorithm step 2(e) in section 4.3, when we do the crossover or mutation operation, we choose to do mutation on the parents if their input do not overlap with the input provided by users. In this case, doing mutation will have more chance to get better children because the children individual's input will still have no overlap with the user's input if we do crossover. Based the same parents, we can improve their children's quality through choosing a more proper operation during the process of running genetic programming.

We use crossover or mutation to generate new individuals. Example of crossover and mutation are shown in Figure 4.3 and Figure 4.4 respectively. In Figure 4.3 regarding crossover operation, sub-trees or leaves of the parents are exchanged. In Figure 4.4 regarding mutation operation, one sub-trees or single nodes are mutated.

Figure 4.3 Crossover Operation

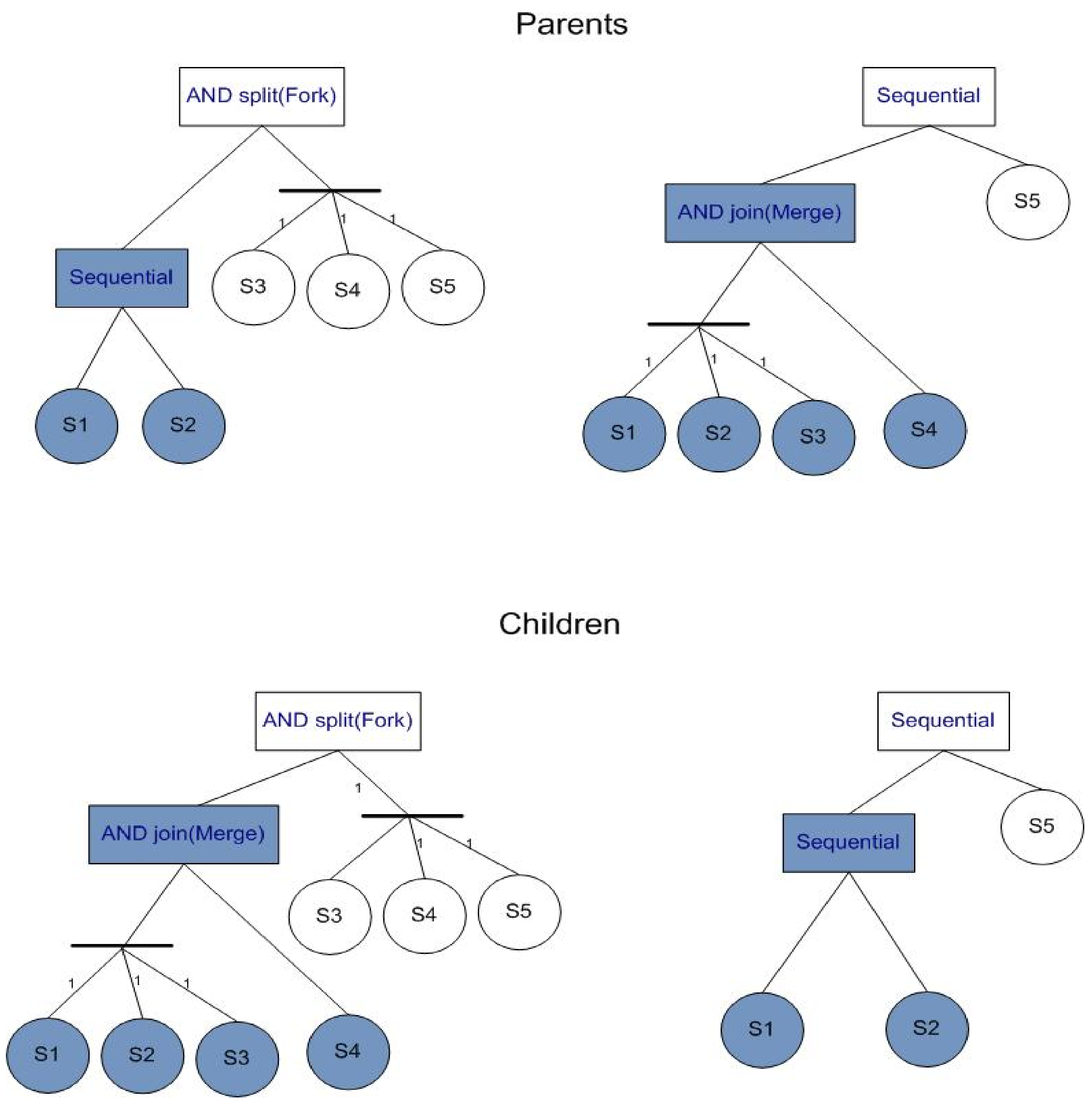
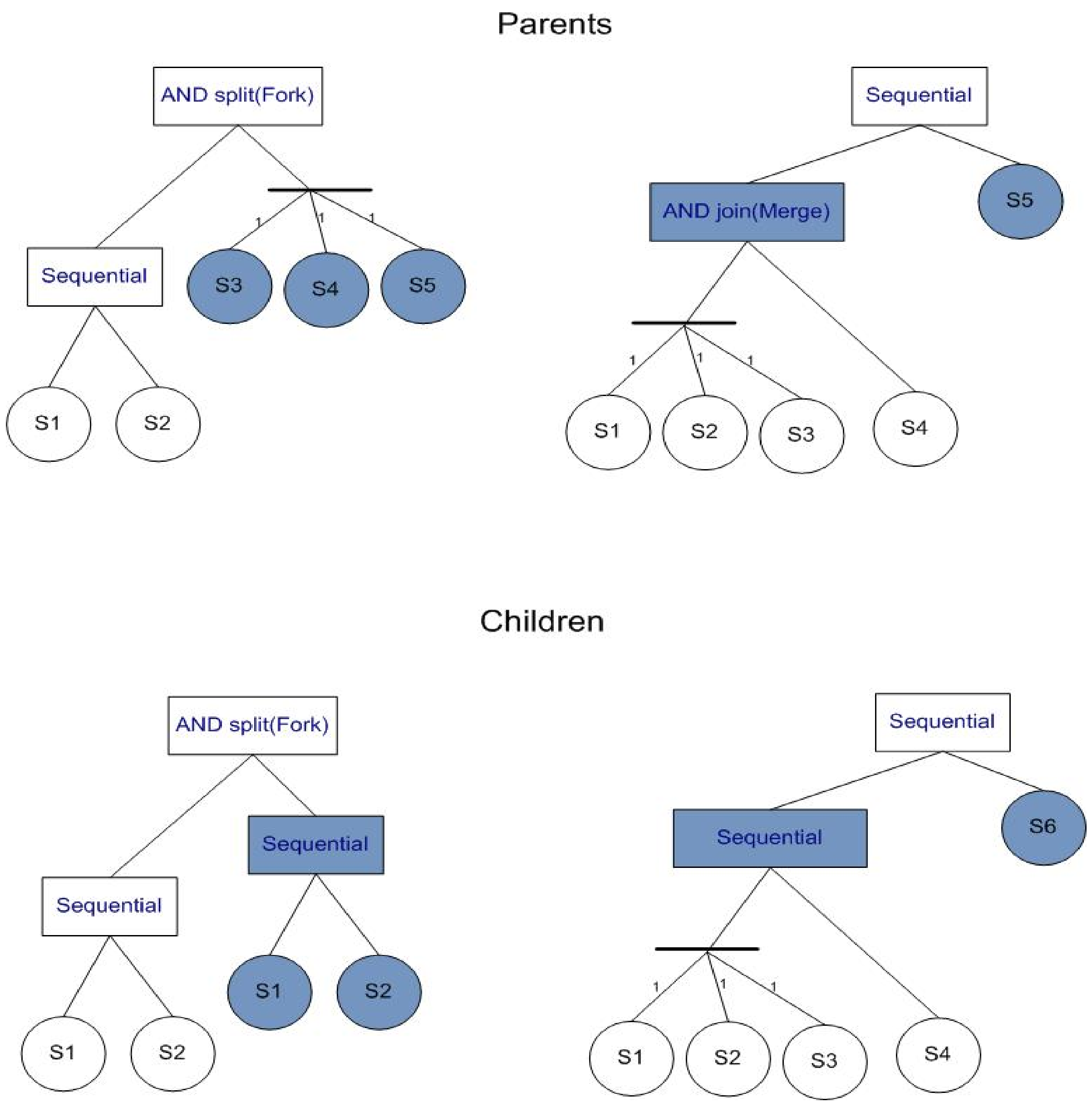


Figure 4.4 Mutation Operation



4.6 Fitness Function

The fitness function is one of the most important components of the proposed technique. In algorithm step 2(b), we do the black-box testing for those individuals and then put the ratio of the number of success test cases to the number of total test cases as one part in the expression of the fitness function. Comparing to the work of Aversano et al. (2006) and Mucientes et al. (2009), the fitness function is enhanced in our method because the result of individual black-box testing is a convincing standard to judge how good an individual's functionality is.

As shown in the framework in Figure 4.1, for service composition individual SC_i of a population, we calculate its fitness by function considering four factors: (1) Success ratio when we do black-box testing on individuals; (2) The overlap between individual's input/output and user's input/output requirements; (3) The depth of the syntax tree for the individual; (4) The number of leaves in the syntax tree of the individuals. We define the fitness function as:

$$\text{Fit}(SC_i) = w_1 * \phi(i) + w_2 * \text{IOM}(i) + w_3 * \{-\text{Treecom}(i)\}, \tag{4.1}$$

where $w_1 > w_2, w_1 + w_2 = 1, 0.0001 \leq w_3 \leq 0.0009$.

1. $\phi(i)$: the success ratio of individual Si resulted from black-box testing. We define it as

$$\phi(i) = \{\text{number of success test cases/number of total test cases}\}. \tag{4.2}$$

   For each individual in each generation, we use the same set of test cases to do black-box testing. Then each individual will have a value of $\phi(i)$ which is the ratio of passed test cases number to failed test cases number. This value indicates how well this individual satisfies the user's functional requirements from the input/output perspective. When the optimal service composition is found, $\phi(i)$ should take value 1, which means that all the test cases passed the black-box testing.

2. IOM(i): Stands for input and output matching of individual SC_i. We define it as

$$\text{IOM}(i) = \text{IM}(i) * \text{OM}(i), \tag{4.3}$$

where IM(i) indicates how well the individual's input and user's input match, OM(i) indicates how well the individual's output and user's output match. The form of IM(i) and OM(i) are

$$\text{IM}(i) = \frac{|I_i \cap I_u|}{|I_i \cap I_u| + |I_i - I_u|} \tag{4.4}$$

$$\text{OM}(i) = \frac{|O_i \cap O_u|}{|O_i \cap O_u| + |O_u - O_i|}, \tag{4.5}$$

where I_i is the set of individual SC_i's input, I_u is the set of input user can provide, O_i is the output set of individual and O_u is the output set user desired. We hope I_i set and I_u set have more overlap so the bigger $|I_i \cap I_u|$ is, the better the individual is. The same principle as O_i and O_u. But at the same time, we do not want set I_i's elements which can not be provided by users, so we let $|I_i - I_u|$ be a penalty of IM(i). Also, we do not want set O_u's elements which can not be outputted by service composition candidate, so we let $|O_u - O_i|$ be a penalty of OM(i).

3. TreeCom(i): It is used to control the complexity of the syntax tree. We define TreeCom(i) as

$$\text{TreeCom}(i) = \text{depth}(i) + |\text{leaves}(i)|, \tag{4.6}$$

where depth(i) is the level numbers of the syntax tree for individual SC_i and |leaves(i)| is the number of leaves in the syntax tree of individual SC_i. We adopt TreeCom(i) to prevent very large and complex syntax tree from being generated which may contain redundant atomic services and service structure patterns.

4. w_1, w_2 and w_3 indicate weights of $\phi(i), \text{IOM}(i), \text{TreeCom}(i)$, respectively. Note that $\phi(i)$ is more intuitive in evaluating the individuals so we let $w_1 > w_2$. We let w_3 be in the range of [0.0001, 0.0009], which is far lower than w_1 and w_2. So only when the syntax

tree contain a large number of services and structure patterns, $w_3 * \mathrm{TreeCom}(i)$ will work. Otherwise, $w_3 * \mathrm{TreeCom}(i)$ is very small and can be ignored. In our case, suppose we can not accept the composition which is too complex such that $w_3 * \mathrm{TreeCom}(i) > 0.1$. Then we can determine the termination condition as $\mathrm{Fit}(SC_i) < 1.9$. In this way, we can control the number of services and the levels of syntax tree so that a large number of redundant services and structure patterns can be avoided.

Clearly, the higher fitness function value is, the better the candidate is. We stop the iteration when we find an individual whose fitness function reaches 1.9 when $\phi(i) = 1, \mathrm{IOM}(i) = 1$ and $\mathrm{TreeCom}(i) \leq 0.1$. At this time, the composition we found have three properties: be able to pass all the test cases; posses a very good input/output matching with the user's requirements and have complexity that we can accept. This composition found by genetic programming is the correct solution with above 90% probability. Or we can specify a number of evolving generation to stop the genetic programming running. In this setting, we may get the near optimal solution instead of the optimal solution.

CHAPTER 5. COMPLEXITY ANALYSIS AND SCALABILITY

5.1 Using Cache to Reduce the Running Time

For a specific available services set and a specific number of test cases, the time to compose the web services can be calculated as Time = G * P * E, where

- G — Represents the number of generations
- P — Represents the population size
- E — Represents the time to get the fitness value for a candidate.

We know that the number of the generations G depends on the running of implementation itself and it can not be known in advance. The population size P represents how many candidates will be in each generation and it can not be defined to be too small as decreasing it would increasing the number of generations G relatively. The only parameter on which we can work is the E. The time to evaluate each candidate E can be calculated as: E = test cases number * Candidate runtime. The test cases number of us is usually predefined. To reduce the evaluation time for each candidate, the only way we have is to decrease the candidate runtime.

Although using the black-box testing result to evaluate the quality of composition candidates is a very intuitive way, it takes long time if we really call the service every time we need it. To reduce the time for waiting for the response from calling services, we use cache to store the input/output set for those services which have been called with the same input before. We identify each service in the available services set with a unique ID. When a service is called on the first time, input and output information will be stored in the cache and if this service needs to be called with the same input next time, we can get the output without really running this service. This can save a lot of time on sending request to services and waiting for response from services. Through the using of cache, we can reduce the genetic programming running time finally. To reduce the running time of the whole genetic programming, we also can do the black-box testing for individuals parallel in each generation. Then the run time could be reduced to G*E.

5.2 Computation Complexity

The computation complexity will be different depends on the expected composition tree. The simplest case is that, in the composition tree, each node at most has one child. That means the expected composition is actually a sequence composition. In this case, the complexity will be $O(nd)$ where n is the number of services in the available service set and d is the number of services needed in this composition. Only one sequence structure pattern is used in this composition.

If considering all 6 structure patterns that may be used in the composition and suppose each non-terminal node has two children, then the complexity would be $n^{2d-1} * m^{2d-2-1}$ where n is the number of the services in the available service set, d is the depth of the composition tree, and m is the number of structure pattern. In our paper, $m = 6$.

CHAPTER 6. EXPERIMENT DESIGN AND RESULTS

6.1 Comparing Our Method with Existing Methods

In Rodriguez-Mier et al. (2010), it is assumed that web services are only characterized by their functional features which are semantically described through ontology. With the assumption that we know all the input/output name of the atomic services in the service repository, taking the value of the semantic matching of the input/output between the composition candidate and the users requirements as the major factor in the fitness function would lead genetic programming to find a proper service composition. It was also proved in Rodriguez-Mier et al. (2010) that a web service composition can be obtained in several public repository in this way.

However, a problem would arise in some situations if we only consider the input/output semantic matching. Consider the situation where we do not know the semantic information of the services in service repository. In this case, we even do not have the information to do the semantic matching analysis. Even that we do have the semantic information for each service in available services set, it is possible that two different variable name may have the same semantic meaning indeed. It is also very possible that two services have exactly the same input name and type and output name and type but do totally different functionality. Take a very simple example as follows: we have service 1 and service 2 which are provided by different providers. For service 1, its function is to do add calculation and its input and output type is float. And the input names are number1 and number 2 and output name is number 3. For services 2, it has the same inputs/outputs name as service 1, but its function is to do subtract calculation. Based on the method in Rodriguez-Mier et al. (2010), assuming that we get a solution of Sequence {Merge (S_a, S_1, S_b), S_c, S_i} in the first genetic programming run, it is totally possible that we get another solution of Sequence {Merge (S_a, S_2, S_b), S_c, S_i}

at the second run because this two compositions would be the same if only considering the input/output semantic matching. However, these two compositions actually perform totally different functionality. Thus, Pablos method can not tell which composition is the one we really needed and expected. In this method, it is very possible to regard a totally wrong composition as the user's expected one.

To avoid this drawback and improve the fitness function, we decide to use the black-box testing to test the service composition candidate and take the black-box testing result in computing our fitness function values. Comparing to the Pablos method, our method also can be used in the case of absence of any semantic information of the atomic services. To show that our method can be used in this case, we run an experiment of mathematical service composition in this section. In this experiment, each service has different function but almost all of them have the same input/output name and type. From the experiment result in section 6.3, we can get the accurate web service composition with above 90% accuracy which is the probability to get the expected web service composition.

What should be mentioned here is that although Pablo's method can not ensure the correctness of the outputted composition, semantic matching is an effective way to lead the genetic programming to find a proper web service composition. Although black-box testing can ensure the correctness of outputted composition with a high probability to succeed, it is very time consuming especially in running genetic programming. So if we have the semantic information of the available services, we can trade off those two factors in our fitness function. Aside from doing input and output type matching as we mentioned in section 4.6, we can also take the semantic matching into account for the second part of our fitness function. Through adjusting the weight of w_1 and w_1 in our fitness function based on the information of the service repository, it is very intuitive that our fitness function will work at least as well as Pablos fitness function since we did consider all the factors in Pablos fitness function.

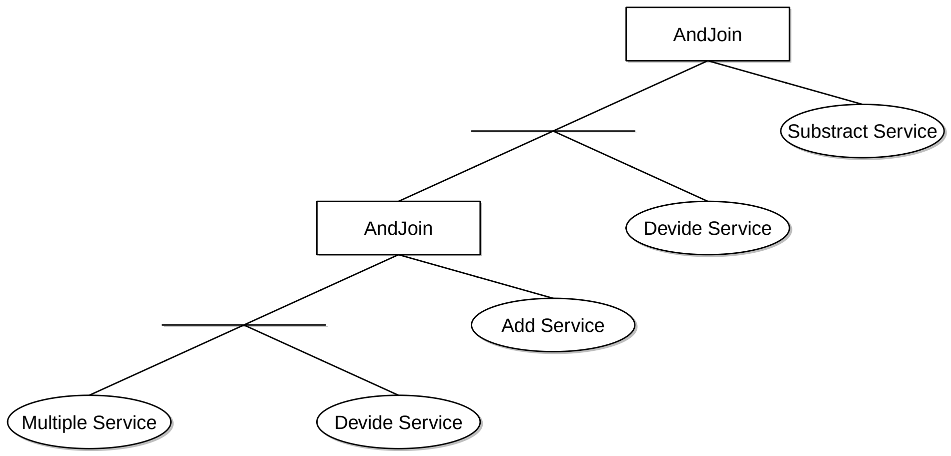
6.2 Experiment Scenarios

We use Apache Axis2/Java as the web service engine and use it to transfer Java class to web services. We use Tomcat as the server to contain web services. Then we wrote some classes in Java to call the services in Tomcat and the only function of the class is to call its corresponding service. In this scenario, the process to compose those classes is equivalently to composing the services in Tomcat. To reduce the parsing time of the services, we can also just compose the java classes to simulate the process of composing the services.

In this experiment, we suppose that the user can provide two inputs and one output, which have float data type as the input data type and output data type. When the input/output data types vary, we can utilize those type information to get some constraints added into the SDG as mentioned in section 4.4. In our experiment, we used the JGAP open source and create our whole experiment based on this open source.

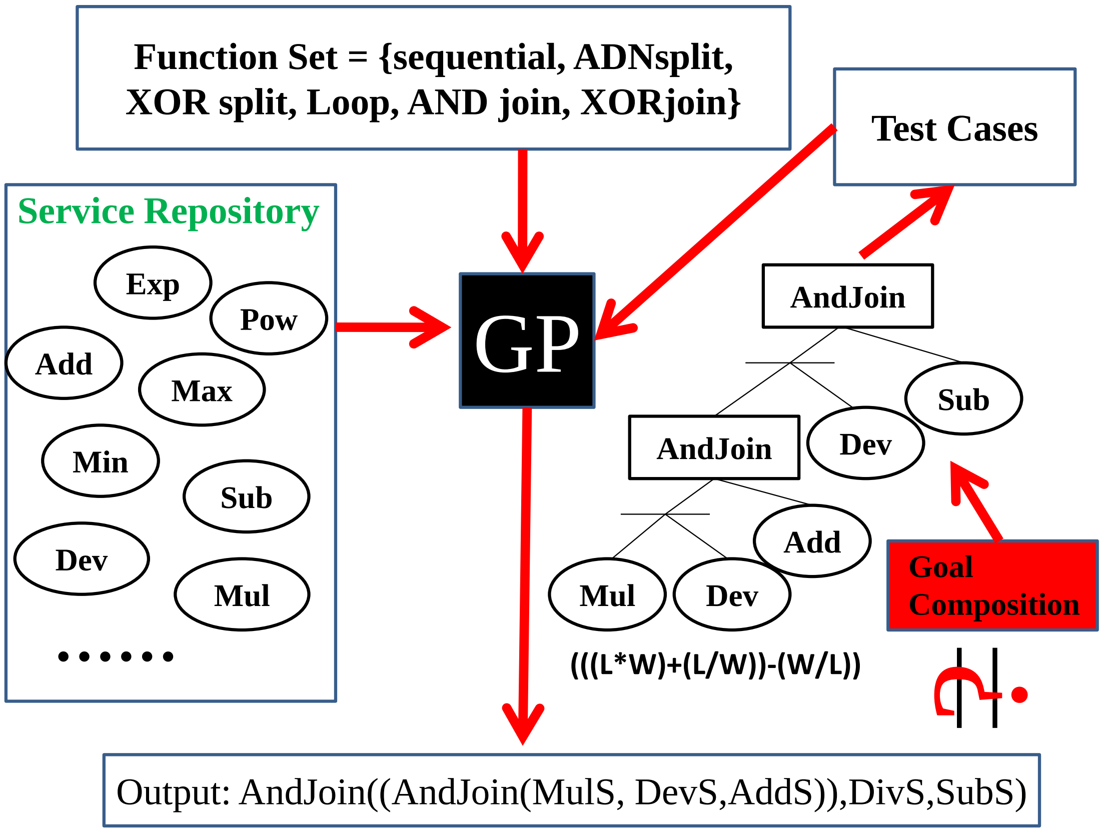
Suppose our goal composition is: AndJoin((AndJoin(multipleService, divideService, addService)), divideService, subtractService). In this experiment, the expected composition composed of 5 services totally and 4 of them are different services. This expected composition is used as oracle in the black-box testing when running genetic programming. It is assumed that the input and output variable names of all the services are unknown. The services needed in the expected service composition are addService, subtractService, multipleServive and divideServce. The size of the service repositories vary from 4 to 20. The syntax tree of the expected composition can be represented as Figure 6.1

Figure 6.1 Expected Web Service Composition



The process of web service composition experiment is presented in Figure 6.2. First, we created the goal composition which is described above. Then we made some test cases based on this goal composition. The genetic programming does not know our expected web service composition. We take the generated test cases, the function set and the service repository as genetic programming's input and run it to search for a composition which is expected to perform the same function as our goal for web service composition. For each set of data setting, the run time and the times of getting the correct services composition are recorded. We use records to calculate the accuracy of genetic programming in section 6.3.

Figure 6.2 Experiment Process


Function Set = {sequential, ADNsplit, XOR split, Loop, AND join, XORjoin}
Test Cases
Service Repository
Exp
Pow
Add
Max
Min
Sub
Dev
Mul
••••••
GP
AndJoin
AndJoin
Dev
Sub
Add
Mul
Dev
Goal Composition
(((L*W)+(L/W))-(W/L))
Output: AndJoin((AndJoin(MulS, DevS,AddS)),DivS,SubS)

6.3 Experiment Execution and Results

The result of one composition case is summarized in Table 6.1, including the number of test cases, the times of running, the population size, the times of finding correct composition and the accuracy (in percentage) under different experiment settings without changing the size of service repository .

To see the relation between the accuracy and the number of test cases, a scatter plot is drawn as shown in Figure 6.3. As expected, they are positively correlated. Using the spline smoothing technique described in Hastie and Tibshirani (1990), we can obtain a few predicted values when test cases are between 600 and 700, which are plotted as unfilled circle dots in Figure 6.3. In conclusion, the accuracy of output from genetic programming is generally above

Table 6.1 Accuracy under Different Experiment Settings

number of cases	times of finding correct composition	accuracy
50	6	30%
100	9	45%
150	7	35%
200	9	45%
250	9	45%
300	12	60%
350	11	55%
400	15	75%
450	16	80%
500	18	90%
550	18	90%
600	19	95%

90% given proper test cases.

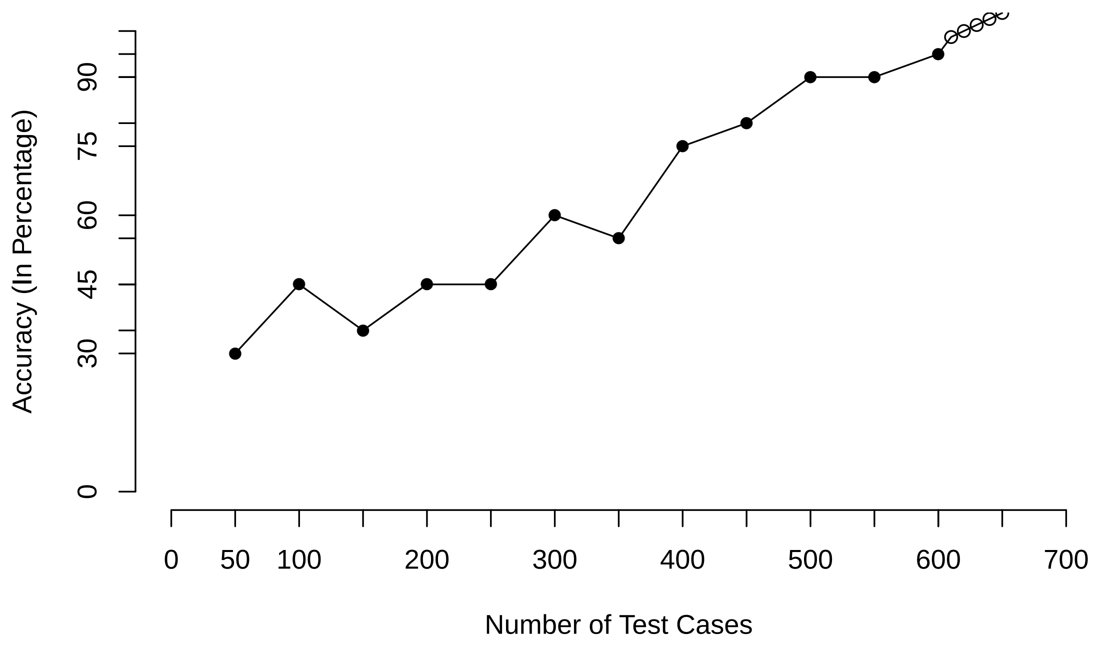
In Table 6.2, the experiment results for more different composition goals are shown. The syntax trees of those composition goals have different depth. They need different number of services and different structure patterns. From the experiment results, we can see that, the accuracy is above 80% in most cases.

Table 6.2 Accuracy for Different Composition Goals

levels of syntax tree	number of test cases	number of needed services	needed structure pattern	accuracy
4	600	5	AndJoin, Sequence	NA[1]
3	600	5	AndJoin	95%
2	600	2	Sequence	95%
3	600	3	Sequence, AndJoin	90%
2	800	3	OrSplit	90%
3	600	4	AndSplit, AndJoin	70%
2	800	3	OrJoin	85%

[1] Program failed to output results after one week of running.

Figure 6.3 Accuracy of Genetic Programming



CHAPTER 7. CONCLUSION AND DISCUSSION

In this thesis, we presented a method applying genetic programming to generate web services composition automatically. We enhanced the original genetic programming algorithm mainly through three ways: utilizing the services knowledge such as services dependency graph to choose parents in each generation; comparing the input/output of candidates with the required input/output from users to guide the choice of applying either crossover or mutation operation; running black-box testing on each candidate in each generation. Therefore, the fitness function is greatly improved in that better parents are chosen and better operations are executed in each generation. Consequently, the convergence rate is also improved during the process of generating new population. Through the experiment, we showed that our genetic programming works well in web services composition in both theory and practice. In addition, our method can produce the expected composition with over 90% accuracy given proper test cases.

We focus primarily on the functional requirements domain in this paper. In the future, we may extend the domain from functional requirements to hard goals described in the introductory part. As mentioned in Chapter 4, we shall do research on automatic test case generation to provide test cases for the black-box testing during the process of genetic programming. Additionally, a good approach for testing the web services composition generated from our method may be needed. Lastly, the whole framework will become a totally automatic system including automatic test cases generation, automatic web services composition generation as well as automatic services composition testing. Note that our method can be used in situations not only when we know the semantic information about the name and type of input and output for the atomic services, but also situations where no information about the name and type is available. In situations where we know the semantic information, the weight of the black-box testing and the weight of the semantic matching and type matching should be considered carefully because those weights may severely influence the fitness function performance depends on the similarity between input/output name and type among the atomic services in the service repository. We will try to find the quantitative relationship among the weights, the similarity and the accuracy. Once we have this quantitative relationship, users can utilize the fitness function more effectively through giving proper weights to different parts in the fitness function according to the similarity between input/output name and types. As long as we work out a good way to generate effective test cases, we will generalize our experiment in this paper to an experiment in real service repository with larger size.

BIBLIOGRAPHY

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

Chang, C. K., Jiang, H., Ming, H., and Oyama, K. (2009). Situ: A situation-theoretic approach to context-aware service evolution. *IEEE Transactions on Services Computing*, 2(3):261–275.

Forrest, S., Nguyen, T. V., Weimer, W., and and, C. L. G. (2009). A genetic programming approach to automated software repair. *GECCO*, pages 947–954.

Gu, Z., Li, J., and Xu, B. (2008). Automatic service composition based on enhanced service dependency graph. *IEEE International Conference on Web Services*, 0:246–253.

Hastie, T. and Tibshirani, R. (1990). *Generalized additive models*. Chapman & Hall/CRC.

Kaschner, K. and Lohmann, N. (2009). Automatic test case generation for interacting services. In Feuerlicht, G. and Lamersdorf, W., editors, *Service-Oriented Computing - ICSOC 2008 Workshops*, volume 5472 of *Lecture Notes in Computer Science*, pages 66–78. Springer Berlin, Heidelberg.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press, 1 edition.

Koza, J. R. and Rice, J. P. (1992). Automatic programming of robots using genetic programming. *Proceedings of the tenth national conference on Artificial intelligence*, pages 194–201.

Lallali, M., Zaidi, F., Cavalli, A., and Hwang, I. (2008). Automatic timed test case generation for web services composition. In *on Web Services, 2008. ECOWS '08. IEEE Sixth European Conference*, pages 53–62.

Lu, P. (1994). Test case generation for specification-based software testing. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '94, page 41. IBM Press.

Majithia, S., Walker, D. W., and Gray, W. A. (2004). A framework for automated service composition in service-oriented architecture. In *In 1st European Semantic Web Symposium*, pages 10–12.

Mucientes, M., Lama, M., and Couto, M. I. (2009). A genetic programming-based algorithm for composing web services. In *2009 Ninth International Conference on Intelligent Systems Design and Applications*, pages 379–384.

Pedro, G., Ventura, S., and Herrera, F. (2010). A survey on the application of genetic programming to classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 40(3):121–144.

Rao, J. and Su, X. (2004). A survey of automated web service composition methods. In *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition, SWSWPC 2004*, pages 43–54.

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

Weimer, W., Nguyen, T. V., Goues, C. L., and Forrest, S. (2009). Automatically finding patches using genetic programming. *ICSE*, pages 364–374.

Yu, T., Zhang, Y., and Lin, K.-J. (2007). Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on The Web*, 1:1–26.