Scalable Automatic Service Composition using Genetic Algorithms

Pouria Roostaei Ali Mehr

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

January 2023

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By:	**Pouria Roostaei Ali Mehr**
Entitled:	**Scalable Automatic Service Composition using Genetic Algorithms**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________ Chair
Dr. Denis Pankratov

______________________ Examiner
Dr. Denis Pankratov

______________________ Examiner
Dr. Nikolaos Tsantalis

______________________ Supervisor
Dr. Joey Paquet

Approved by ______________________
Chair of Department or Graduate Program Director

____
Dr. Mourad Debbabi, Dean of
Gina Cody School of Engineering and Computer Science

Abstract

Scalable Automatic Service Composition using Genetic Algorithms

Pouria Roostaei Ali Mehr

A composition of simple web services, each dedicated to performing a specific sub-task involved, proves to be a more competitive solution than an equivalent atomic web service for a complex requirement comprised of several sub-tasks. Composite services have been extensively researched and perfected in many aspects for over two decades, owing to benefits such as component re-usability, broader options for composition requesters, and the liberty to specialize for component providers. However, most studies in this field must acknowledge that each web service has a limited context in which it can successfully perform its tasks, the boundaries defined by the internal constraints imposed on the service by its providers. The restricted context-spaces of all such component services define the contextual boundaries of the composite service as a whole when used in a composition, making internal constraints an essential factor in composite service functionality. Due to their limited exposure, no systems have yet been proposed on the large-scale solution repository to cater to the specific verification of internal constraints imposed on components of a composite service. In this thesis, we propose a scalable automatic service composition capable of not only automatically constructing context-aware composite web services with internal constraints positioned for optimal resource utilization but also validating the generated compositions on a large-scale solution repository using the General Intensional Programming System (GIPSY) as a time- and cost-efficient simulation/execution environment.

Acknowledgments

First and foremost, I would like to say a special thank you to my supervisor, Dr. Joey Paquet. His support, guidance and insights in this field have made this an inspiring experience for me. I would also like to thank him, the university and NSERC for providing the funding for this research.

I am deeply grateful to Dr. Serguei Mokhov, Dr. Touraj Laleh, and Jyotsana Gupta not only for their excellent research that serves as the foundation for this thesis, but also for sharing their invaluable knowledge on a wide range of topics, simple and complex, that helped me better understand my research problems and design an effective solution.

I consider myself extremely fortunate to have been a part of a fantastic research team where there is never a lack of motivation, encouragement, support, and I'd like to thank everyone involved - Joey, Serguei, Pouria, Vashisht, Rostislav, and Peyman.

I've come a long way since the day I first started preparing for graduate program applications, and the journey, especially the initial steps, would have been much more difficult and might not have happened at all if it hadn't been for the selfless help and encouragement of my old friend, Navid, his lovely wife Golnoush, and my best friends, Pouyan and Setayesh.

There are no words to express my gratitude and love for my wonderful family. The unconditional love, unwavering faith, and boundless patience of my parents and grandmother have shaped me into the person I am today. I dedicate this thesis and all of my work on this research to them, especially my mother and father, who worked tirelessly to make my dreams come true. And for *Babei*, my grandfather who is not among us but I will be his little son forever.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter begins with a thorough explanation of the problem domain and the particular issues that this thesis addresses. Then we discuss the contributions and establish the scope of our research. We conclude by giving a succinct summary of our research problem and then give an introduction to the chapters that go into more detail about it.

1.1 Problem Analysis

A web service is a software application accessed through a web programming interface which can be distributed and identified by a uniform resource identifier (URI) over the Internet. As the Internet progresses more and more towards cloud computing, the evolution of web services presents new trends. First, more and more companies are providing services on the Internet, making the number of available Web services to increase rapidly. Second, semantic information is introduced to web services to describe their functionality in a computer-readable way. Finally, when using web services to build applications, constraints become more important than performance. Since there are so many potential services on the web, it is difficult for users to identify and find the right provider quickly. Therefore, a good service discovery technique is required, and semantic information and service limitations should be considered during the discovery process [1, 2]. Figure 1 depicts the relationships between the entities involved in the global interpretation of the web service domain, as discussed here.

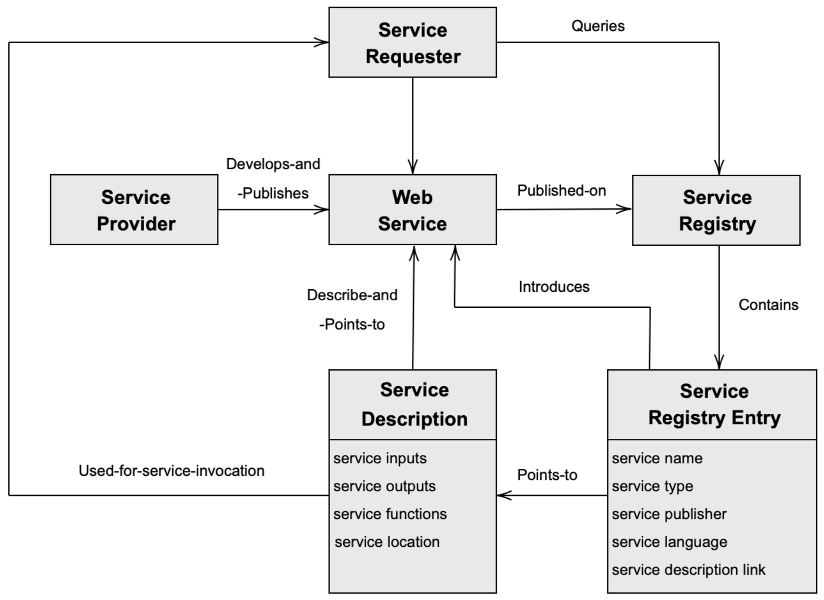


Figure 1: Web Service Domain Model

Web services are usually designed to perform straightforward and specific tasks to ease the discovery process. This simplicity allows the application to reuse it for similar requests. For example, consider a web service that processes credit card payments. Such a service accepts credit card information and payment amount as input and generates payment status (completed/rejected) as output.

In addition, it should facilitate the combination of services with diverse functions to meet user requests when no single service can satisfy them, which is a huge advantage. In the previous example, suppose the output parameters of one service can be used as input parameters of another service. If the service provider uses only one domain, primarily for providing services, then it cannot respond to the requests of customers.

In the example of online shopping (similar to the one depicted in Figure 2), if a web service creates more complex requirements, such as a catalogue, order, payment and product delivery before placing the order, the scope of their potential customers is significantly reduced because such shopping services are only suitable for online shopping stores. Furthermore, if all its components - catalogue, ordering, payment and shipping - are fully compatible with the customer's needs, then the shopping service is appropriate for the customer. However, since the service depends on all components, the entire service is rejected if it cannot perform even one required task. For example, if the service requester needs the service to be able to ship products throughout the North American region, but the service cannot process addresses in Quebec, the entire shopping service will be rejected if a particular user is from Quebec. This is why it is often difficult to find a ready-made integrated service on the web that can meet a complex set of requirements and operational constraints.

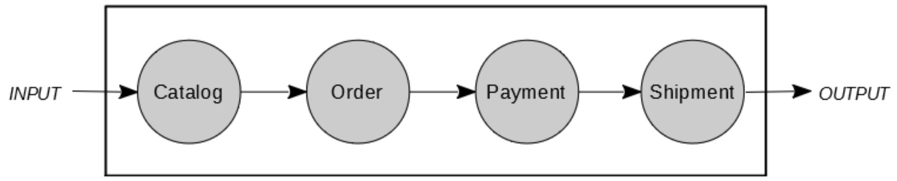


Figure 2: Shopping Service

Therefore, to perform such complex tasks, more specific web services (called *component services*) are selected for each subtask and combined in a workflow to form what is known as a composite web service.

In that case, two services can be connected as a new service with input parameters that are the same as the input parameters of the first single service and output parameters that are the same as the output parameters of the second single one. This new service is called a *composite service*, and the essential services are referred to as the member services or component services. This raises another critical term, namely *service composition*, in the research field of service computing [3].

Automatic web-service composition (AWSC) entails the automated creation of the ideal combination of already-existing web services to accomplish a general objective. There has been many different methods for solving AWSC, and most involve mixing and matching different web service components based on their input, output, and QoS features.

For composite services to be executed properly, certain restrictions and preferences, also known as *constraints*, must be considered. Different kinds of constraints can be defined, such *customer constraints*, which are defined as the preferences and restrictions that customers express. In addition, services are subject to usage limitations and QoS constraints (also known as *service constraints*) that their providers set. The set of constraints for a composite service is derived from the union of all the constraints associated with the services that compose it. When a composite service is executed, its constraints should be verified to ensure proper execution. The verification of constraints for a composite service differs from the constraint verification for a single service. Individual service constraints must only be verified prior to execution of this particular service. However, as each component service is executed, constraints applied to a composite service can be verified during execution. Admittedly, verifying some individual service constraints depends on the values provided by users or other services during the execution of a composite service. So most constraints can in fact only be verified at runtime.

Most existing research on composite web service verification/simulation/execution has focused on validating their Quality of Service (QoS) constraints, functional requirements, or Linear Temporal Logic (LTL) properties. The emphasis of QoS constraint verification researchers has been on ensuring that services maintain certain pre-defined levels of QoS standards, i.e., they provide optimum values for one or more QoS features such as cost, availability, response-time, and reliability. [4–10]. In the meantime, the systems proposed for functional requirement verification have been focused on ensuring that services perform the tasks claimed in their descriptions [4, 8–14].

For instance, when testing a credit card payment service, such systems would seek to ensure that given the credit card information and the amount to be paid, the service would generate a receipt if the card information is valid and the payment amount is within the credit limit. However, these systems neglect to consider that not all credit card brands can be processed by a single service, even if the card details are valid and the credit limit permits the payment. In this case, an error message should be displayed if the credit card information is provided and the service does not recognise it. The systems that have been suggested so far do not account for such scenarios. Because they mainly discussed the quality of performance after execution, which is inappropriate for our research goal. Similarly, LTL-validation solutions ensure that the components of a composite service are executed in the correct order[9, 15]. In the case of the online shopping service depicted in Figure 2, for example, an LTL-verification solution would be primarily concerned with ensuring that the Shipment service is executed only after the Payment service has successfully completed its processing.

In the past, we have used the General Intensional Programming System (GIPSY) [16–18] as a simulation/execution-based environment for verification and validation of constraint- and context-aware composite web services to cater to the specific verification and validation of constraints imposed on component atomic services proposed by Gupta [3] based on the research of Laleh et al. [19–23]. According to Gupta and Laleh et al research, each element of an execution context is a name-value pair. Based on this, the execution context of a service is defined as the collection of all its input parameters and the values assigned to them during the service call. While the service provider specifies these parameters as part of the service's definition for an atomic service, the execution context for a composite service - created in response to a composition request - is viewed as a collection of the input parameters indicated as part of the request for execution of the entire composite service, i.e., the input parameters whose values can be provided by the customer for whom the composite service is aimed.

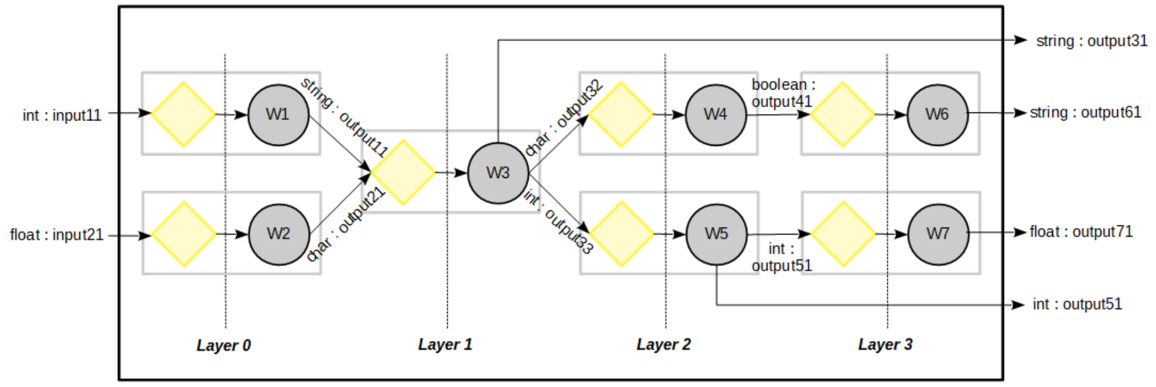


Figure 3: Constraint-and Context-Aware Composite Web Service example.

After our investigation of GIPSY as a simulation/execution-based environment for verification and validation of constraint- and context-aware composite web services

[3] on a large scale, the graph grows exponentially in the forward expansion phase, which turns into another problem in terms of failure to generate any solution for the user's request. This problem is very challenging in the following aspects:

The problem itself, as well as our solution, are significant in scale. For instance, for the constraint-aware Web service selection problem addressed in this thesis, the challenge is selecting a web service for each task involved in a composite web service to generate an execution plan for the composite service. If a composite service is composed of 1000 web services, each with 10 output variables for each web service (a.k.a. *fan-out*) leads to a solution space in the range of 10^{1000}. Clearly, the choice of the best web service among so many potential options can easily become computationally intractable. In this scenario, the experiment on the current version of the service composition algorithm uses an exponential amount of space as the parameters increase. This problem leads to failure in the generation of an execution plan before any one complete solution can be found. Figure 4 depicts a graph with large fan-out branches as an example.

This research investigates how to use genetic algorithms (GA) to address context and constraint-aware service composition problems. GAs are inspired by the natural evolution process and work on natural principles such as inheritance, mutation, selection, and crossover to generate satisfaction. However, they do not always provide optimal solutions to optimization and search problems. Nevertheless, GAs have been successfully applied to complex, large-scale, constrained, and multi-objective optimization problems in a variety of problem domains (e.g., deep learning [24], QoS-aware IoT services composition [25], wireless sensor networks [26], and so on). These positive experiences influenced our decision to use GAs to solve the Constraint-Aware Web Service Composition problem, frequently described as a complex, large-scale, constrained, and multi-objective optimization problem.

In order to address the gaps and problem discussed in this section about web service composition, we propose genetic algorithms for Constraint-Aware Web Services on large scale solution space using General Intensional Programming System (GIPSY) in this thesis.

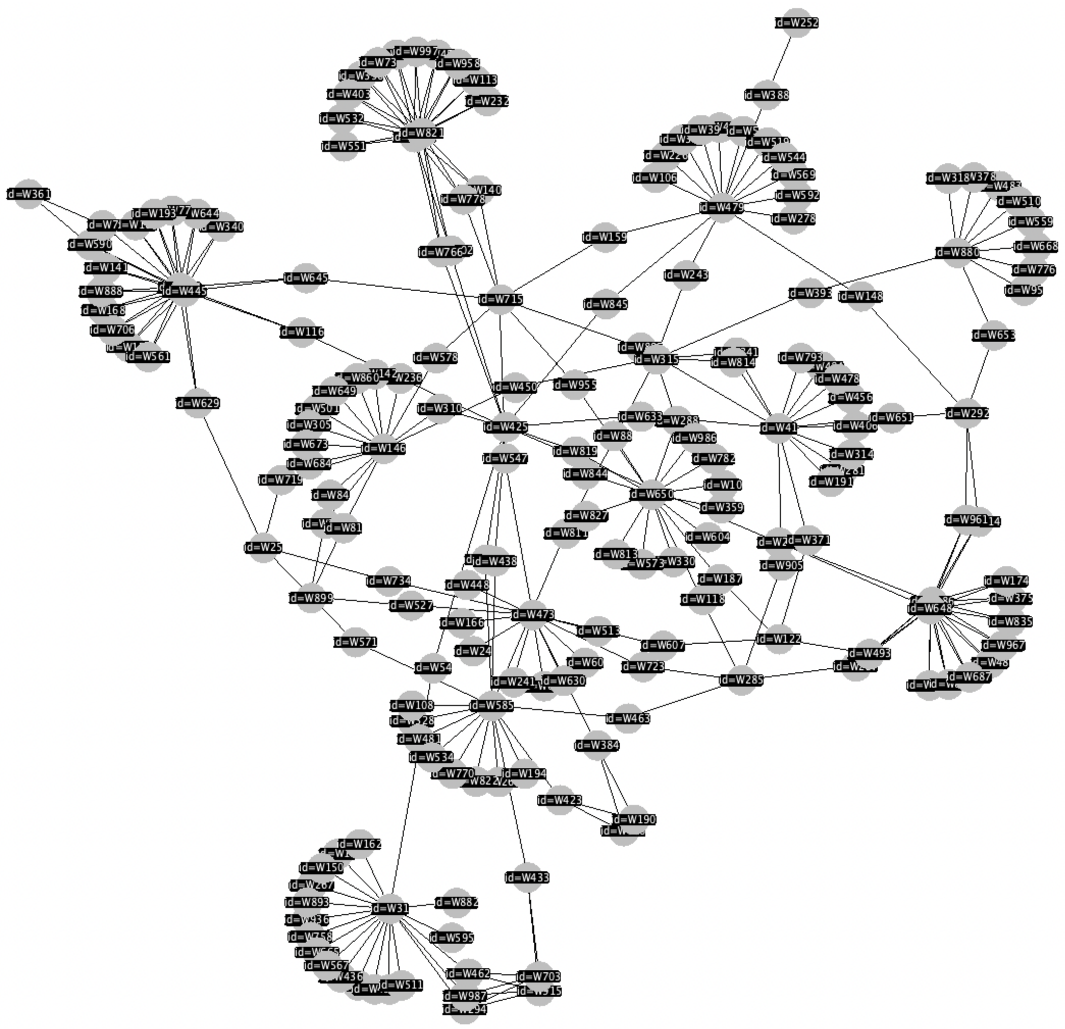


Figure 4: Example of a graph with large fan-out branches.

1.2 Motivation

One topic that makes us focus on web services is the high cost of maintaining and using various web services. For example, Amazon does not have an exclusive need to use all its web services simultaneously, which can be expensive. However, that means it can reduce this cost using pre-designed plans. For example, when a user gets to the payment portal, it can use different options such as PayPal, Visa, MasterCard or other cards. This feature allows providers to prevent services they do not need and reduce maintenance costs. This approach reduces the cost of payment and also reduces the cost of resource consumption. Because of these reasons, it has become popular among cloud provider companies. For example, AWS Compute Optimizer, Kubecost, Datadog [27–29], suggests the optimal computational resources for workloads to reduce costs and enhance performance by analyzing historical utilization metrics with machine learning. Also, the Istio [30] (owned by Google) introduces the practical, real-life expected usage scenarios for service composition. Istio makes it simple to build a deployed service network that includes rich routing, load balancing, service-to-service authentication, monitoring, and more - all without requiring changes to the application code. Istio aims to provide these advantages with minimal resource overhead and support large meshes with high request rates while adding minimal latency. For example, the Istio load tests mesh comprises 1000 services and 2000 sidecars, totalling 70,000 mesh-wide requests per second. The Envoy proxy consumes 0.35 vCPU and 40 MB of memory per 1000 requests per second that pass through it. The Envoy proxy increases the 90th percentile latency by 2.65 milliseconds. However, overprovisioning and underprovisioning computing can result in unnecessary infrastructure costs and poor application performance. However, we restrict the use of these technologies because of the high price of services and their limitations in research studies.

So, we understood that the constraint-aware web service composition has become popular, and according to our study, we supposed that there is much room for development and research on this issue. The research done by Laleh [19] and followed by a corresponding execution/simulation implementation by Gupta [3] was a proposed solution for this problem. However, when we researched their approach, we realised that there is some significant issue when the problem scaled for a higher number of services in a composite service.

In the forward expansion phase, the graph grows exponentially, which causes the failure to generate any solution for the user's request. To avoid this problem, we have substituted the global search algorithm (a.k.a. a *brute-force method*) with a local search. This research aims not to reach all the answers but at least one answer according to the technique we use compared to the existing techniques.

A composite web service's abstract specification indicates that it is a workflow process. So how can we decide which web service to use to complete each of the tasks in the abstract specification while accommodating constraints imposed on the chosen web service implementation to verify and validate the correctness of the composite service?

This problem is a typical combinatorial optimization problem. There are an exponentially growing number of possible combinations of web service implementations for a composite web service as the number of tasks included in the composite service and the number of web service implementations for each task rise. Therefore, finding an optimal solution and execution plan for a composite service is challenging. It may also be very challenging to develop a feasible solution due to the complex restrictions on inter-service dependencies. Finding a feasible and satisfactory solution requires web service selection and algorithms. However, this solution might not be the best for the composite service in a reasonable time. This leads us to the ***first goal*** of our research: ***"Formally define the new concept of operational service composition mechanism and use it to formalize the problem of constraint-aware service composition".*** For this thesis, we assume service requesters have constraints and object to the internal constraints imposed on their requested composite services.

After the composition engine has assembled a suitable composite, it must subject the service to some basic behavioural competency tests, such as the ones listed below, before it can be proposed to the service requester as a viable solution:

- Component services should generate outputs following their interfaces and accept inputs (whether generated by the user or other component services in the composition).

- The values of the requested outputs for a valid set of user inputs fed to the composite service should fall within the expected range. This is how component services should work together cohesively as a single unit.

- Internal restrictions should be properly enforced where applicable. For example, when a contextual value is discovered to violate a restriction, the composite service's execution should be prevented.

The ***second goal*** of this thesis is: ***"To scale the solution scope for the execution context of services and the restrictions/constraints imposed on them to generate at least one solution, if possible, to any valid composition request".*** This study aims to determine efficiency of the proposed algorithm on large scale solution space. Different test instances are built based on the statistical factors to examine effect of scalability parameters . Finally, the new genetic algorithm was tested on these test instances to provide precise results about the algorithm's strengths and weaknesses.

Hence, the ***third goal*** of our research is: ***"To enable our verification system to simulate and execute context- and constraint-aware composite web services for large scale studies".*** As stated in Section 1.1, there is a system for verifying and validating internally-constrained composite web services. Moreover, the proposed algorithm resolved the mentioned problem and prevented the whole simulation/execution from failing during web services composition phase.

1.3 Thesis Contributions

In order to achieve the objectives listed in Section 1.2, we principally want to contribute the following knowledge to the study of web service composition and verification:

1. ***A generic optimized operational constraint-aware service composition mechanism:*** Only a few research teams have sufficiently investigated the composition of internally-constrained services, as mentioned in Section 1.1. This thesis is based on the research of Laleh et al. [19–23] and following execution/simulation by Gupta [3] because they not only offer a formal model for composite services that are aware of constraints but also incorporate a novel constraint-adjustment method into their service composition. As a result, we proposed a genetic algorithm to solve and improve their research methods' limitations and problems on large-scale context/constraint-aware composite services using gipsy studies.

As our first contribution, we design the genetic algorithm for web service composition which helps to optimize search space to minimize the processing effort spent on generating composite service and unnecessary validation checks, and prevent errors caused by memory leaks. These optimizations are represented in the implementation of our algorithm, which we designed as an independent and generic application with an extensible multi-modal input system capable of composing constraint-aware solutions for any valid composition request and set of available atomic services. Chapter 3 contains complete details on this contribution and the necessary explanation of the relevant features of Laleh's research.

2. ***A dataflow execution model that is inherently concurrent and is designed for use with constraint-aware composite services:*** As stated in Section 1.1, we used GIPSY as composite web services simulation and execution environment. It is necessary to convert composite services intended for GIPSY execution into Lucid programs because GIPSY is a system exclusively used for the compilation and execution of LUCID programs. Furthermore, programs written in LUCID are fundamentally formalized textual representations of dataflow networks because it is a dataflow programming language. Consequently, a Lucid program is transformed into a virtual dataflow network of parallel-processing components called filters when executed. For example, if a LUCID program were to stand in for a constraint-aware composite service, its corresponding dataflow network would be composed of concurrently executing component service filters encased in wrappers that served as internal-constraint-verification layers. This concurrency, which is an inherent property of the LUCID execution model and does not require the programmer to launch, synchronize, or manage threads, eliminates the possibility of thread-mismanagement-related errors.

   For this reason, as our second contribution, we converted the web service composition generated by proposed algorithm to be consumable by Gupta's simulation/execution of constraint-aware composite services.

3. ***GIPSY is a composite service verification system that is efficient, aware of its context and constraints, and based on simulation and execution on large scale datasets:*** There are several advantages to utilizing a LUCID/GIPSY combination solution to simulate and execute composite web services. As an intensional programming language, LUCID, unlike other existing composition/verification systems, allows for the effortless incorporation of contextual elements into its programs, while its "whenever" construct enables the clear and straightforward definition of service constraints. At the same time, GIPSY, which is an execution environment for Lucid, is conveniently transforming the programmatic version of a composite service into a context- and constraint-aware dataflow network of component services. These component services have an inherent concurrency and a virtual nature, which results in a minimal and efficient consumption of resources (see Contribution 2). Additionally, GIPSY's deductive, demand-driven approach to execution, along with its warehouse unit that is capable of storing and being queried for execution results paired with the specific context in which they were achieved, significantly reduces the overall amount of time, effort, and cost spent on the simulation and execution of composite service composites. In addition, in Section 1.1, we discuss the need for simulation and execution capabilities in a verification system for composite web services and note that Gupta's research [3] is the only one to offer a dual-mode system for constraint-aware services, lacking in large-scale datasets. A Lucid dialect comprised of Java and Lucid constructs, makes it possible to simulate (using Java methods to emulate service definitions) and execute (by replacing mock definitions with links to actual service implementations) composite services without difficulty within the same system.

   As a result, as our third contribution, we used the Lucid/GIPSY system as a composite service verification solution capable of testing whether the internal constraints placed on component services are correctly verified at their optimal locations within a composition plan as defined by genetic algorithm (discussed in Chapter 3). Furthermore, it provides composite service execution statistics to ensure no demands are generated for component services guarded by internal constraints that fail verification and ensures results for duplicate demands are fetched from the GIPSY warehouse instead of being computed each time explicitly and assesses the improvement in simulation/execution time and cost efficiency. In addition, we investigate those aspects of the GIPSY architecture that are pertinent to its use as a verification system, present a comprehensive analysis of the solution's background, including all related concepts, and examine the advantages of employing the proposed solution in greater depth. Chapter 2, Chapter 3, and Chapter 4 contain in-depth discussions of this contribution and its limitations and evaluation.

1.4 Thesis Scope

The main goal of this thesis is to present a scalable system for verifying and validating context- and internal-constraint-aware composite web services. Since this subject has not been addressed in any prior research, it was our obligation to investigate all of its complexities and include strategies for handling them in our suggested solution. However, it would have been challenging to handle a wide range of criteria in sufficient depth given the time and resources we had at our disposal, and this could have led to an insufficient overall solution. As a result, for this thesis, we decided to concentrate on achieving a restricted but clearly defined set of goals to the best of our abilities, as mentioned in Section 1.2. The following concerns or features of web service composition and verification are outside the scope of this thesis:

- In this research, we do not evaluate the QoS as our computational factor (details about fitness function discussed in Chapter 3), and the main focus is on the proposed evaluation method for internal constraints of web services and their dependencies.

- Although Laleh's research offers an algorithm for all possible solutions inside the dataset, our experiments show that the algorithm was not able to produce any plans in some requests on large-scale datasets. Therefore, our primary focus is to generate at least one solution to overcome this failure. However, this method can only determine the number of generated solutions. Thus, generating the total solutions inside the dataset is unnecessary but might need to be discovered.

- The chromosomes' length is fixed and will be initialized at the initialization phase based on the dataset's statistical information. Therefore, dynamic length has not been used in this algorithm due to the difficulty of evaluating each population.

- Our proposed genetic algorithm is not parallel, Although it can provide considerable gains in scalability and performance.

1.5 Research Methodology

As shown in Figure 5, our standard operating procedure for validating composite web services that are aware of their contexts from within their environments is shown here. The procedure is as follows:

- A composition request specifying the inputs provided by and outputs required by the service requester, as well as a set of atomic services (with or without internal constraints) available for assembly into a workflow, are fed to our genetic composition service composition mechanism (see Contribution 1).

- The composition application generates one or more constraint-aware composite services as probable solutions to the given composition problem based on the problem's validity and solvability using the available atomic services. The requester is notified of invalid or unsolvable problems.

- Then, if any genetic composite services are made for the request, they can all be sent to our service translator application, which turns them into equivalent

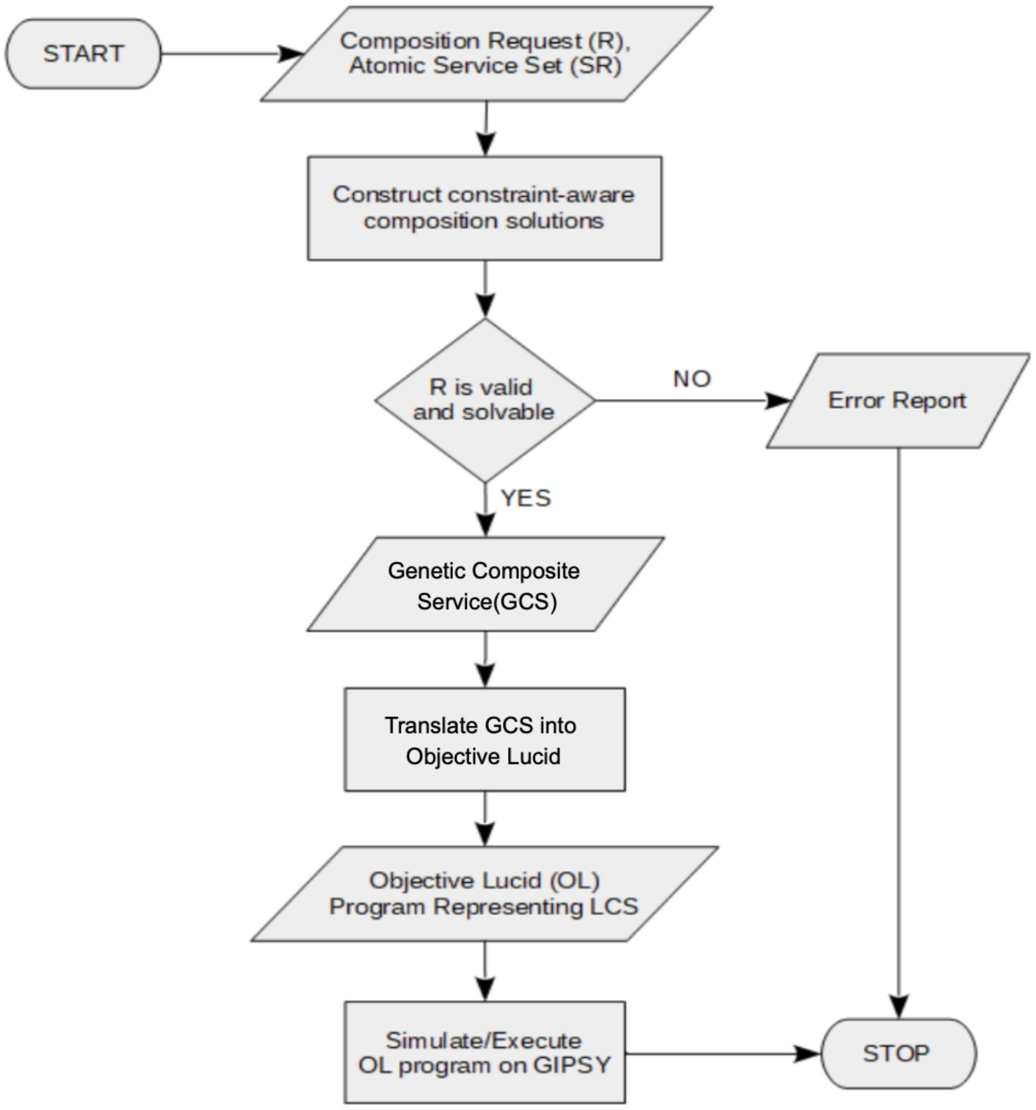

START
Composition Request (R), Atomic Service Set (SR)
Construct constraint-aware composition solutions
R is valid and solvable
NO
Error Report
YES
Genetic Composite Service(GCS)
Translate GCS into Objective Lucid
Objective Lucid (OL) Program Representing LCS
Simulate/Execute OL program on GIPSY
STOP

Figure 5: Research Methodology

Objective Lucid programs that GIPSY can run.

- The Java definitions of the component services that comprise an Objective Lucid program can be generated by the translator to emulate the behaviour described by their corresponding descriptions, or they can be sourced from their providers.

1.6 Thesis Outline

The following chapters' objectives are summarised below:

Chapter 2 provides a comprehensive discussion of all pertinent aspects of the Lucid/GIPSY verification system necessary for thoroughly comprehending the proposed solution. It also examines other related research conducted in the field and compares it to the methodology described in this thesis.

Chapter 3 describes the genetic service composition model and method on which this thesis is based. It also describes the composition application's architecture, usage, and other distinguishing characteristics.

Chapter 4 describes the tests performed to evaluate our proposed verification solution on large-scale datasets (focusing on service composition and translation units). The results and inferences are drawn from them.

Chapter 5 concludes the thesis by arguing that we have in fact met all the objectives that were stated in this chapter. We also identify the solution's limitations and improvements that must be addressed in future works.

1.7 Summary

We still find a gap in the verification and validation of context- and internal-constraint-aware composite web services on large-scale datasets, despite the significant amount of research that has been carried out up to this point in the field of web service composition. To address this deficiency, we propose a simulation/execution-based solution to the verification problem using a combination of Lucid and GIPSY. In this chapter, we have identified our specific research objectives and the methodology to achieve them. In the following chapter, we explore the unique features of the Lucid/GIPSY system and compare it to other related research works to facilitate a better understanding of our proposed solution and support our rationale for employing it for context and internal-constraint-aware composite service validation.

Chapter 2

Background

For a simulation/execution-based verification system for context- and internal-constraint-aware composite web services, we suggest using a Lucid/GIPSY combination in Chapter 1. This chapter explains the fundamental concepts, characteristics, and architectural features of the GIPSY environment and the Lucid programming language that are crucial to fully comprehend our proposed solution. Additionally, it is accountable for the solution having an advantage over the other studies carried out up to this point in composite web service verification, simulation, and execution. As in this thesis we propose to use Genetic Algorithms (GA) to solve the scalability problem of the service composition algorithm proposed by Laleh and implemented by Gupta, we also introduce here the notions of GA involved in our solution.

2.1 Lucid Programming Language

The distinctive program structure, programming constructs, and execution model of the intensional [31] and dataflow [32] programming language Lucid make it an ideal choice for effectively representing and executing constraint- and context-aware composite web services. This section begins with definitions of the fundamental concepts underlying these distinguishing features of Lucid, followed by an explanation of the features and the composite web service domain's advantages.

2.1.1 Intensional Logic

Intensional programming stems from intensional logic, a mathematics subfield that describes entities whose evaluation changes depending on the context in which they are evaluated [33, 34]. It was developed as a method for formally describing the meaning of natural languages, considering that a sentence can be interpreted differently depending on the context in which it is used, the audience for whom it is intended and so on. In other words, a sentence can have a different meaning depending on the context in which it is used, making it a context-dependent entity. Consider the following expression as an example:

E: The temperature is below the freezing point.

Suppose we do not specify the precise date on which the temperature is referred to in the expression is recorded as well as the specific city. In that case, the meaning of the preceding expression will continue to be ambiguous. Therefore, interpreting expression E requires knowledge of the *Date* and *City* factors. In the intensional branch of mathematical logic, these variables that affect the interpretation or evaluation of an expression are known as dimensions. Depending on the type of information it represents, a dimension can assume a variety of different values (see Table 2 for sample dimension values). When each of an expression's dimension names is paired with one of its possible values, the resulting set of dimension names and values is called a context or possible world in which the expression can be evaluated. The context-space for an expression is a collection of the different possible worlds in which the expression can be evaluated (possibly leading to a unique result in each of them). A context space can be described as one-dimensional or multi-dimensional depending on its number of dimensions; in theory, it is even possible for a context space to have an infinite number of dimensions. The intension of an expression refers to the relationship between the contexts of an expression and its calculated values. The extension of an expression refers to the set of all specific values of an expression's intension that correspond to any particular context.

Using the terminology introduced in the previous paragraph, we can deduce that

Table 1: Extension for Temperature in Expression E

City / Date	**Montreal**	**Toronto**	**Ottawa**	**...**
01/01/2022	-10	-5	0	...
02/01/2022	-9	-5	-1	...
03/01/2022	-7	-4	-2	...
...	...	...	...	...

Table 2: Intension of Expression E

City / Date	**Montreal**	**Toronto**	**Ottawa**	**...**
01/01/2022	*true*	*true*	*false*	...
02/01/2022	*true*	*true*	*true*	...
03/01/2022	*true*	*true*	*true*	...
...	...	...	...	...

the meaning of expression E is a function in $(D \times C) \rightarrow B$ where D and C are the sets of values that can be assigned to the Date and City dimensions, and B is the set of boolean values that can be attained for each combination of Date-City values. Table 3 contains a sample mapping of this function (i.e., E's extension). E's extension is determined by the extension of the temperature records for each given city on each respective date (presented in Table 2). In this instance, the context for the first recorded temperature would be: *{Date : 01/01/2022, City : Montreal}* whereas the *extension* of *temperature* in that *context* would be -10, and the extension of expression E would be true. These contexts listed in the two tables would comprise the two-dimensional context space for both temperature and expression E.

2.1.2 Dataflow Networks

The dataflow execution model is the foundation of the generally accepted semantic model that is used to describe Lucid programs. This means that when a Lucid program is being run, it is interpreted as a dataflow network. In order to gain an understanding of this execution model, it is necessary to have a comprehensive understanding of dataflow networks in general. Channels serve as the connecting links in a dataflow network, which consists of filters. When data flows through a network, it passes through a series of filters, each representing a processing unit representing a function that can transform the data elements from one form to another, from input to output. The term "channel" or "transition" is used throughout the network to refer to a stream that connects two filters and acts as a conduit for transferring data elements from one filter to another. The primary advantage of a dataflow network is its inherent concurrency, which enables its components to perform computations concurrently. In addition, each of these components is a black box, which means that it does not interact with any of the other components in the network other than when it receives inputs from those components or sends outputs to those components. Each filter's inner processing is completely concealed and shielded from the side effects of other filters. Consequently, this makes it possible for the functions to keep their referential transparency, which means they can guarantee that the values of their outputs are entirely dependent on the values of their inputs. This implies that for a given set of inputs, a filter produces the same output each time they are processed, regardless of its previous results or the state of the other filters in the network [32,35].

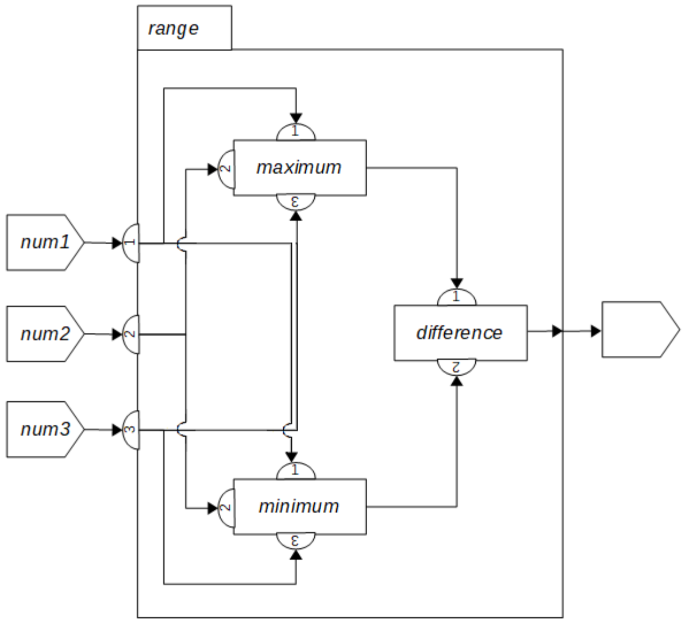


Figure 6: Dataflow Graph for the *range* Program Shown in Listing 2.1

Consider, for instance, the network of dataflow depicted in Figure 4. The objective of the network is to compute the range (i.e., the difference between the maximum and minimum) of the three input numbers, *num*1, *num*2, and *num*3. In the network, each *maximum* and *minimum* filter has three entry points (one for each of the three input nodes) and one exit point (where the calculated output is put into the data stream going to the difference filter). On the other hand, the difference filter only has two entry points (one for each of the other filters) and one exit point, which sends the final result to the network's output node. The result computation begins as soon as all data elements are made available at the various input points of a filter. Because all necessary inputs for the *maximum* and *minimum* filters are simultaneously available, the two filters can operate concurrently and independently. The difference filter can receive tokens at its input points simultaneously or at different times, depending on the rate at which these filters compute their respective results [35]. The results, however, are guaranteed to be the same regardless of the order in which the two filters complete their processing due to the referential transparency property. A dataflow network allows its filters to work simultaneously and asynchronously. However, it still produces the same results for any given set of inputs as its sequential equivalent. In other words, it combines the best parts of both sequential and concurrent computations.

2.1.3 Lucid Program Structure and Execution

Since Lucid is a functional programming language, every Lucid program is an expression of one or more literals, variables, operators, and/or functions. Therefore, each Lucid program also includes definitions of all the constituent variables and functions as expressions, with further definitions for each component. Evaluating the primary expression of these programs is the aim of running them. This requires evaluating each of its constituent identifiers, which in turn depends on evaluating each of its constituents, and so forth. However, unlike other functional programming languages, Lucid evaluates each statement in a specific context that may have many dimensions and produce a different answer in each possible context (as explained in Section 2.1.1). As an intensional language, Lucid not only provides the operators # and @ for directly extracting values from and declaring values for the contextual dimensions but also allows for precise and succinct definitions of each contextual dimension as any of its regular variables. Most popular imperative languages must rely on time-consuming extensional branching to analyze such phrases in all potential situations. Additionally, variable and function declarations can have context-dependent conditions imposed on them using Lucid's whenever operator, allowing them only to be evaluated when the conditions pass [31,32,34].

Consider the Lucid program presented in Listing 2.1 as an example [3]. Calculating the range of three numbers is the purpose of the program. Its primary expression is, therefore, a variable called range (defined later in the program) whose value depends on the values of the three numbers in question. As a result, it is evaluated in a three-dimensional context, with each dimension, g_num1, g_num2 and g_num3, standing for one of the three numbers. The range expression may evaluate to a different value at each point of reference in this context space. The @ operator must be utilized to compute it at a particular moment in time or to determine a particular range. The @ operator's responsibility is to return the value of its first argument, the expression, at the location and in the suitable dimension(s) defined by its second parameter, the precise point of context [31]. The range's value is returned in line 1 of the example based on the definition of the range at a point of reference where the three numbers or dimensions are, in order, 110, 12 and 14. The where clause linked to an expression in a Lucid program specifies all the definitions related to it, starting with the dimensions that specify its context.

Listing 2.1: LUCID Program to Calculate Range of Three Numbers

```
1  range @.g_num1 10 @.g_num2 12 @.g_num3 14
2  where
3  dimension g_num1, g_num2, g_num3;
4
5  range = difference (#.l_max, #.l_min)
6  @.l_max max
7  @.l_min min
8  where
9  dimension l_max, l_min;
10  difference (x, y) = x - y;
11 end;
12
13 max = maximum (#.l_num1, #.l_num2, #.l_num3)
14 wvr c_max
15 @.l_num1 #.g_num1
16 @.l_num2 #.g_num2
17 @.l_num3 #.g_num3
18 where
19 dimension l_num1, l_num2, l_num3;
20 c_max = #.l_num1 >= 0 and #.l_num2 >= 0 and #.l_num3 >= 0;
21 maximum(x, y, z) = greater(x, greater(y, z ));
22 greater(a, b) = if a > b then a else b fi ;
23 end;
24
25 min = minimum (#.l_num1, #.l_num2, #.l_num3)
26 wvr c_min
27 @.l_num1 #.g_num1
28 @.l_num2 #.g_num2
29 @.l_num3 #.g_num3
30 where
31 dimension l_num1, l_num2, l_num3;
32 c_min = #.l_num1 >= 0 and #.l_num2 >= 0 and #.l_num3 >= 0;
33 minimum(x, y, z) = lesser(x, lesser (y, z ));
34  lesser (a, b) = if a < b then a else b fi ;
35 end;
36 end
```

Each of these dimensions is specified using a dimension clause at the top of the where clause's body, indicating that it is a brand-new dimension that will only be used inside the where clause enclosed in [31]. According to the location at which it is declared, a dimension may have a global or local scope. For example, the dimensions g_num1, g_num2 and g_num3 are stated in the outermost where clause (line 3) and consequently have a global scope, as indicated by the 'g' prefix. However, as their prefix "l" indicates, the dimensions l max and l min (line 9) can only be used within the local scope of the difference function and not outside of it. The definitions of the relevant constraints, variables, and functions follow the declaration of the dimensions in the where clause of an expression. Lines 5 and 6 define the difference function, which defines the range variable. Lines 6 and 7 evaluate the function in a two-dimensional context, defined by the dimensions l max and l min. The difference function, which creates the range variable, is defined on lines 5 and 6. The two-dimensional context, represented by the dimensions l max and l min, is evaluated in lines 6 and 7. The # operator, which is in charge of returning the current value of the dimension specified as its argument, is used by the function's input parameters to extract their values from its contextual dimensions [31]. The function's dimensions, in turn, get their values from later-defined variables (max and min) in lines 13-23 and 25-35 of the program. After its dimensions declaration, the difference function is where clause (line 10) defines the calculation it will perform. With two key exceptions, the variable range definition is the same for the max and min variables that are discussed in this definition. First, the global dimensions, g_num1 (10), g_num2 (12) and g_num3 (14), are obtained instead of being used to compute the values of the local dimensions, l num1, l num2, and l num3, of the maximum and minimum functions. Second, the *wvr* operator - an alternative form of whenever - is used to provide the limitations that apply to these functions. These functions, which make up the operator's first argument, are only computed if the constraints, which make up the second argument (lines 14 and 26) [31], evaluate to be true. In order to maintain a clear and consistent program structure, all the constraints placed on a function (maximum or minimum) have been specified as part of its where clause (lines 20 and 32) and their result has been represented as a variable (c max or c min) to serve as the second argument to the *wvr* operator. Lucid also offers the or operator to indicate optionality in conditional expressions, even though this example uses the and logical operator to combine numerous conditions. These conditional expressions are evaluated in a particular context, which may be specified as a part of a where clause associated with them, just like the functions they constrain.

However, in the provided example, the constraints and the related functions are part of the same context; thus, the where clause for the conditions is unnecessary.

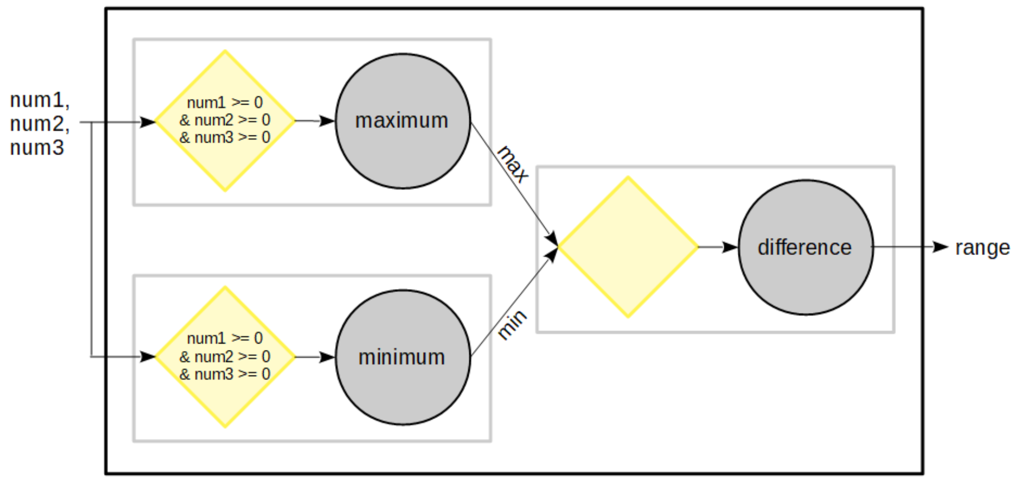


Figure 7: Range Composite Service

After describing the program's essential operation, syntax, and structure, as shown in Listing 2.1, we will now go over the program's distinctive features. The program also uses the contextual values of the functions maximum, minimum, and difference as arguments or inputs to the appropriate functions, meaning that the values of these inputs are determined by the situation in which the function is evaluated. The outcomes of specific computations are then used as the values for the contextual aspects. In other words, contrary to custom, the contextual dimension of a function in the supplied program refers to the object that provides context and input to the function and the computed data. However, one of the goals of this thesis is to express composite web services as Lucid programs; thus, we give this specific example here since it demonstrates the approach we employ. Figure 7 shows a rough illustration of a composite service in charge of computing the range of three values. While the range determined by the program serves as the composite service's clients' expected result, the three numbers in question serve as the inputs that they can offer. While the conditions attached to each function, difference, maximum, and minimum, using the *wvr* operator, serve as internal service constraints, they also serve as components of the composite service (shown as circles in Figure 7). The constraints are depicted as diamonds in Figure 7. It is interesting to note that the constraint diamonds and service circles are shown in Figure 7 perform tasks that are similar to those performed by the input/pre-condition place circles and transition bars that typically constitute Petri net graphs [7, 14, 36–39], even though we do not use Petri nets to represent composite web services in this research. We define a service's contextual dimensions, whether atomic or composite, as the set of all its input parameters. Using the same definition, the three numbers that are input to the composite service and its maximum and minimum components serve as the contextual dimensions for the range variable, maximum function, and minimum function in the corresponding Lucid program. In contrast, their values are used as arguments for the maximum and minimum functions. The same holds true for the difference function, whose dimensions match those of the inputs to the different services and whose values serve as arguments to the function. While the difference service draws its input values from the outputs of other component services, the maximum and minimum services, on the other hand, receive their input values directly from the customer. Thus, the services that give the difference service's inputs must first be executed, and their outputs must then be computed before the input values for the different services can be computed. Lines 6 and 7 of the provided Lucid program depict this predecessor-successor relationship between component services. Here, the contextual values for the different functions are variables that evaluate the outcomes of calculations made by the maximum and minimum functions.

Lucid uses a demand-driven, lazy method evaluation to perform the computations necessary for program execution. A Lucid program is an expression that includes definitions of the identifiers that make up that expression, as was previously stated.

The values of each of this primary expression's components must be known to evaluate it. Examining the expressions that define these constituents, as well as those that define their constituents, will result in the same results. When such an expression needs to be evaluated, the eductive model of computation creates a demand or request for the value of each of its component identifiers in the current context. The eduction engine examines the defining expression for each of these demands to determine its corresponding identifier before generating demands for each of the expression's components, which may result in the development of additional demands.

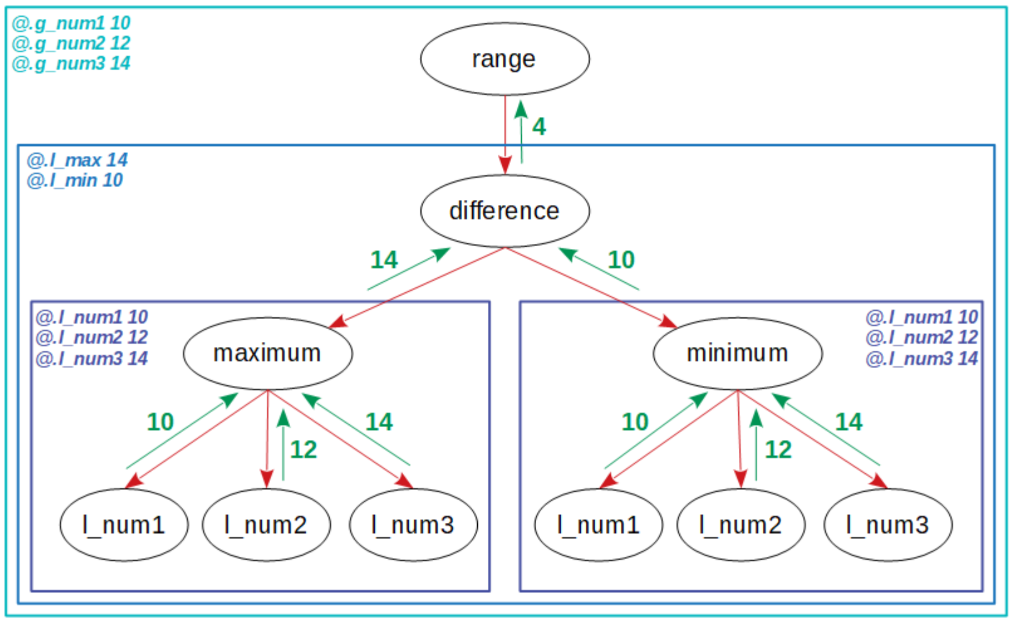


Figure 8: Demand Generation and Computation Tree for the *range* Program

Essentially, for each identifier appearing in the primary expression of a Lucid program, a tree-like structure (illustrated in Figure 8) is constructed incrementally, where each node is an identifier-demand whose children are the demands generated for evaluating the identifier's defining expression. Each branch in this tree continues to grow from the top to the bottom (as shown in red in Figure 8) until its lowest node/demand evaluates to an actual value. The node's parent in the branch receives this value after that (indicated in green in Figure 8). The definition of the identifier is applied to them, and its value is computed and propagated higher up the tree once an identifier-demand node has received the values of all of its child nodes similarly. This process of generation, propagation and consumption of demands and their computed values continues until all the identifiers for the primary expression are evaluated, and its value can be computed, thereby achieving the program's goal. The education model's frugal approach to computation, which only generates a demand for a value if and when necessary for the computation of another demanded value, is a significant advantage. Non-essential values are never used to compute results. Such a strategy optimizes the time needed to run a Lucid program while simultaneously saving execution resources [18,31].

The dataflow execution mechanism of Lucid is another feature that increases its run-time effectiveness. A Lucid program is a textual representation of a dataflow network, and when it is executed, it turns into that network, as was discussed in Section 2.1.2. A filter in a related dataflow network can be used to represent each definition in a Lucid program, doing the computation that the definition itself specifies. While the internal input-output relationships among defining expressions take the form of the network's channels, the external inputs given to and the outputs generated by the program serve as the network's respective inputs and outputs. Consider Listing 2.1's range program and Figure 6's dataflow network as an illustration. Only the three main defining functions of the program—maximum, minimum, and difference—have been depicted as filters in the dataflow graph for clarity and simplicity. Similar graphs, made up of filters that represent their respective definitions, can be created separately for each of these functions, though, in order to illustrate how they each work more fully. Considering the structure of the range program, its corresponding dataflow graph, and the composite service (illustrated in Figure 7) that it roughly represents, it is evident that while the dataflow graph represents the range composite service as a whole, each of its constituent filters represents one of its components in the current example. Additionally, each filter in a dataflow network can work concurrently with other filters, as we already know. Translating a composite service into a Lucid program makes it possible to achieve parallel operation of its component services, reducing the time required for the service to complete its processing. Furthermore, Lucid's inherent concurrency of operation prevents the possibility of complex problems, which are frequently known to result from improper thread management because it requires no additional programming effort to create or maintain numerous threads. Lucid, in other words, offers a way to increase the effectiveness of composite service execution through intrinsic concurrency without subjecting it to the dangers of subpar multi-threading [32, 35].

2.1.4 Objective Lucid

Several different Lucid dialects have developed since the language's creation in 1974 [32], each with its unique traits. One of these variants, GLU (Granular Lucid), used the intensional language Lucid to specify the parallel structure of an application and the imperative language C to specify the application's functions. This allowed programmers to benefit from both the simplicity of programming in mainstream languages and the effectiveness of intensional dataflow languages [40,41]. This hybrid language employs C to define the actions carried out by each of these filters, while Lucid is used to declaring the filters and linking channels that make up the dataflow network equivalent of a program. Objective Lucid, another Lucid dialect based on the same hybrid paradigm, replaces C with Java as its imperative component, enabling its imperative segment to control Java objects as first-class values and use Java's dot-notation to modify the objects' members [42,43]. To simulate and run composite web services on GIPSY in our research, we model them using Objective Lucid (the rationale for which is discussed later in this section). Each Objective Lucid program consists of two code segments: one in Java and denoted by a #JAVA tag, and the other in Lucid and denoted by a #OBJECTIVELUCID tag. For instance, take a look at the Objective Lucid translation of the pure (Indexical) Lucid program in Listing 2.1, which calculates the range of three values displayed in Listing 2.2. Three key differences exist between this Objective Lucid program's Lucid segment and its Indexical Lucid counterpart. First, Objective Lucid replaces the declarative definitions of the difference of the functions, maximum, and minimum

Listing 2.2: Objective Lucid Program to Calculate Range of Three Numbers

```
#JAVA
public class ReqComp
{
    private int diff;

    public ReqComp(int diff)
    {
        this.diff = diff;
    }
}

```

```
public class Difference
{
    private int x;
    private int y;
    private int diff;

    public Difference(int x, int y)
    {
        this.x = x;
        this.y = y;
        diff = 0;
    }

    public void process()
    {
        diff = x - y;
    }
}

public Difference calcDiff(int x, int y)
{
    Difference oDifference = new Difference(x, y);
    oDifference.process();
    return oDifference;
}

public class Maximum
{
    private int x;
    private int y;
    private int z;
    private int max;

    public Maximum(int x, int y, int z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
        max = 0;
    }

    public void process()
    {
        max = x;
        if (max < y)
        max = y;
        if (max < z)
        max = z;
    }
}
```

```
        public Maximum calcMax(int x, int y, int z)
        {
            Maximum oMaximum = new Maximum(x, y, z);
            oMaximum.process();
            return oMaximum;
        }

        public class Minimum
        {
            private int x;
            private int y;
            private int z;
            private int min;

            public Minimum(int x, int y, int z)
            {
                this.x = x;
                this.y = y;
                this.z = z;
                min = 0;
            }

            public void process()
            {
                min = x;
                if (min > y)
                min = y;
                if (min > z)
                min = z;
            }
        }

        public Minimum calcMin(int x, int y, int z)
        {
            Minimum oMinimum = new Minimum(x, y, z);
            oMinimum.process();
            return oMinimum;
        }

        #OBJECTIVELUCID
        oRange @.g_num1 10 @.g_num2 12 @.g_num3 14
        where
        dimension g_num1, g_num2, g_num3;

        oRange = ReqComp(#.l_diff)
        @.l_diff oDifference.diff
        where
        dimension l_diff;

```

```
112     oDifference = calcDiff(#.l_max, #.l_min)
113     @.l_max oMaximum.max
114     @.l_min oMinimum.min
115     where
116     dimension l_max, l_min;
117     end;
118
119     oMaximum = calcMax (#.l_num1, #.l_num2, #.l_num3)
120     wvr c_max
121     @.l_num1 #.g_num1
122     @.l_num2 #.g_num2
123     @.l_num3 #.g_num3
124     where
125     dimension l_num1, l_num2, l_num3;
126     c_max = #.l_num1 >= 0 and #.l_num2 >= 0 and    #.l_num3 >= 0;
127     end;
128
129     oMinimum = calcMin (#.l_num1, #.l_num2, #.l_num3)
130     wvr c_min
131     @.l_num1 #.g_num1
132     @.l_num2 #.g_num2
133     @.l_num3 #.g_num3
134     where
135     dimension l_num1, l_num2, l_num3;
136     c_min = #.l_num1 >= 0 and #.l_num2 >= 0 and    #.l_num3 >= 0;
137     end;
138     end;
139     end
```

with procedural definitions of the respective Java methods *calcDiff* (lines 31–36), *calcMax* (lines 63–68), and *calcMin* (lines 95–100), which can be called from the Lucid segment using regular method call statements (lines 112, 119 and 129). Furthermore, unlike their declarative equivalents, which only return simple variables, these procedural functions may additionally return Java objects (such as *oDifference*, *oMaximum*, and *oMinimum*), which may include one or more data members and/or member functions of various data/return kinds. The dot operator on the relevant object can be used to access each of these members within the Lucid segment. For instance, the function *calcDiff* in the provided Objective Lucid example first creates an object *oDifference* of the class *Difference* (line 33), then calls its member function process (line 34) to compute the difference between its two arguments and saves the result in *oDifference*'s data member diff (lines 25–28), before returning the object (line 35). After that, in the Lucid segment, the object *oDifference* (line 108) returned by *calcDiff* is accessed to obtain the computed difference value (line 112). The computation and access of the program's maximum and minimum values work the same way. Third, the final output of the Indexical Lucid program is a simple variable called range, which holds the return value of the difference function. The final output of the Objective Lucid program is an object called *oRange*, which is made by calling the ReqComp function *Object()* [native code] and has a single data member called diff. In this program, diff is responsible for holding the return value of the difference function. Additionally, as is clear from the program, the value of *oRange* depends solely on *ReqComp*, which depends on the functions that compute the difference, maximum, and minimum values. As a result, in the Objective Lucid program, these functions have been shifted from the global scope (such as in the Indexical Lucid program) to the local scope specified by ReqComp's where clause. The goal of using Objective Lucid and the specific language structures covered in the above sentence is to create a comprehensive and understandable representation of context- and constraint-aware composite services that may be emulated or performed on GIPSY. It becomes required for composite services intended to be executed on GIPSY first to be translated into some Lucid dialect because GIPSY is dedicated to the compilation and execution of Lucid applications. This dialect must possess some particular properties in order to be able to represent all the necessary features of the composite service model that we use (discussed in Chapter 3). Based on what we know about Lucid variants, we can conclude that only Objective Lucid can do this (without introducing superfluous characteristics). We justify this decision by discussing the requirements and the Objective Lucid constructs that help meet them:

1. **Requirement:** Parameters that serve as inputs to a service (whether atomic or composite) or on which constraints are imposed must be permitted to serve as contextual dimensions while keeping the capacity to be defined, computed, and utilized as regular variables.

   Solution: As shown in Section 2.1.3, Indexical Lucid permits the definition of service inputs and constraint features as contextual dimensions using the dimension clause and the @ operator. The same is true for the Lucid portion of a similar Objective Lucid software. In addition, in the Java portion of the program, service inputs can be given and processed as regular function arguments and as data members of objects within their respective service definitions. In the meantime, constraint features can be used as regular variables in the conditional statements that define these service constraints in Lucid. For example, inputs max and min to the different components of the range composite service (illustrated in Figure 7) are defined as both inputs and dimensions, l max and l min, for its corresponding function *calcDiff* in Listing 2.2 (lines 112 - 117), while being processed as regular variables as part of their service definition in the Java segment (lines 12 - 36). Similarly, the values of l_num1, l_num2, and l_num3, which serve as shared contextual dimensions for function calcMax and constraint c_max (lines 121 - 123 and 125), are utilized as convenient components of the constraint's specification (line 126) in the Lucid portion of the program.

2. **Requirement:** It should be permitted for services (whether atomic or composite) to accept multiple input parameters and generate multiple output parameters. The input values are extracted from the context. The output values should then be applied to the context upon output. In addition, the outputs of a component service should be passable as inputs to other components within the same composition.

   Solution: As mentioned in Section 2.1.3, each component service of composition is represented as a function in an Indexical Lucid program and, consequently, in the Lucid portion of an Objective Lucid program. In the event that a service generates multiple outputs, Objective Lucid permits them to be composed into a Java object and returns from the service's Java segment definition to the Lucid segment. The dot operator can then be used to access individual output parameters (or data members) from these objects and pass them as inputs or arguments to other service definitions or functions in the Lucid segment. Consider the call to the *ReqComp* function *Object() { [native code] }* in Listing 2.2 as an example (line 107). Even though it seems unnecessary in the given example where the range composite service is expected to produce only one output parameter, if a composite service produces more than one output parameter, outputs (possibly from different component services) can be passed as arguments to this function *Object() { [native code] }*, put together as data members of a Java object (such as *oRange*), and returned as the program output.

3. **Requirement:** A service should only be permitted to operate if all its constraints evaluate to be true.

   Solution: As described in Section 2.1.3, the *wvr* (or whenever) operator provided by Lucid permits constraints to be set on expressions, including those representing web services. Such phrases are computed, i.e., the services are executed if and only if the requirements imposed upon them are true. In order for the function *calcMax* to be evaluated in Listing 2.2 (line 119), the conditions (line 126) that specify its constraint variable c_max (line 120) must first evaluate to true.

4. **Requirement:** In a program representing a composition, the simulated implementation of its component services should be easily embeddable and, if necessary, replaceable with links to real services to facilitate a seamless transformation of service simulation into execution.

   Solution: In Objective Lucid, the declarative descriptions of functions representing component services in an Indexical Lucid counterpart of a composite service are replaced with procedural function definitions written in Java — a popular imperative programming language. When compared to Lucid, programmers who are more familiar with Java can write placeholder implementations of component services and even reuse code that may already be available for simulation purposes much more quickly. In addition, Java [44] makes it simple to replace the simulation code of a service with an implementation that summons the basic online service, hence reducing the work required to transition between simulation and execution of service compositions. We have to make this work for any Java program (pre-existing). The solution is to create a worker function wrapper calls the main function of a pre-existing Java implementation for a service.

2.2 GIPSY

The *General Intensional Programming System* (GIPSY) is a platform for programming in several languages and a demand-driven distributed execution environment for all Lucid dialects [18]. It is a continuing effort aimed at evaluating the capabilities of the intensional programming model as implemented in the most recent Lucid versions in many domains. GIPSY's architecture is composed of three tiers or independent processing units: the Demand Generator Tier (DGT), the Demand Worker Tier (DWT), and the Demand Store Tier (DST). Each tier is responsible for executing a distinct set of tasks as part of a program's execution process. All these levels (whether implemented on the same or different computers) must interact and collaborate for this process to be completed, working together to achieve a common objective. Computers that register to host one or more of these tiers are referred to as GIPSY nodes, while a collection of interconnected GIPSY tiers deployed on these nodes and executing GIPSY programs is referred to as a GIPSY instance. Through the generation, dissemination, and consumption of demands, a GIPSY instance's tiers can communicate. The operational manner of the GIPSY Multi-Tier Architecture is entirely demand-driven. As stated in Section 2.1.3, a demand in the eductive computing paradigm is a request for the value of a program identifier in a specific evaluation context. While intensional demands are generated for evaluating Lucid identifiers, procedural demands are generated for procedure identifiers by demand generator tier (DGT), i.e., procedural function calls embedded in a hybrid Lucid program.

The declarative specification of every Lucid identifier that appears in a program is handled by this tier, which traverses the abstract syntax tree (AST) representation produced by the GIPSY compiler, the General Intensional Programming Compiler (GIPC). The DGT generates an intensional demand for each ID and sends it to the Demand Store Tier. The DST, also called the warehouse, is in charge of storing computed demands and their results in a way that does not change. It also serves as an asynchronous communication middleware between tiers to process migrating requests and computed values. When a new demand is received from a tier, the DST searches its records to determine if the demand's value has already been computed. If the value is located, it is returned to the requested tier; otherwise, the demand remains in the warehouse until a tier capable of computing it becomes available. If the value of an intensional demand is not already in the DST, it can be picked up by the same DGT that made it or by a different DGT, if one is available to be further processed. Once computed, the resulting value of the demand is conveyed to the warehouse so that it may be saved for future reference, hence improving processing performance by eliminating the need to recompute the value of every demand whenever it is eventually re-generated after being processed. In the event that the demand was remotely processed by a DGT different from the one that created it, the DST delivers its computed value (after being logged) to the original tier. Figure 8 demonstrates that the computation of demands may depend on the values of other identifiers. In such a case, the DGT processing this demand generates additional requests for these constituent identifiers and sends them to the DST for computation using the same procedure as was used to evaluate the initial demand. In such a case, the DGT processing this demand generates additional requests for these constituent identifiers and sends them to the DST for computation using the same procedure as was used to evaluate the initial demand. Once these values have been calculated, they are sent to this DGT. This DGT then uses them to determine the original demand and sends the result back to the warehouse for storage.

A DGT generates a procedural demand when traversing the AST of a Lucid identifier and encountering a procedural, functional call node. In contrast to intensional demands, a procedural demand can only be processed by a Demand

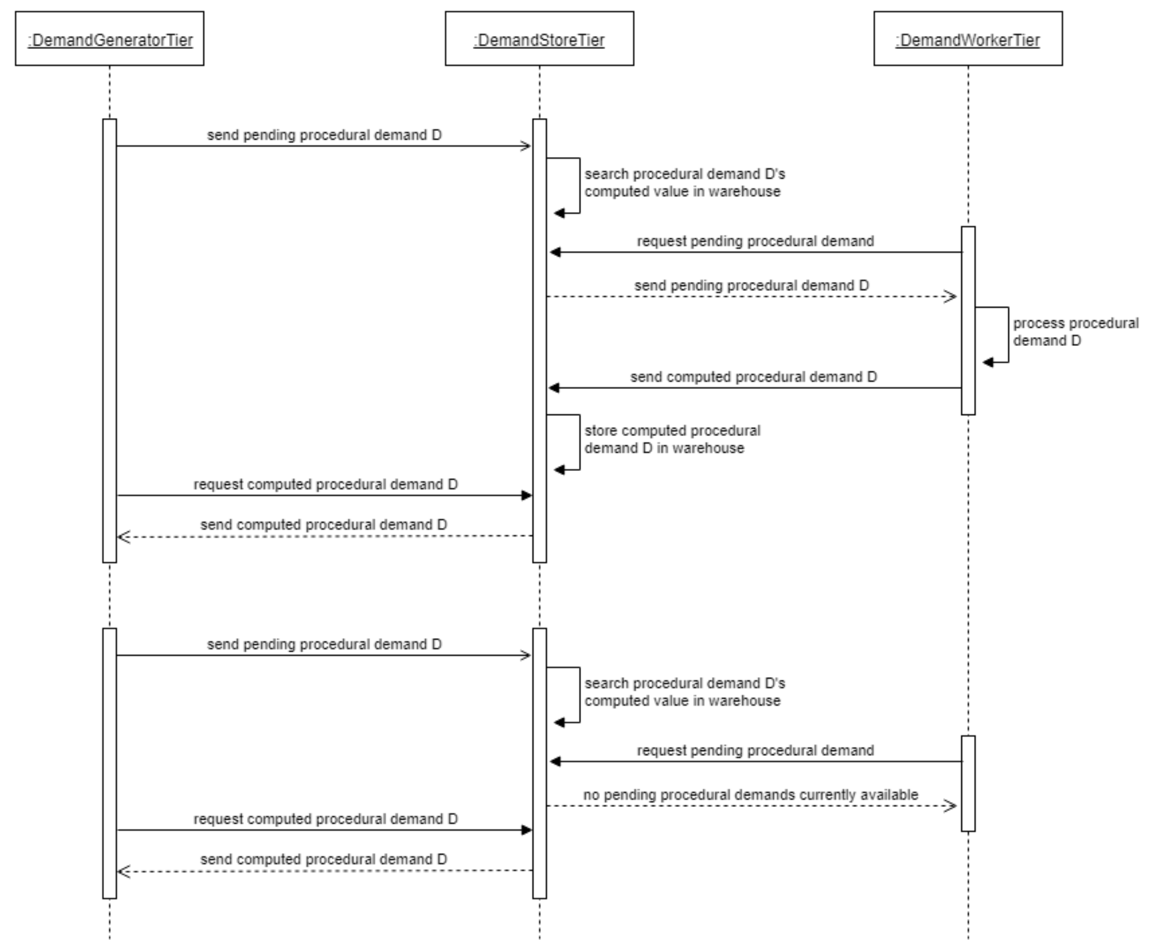

:DemandGeneratorTier
:DemandStoreTier
:DemandWorkerTier
send pending procedural demand D
search procedural demand D's computed value in warehouse
request pending procedural demand
send pending procedural demand D
process procedural demand D
send computed procedural demand D
store computed procedural demand D in warehouse
request computed procedural demand D
send computed procedural demand D
send pending procedural demand D
search procedural demand D's computed value in warehouse
request pending procedural demand
no pending procedural demands currently available
request computed procedural demand D
send computed procedural demand D

Figure 9: Processing of New and Previously-Computed Procedural Demand on GIPSY

Worker Tier, i.e., only a DWT can pick it for computation as it waits in the DST (if its resulting value is not already in the warehouse), execute the corresponding procedure written in an imperative language, and send the result back to the DST. The DST initially maintains this procedural demand and its computed value for future reference in the same manner as intensional demands and then migrates the value back to the DGT that originated the demand. Figure 9 depicts the activities and processes conducted by the DGT, DST, and DWT during the computation of a new (i.e., not previously computed) procedural demand D. When the same demand is produced again, and its result is already in the warehouse, avoiding the need for recalculation.

2.3 Genetic Algorithms

This section provides an overview of genetic algorithms as we shall use them in our solution. It clarifies the terminologies used throughout the thesis and explains how a typical genetic algorithm operates. In addition, it explains some of the more complex aspects of genetic algorithms. These characteristics include constraint handling techniques, strategies for multiple tasking and conflicting problems, and cooperative co-evolution models, all of which are required for resolving context- and constraint-aware composite service challenges.

Genetic algorithms (GAs) [45] are search methods originally designed by John Holland and based on the concepts of natural selection, a biological process in which fitter individuals are more likely to prevail in a competitive environment. GAs are typically employed to generate acceptable solutions to optimization problems whose search space cannot be traversed efficiently using conventional optimization techniques, such as gradient descent or heuristic-based techniques. It has been demonstrated that genetic algorithms are an efficient and effective method for tackling certain types of search and optimization issues. Furthermore, genetic algorithms have been used successfully to solve problems in many different areas, such as scheduling [46], neural networks [47], face recognition [48], and other NP-complete problems. The idea behind GAs is to take the successful optimization strategies that nature uses, called Darwinian Evolution, and change them so that they can be used in mathematical optimization theory to find the global optimum in a given phase space.

A GA operates on a specific population. Each member of the population represents a potential solution to the optimization problem. Individuals are evaluated based on their fitness. The fitness level represents a population member's ability to tackle the optimization challenge. Typically, a GA begins with the creation of a random population. Then, the genetic operators of selection, crossover, and mutation cause a transition from one population to the next, resulting in the evolution of the population. Finally, the fittest individuals will be determined by a selection procedure be selected for the next population. Crossover involves exchanging genetic material

Algorithm 1 A classic genetic algorithm

```
1: INITIALISE population with random candidate solutions
2: EVALUATE each candidate solution
3: while termination condition is not true do
4:     SELECT individuals for the next generation
5:     CROSSOVER pairs of parents
6:     MUTATE the resulting offspring
7:     EVALUATE each candidate solution
8: end while
```

between two individuals to produce two new individuals. In addition, the genetic material of an individual can be arbitrarily altered through the process of mutation. Eventually, the genetic operators are applied to the population members until a satisfactory solution to the optimization problem is found. Typically, the solution is attained when a predetermined stopping condition is met, such as when a particular number of generations are reached, when the degree of diversity of people between generations is decided upon, or when a predetermined fitness value is reached. There may be better solutions than the result, but the algorithm can be calibrated to produce a realistic solution that meets specific criteria consistently. Algorithm 1 displays the pseudocode for a classical genetic algorithm.

A GA typically consists of some components: *Genome Encoding*, *Fitness Function*, *Population*, *Parent Selection* pocess, genetic operators such as *Crossover* and *Mutation*, and *Elitism* are the fundamental components. Below is a description of these components:

Genome Encoding: A GA operates directly on a coding space that can be comprehended by the algorithm instead of a problem-solving solution space. Genome Encoding converts a solution into a code that the GA can interpret. Genome Encoding connects the "real world" to the "GA world", establishing a link between the original problem context and the problem-solving area where evolution occurs [49]. An individual (or chromosome) is a unique solution that contains "genetic" information used by the GA. A chromosome comprises genes, each representing a single factor for a controlling factor. Depending on the nature of the challenge, the chromosomal representation could be encoded differently. A bit string encoding [50] is the traditional method due to its simplicity and traceability. Nonetheless, a string-based representation may give challenges and perhaps unnatural hurdles to specific optimization tasks, such as the graph colouring problem [?]. Therefore, other encoding strategies, including real number representation [51], order-based representation [50] for bin-patching and graph colouring, embedded lists for factory scheduling difficulties, variable element lists [52] for semiconductor designing, and even LISP S-expressions [53], have been investigated.

Fitness Function: The primary mechanism for determining each chromosome's condition is provided by a problem's objective function. This crucial piece connects the GA and the system. A fitness function is a specific objective function that specifies a chromosome's quality in a GA so that it can be compared to all other candidate chromosomes and ranked accordingly. As a result, a better generation will be created by allowing suitable candidate chromosomes to breed and combine their datasets via various methods.

Population: In each generation, the population is a subset of solutions. Also, It is referred to as a collection of chromosomes. Therefore, several considerations must be made when working with the GA population:

- Maintaining the population's diversity is necessary to prevent an untimely convergence.
- A significant population can cause a GA to slow down, but a small population may not provide a sufficient mating pool. Consequently, the appropriate population size must be determined through trial and error.

It has been noticed that the entire population should not be initialized with a heuristic, as this can lead to a population with identical solutions and little variability.It has been observed experimentally that random solutions move the population to its optimal state. Therefore, with heuristic initialization, we seed the population with a few reasonable solutions and then fill in the rest with random solutions instead of populating the entire population with solutions based on heuristics.

Parent Selection Mechanism: A fraction of the current population is chosen to breed a new generation throughout each succeeding generation. Most of the time, individual solutions are chosen based on how fit they are. Solutions that are fitter (as measured by a fitness function) are more likely to be chosen. Selecting the best solutions is prioritized in some selection methods, which rate each solution's fitness. Other methods, which may take a long time, only rate a random sample of the population. The rank-based selection, the proportional selection (for example, a roulette wheel selection is a typical proportional selection), and the tournament selection are all common ways to choose a parent [54].

Crossover: The crossover process is recombining two parent chromosomes to produce a new offspring chromosome. Crossover is based on the idea that by mating two individuals with distinct but desirable characteristics, we can produce offspring that combines these traits. The exchange of genetic material between homologous chromosomes is comparable to reproduction and biological crossover. One-point crossovers, two-point crossovers, and uniform crossovers are all common crossovers [55]. Additionally, we may design a specific crossover operator by incorporating domain expertise for a particular situation.

Mutation: Mutation is a genetic operator that modifies the starting state of one or more gene values on a chromosome. This can lead to adding utterly new gene values to the gene pool. The genetic algorithm can find a previously impossible solution with these updated gene values. The mutation is essential to genetic search because it prevents populations from stagnating at local optimums. Evolutionary mutation occurs according to a user-defined mutation probability. This probability should typically be set to a low value (0.01 is a good first choice). The search will revert to a random, primitive state if it is too high.

Elitism: Elitism is the selection of superior individuals or, more precisely, the selection of superior individuals with a preference. Elitism is essential because it permits solutions to improve over time.

2.4 Related Work

The primary purpose of this thesis is to develop a simulation- and execution-based verification solution for context- and internal-constraint-aware composite web services on large datasets. Specifically, the objective is to test the scalability of the best simulation- and execution-based verification solution we have found so far and resolve its issues and improve limitations. Consequently, our review of the existing literature on composite web services was centred on answering the two questions listed below:

Q1 Is there any research that proposes a method for verifying the internal constraints imposed on composite web services? If there are, do any of their strategies have flaws that our method can compensate for?

Q2 What kind of solutions using Genetic Algorithms exist, in general, for the composition of web services? When compared to our solution, do they have any similarities or differences that we should be aware of?

In this section, we examine the findings of our review procedure, which help us answer the questions mentioned earlier.

The answer to **Q1** begins with a discussion on the research works that have been done up to this point on the composition of internally-constrained web services. In [56], Wang et al. admit that the majority of web services can only function correctly inside a specific environment whose boundaries are specified by the constraints imposed on them by their providers, which we refer to as internal service constraints, and that these constraints have an immediate effect on the compositions that employ them as components. They describe how a composite service can fail during execution if one or more restrictions imposed on its component services are not satisfied, although all input values comply with the requisite input types. The authors offer a graph-based approach coupled with unique preprocessing techniques for constraint-aware web service design to prevent such failures. Each component service in these compositions may be substituted by a branched combination of services, each capable of doing the same task but under different restrictions (i.e. in different contexts). The execution context determines which service from each group is invoked at runtime. Although this method expands the contextual range of a composite service, it does not ensure coverage of the entire context space, which is entirely dependent on the combined contextual range of the available services for composition. As a result, these composite services still risk failing during execution due to a constraint violation if the input values do not match the combined contextual scope of their component service alternatives. Similarly, Laleh et al. [19–23] conducted comparative research. This work provides graph-based planning algorithms for the automated construction of internally limited web services based on their input-output links. The article proposes a unique method for moving the constraints in each composition plan to the earliest possible place at which they can be checked correctly to decrease the number of component service rollbacks resulting from a constraint-verification failure during the execution of a composite service. Once all constraint-aware composition plans for a given composition request have been developed, according to the study, they can be integrated to build a larger package, including multiple possible solutions for the same composition request. In the event that the internal constraints of one plan fail verification in a specific execution context, the plan can be rolled back, and the next plan in the set can be chosen for execution. Although this method expands the contextual scope of the composition solution while minimizing the danger of failure due to constraint violation (similar to [56]), it does not eliminate the risk of runtime failures. As a result, a verification system is required for detecting the scenarios or regions of execution context space in which a composite service could fail and validating that it behaves as expected within its appropriate contextual boundaries so that unexpected post-deployment failures and their resulting damages (as discussed in Section 1.1) can be avoided or, at the very least, prepared for. Petri nets for the composition of internally-constrained web services allow for some simulation-based verification. Cheng, Liu, Zhou, Zeng, and Yla-Jaaski [36],present an automatic composition method for internally- constrained fuzzy semantic web services utilizing Fuzzy Predicate Petri Nets (FPPN), where fuzzy semantics (or fuzziness) refers to syntactic and semantic representations involving fuzzy variables and fuzzy membership functions. A composition request from the user is taken in this technique, and its elements are described as a set of facts (user-provided inputs), rules (user-imposed behavioural constraints), and a goal statement (user-expected outputs) in the form of Horn clauses. The Horn clauses are then subjected to a T-invariant analysis technique to create a collection of internally-constrained component services that can satisfy the user's fuzzy input/output and behavioural constraint needs by ensuring that the services' internal constraints do not conflict with the requester constraints. The T-invariants are then represented as an FPPN (a fuzzy extension to the usual predicate/transition nets) and examined to guarantee that the composite service is entirely free of deadlocks. Finally, the reachability graph of Petri nets is utilized to identify the execution order of the composite service's components. The QoS value of the composite service can be calculated and used to select the optimal composition among all those generated for a given request based on this sequence and the QoS parameter of each component service. Modelling composite services using Logical Petri Nets (LPNs), a high-level abstraction of Petri nets with inhibitor arcs, is a comparative approach to constraint-aware web service composition presented by Zhu and Du in [7]. Following this methodology, the input/output requirements derived from a composition request are translated into input parameters required by and output parameters generated by the accessible composition services. In the meantime, the user's behavioural and qualitative constraint requirements are formalized as logical expressions to safeguard the inputs and outputs of the resultant Petri net's transitions. During composition, only those atomic services are selected that meet not only the input/output needs of the user but also display internal behavioural limitations that align with the requester's constraints. It is common knowledge that Petri nets can simulate the behaviour of the systems they represent, and some online tools assist users in observing this simulated behaviour [37, 38], although the two studies above primarily focus on composition and not verification of services. Consequently, utilizing the models above for simulation-based testing of internal and user-constraint-aware composite services may be viable. Petri nets are incapable of executing real services and cannot be utilized for execution-based verification of composite services, which must confirm that a service's actual behaviour matches its description.

Wang and Yu [13] present another simulation-based verification technique for composite web services. According to this technique, the OWL-S process definition of the composite service to be tested is first transformed into a finite state program written in an executable subset of Projection Temporal Logic known as object-oriented MSVL (PTL). The features that must be validated are described as Propositional Projection Temporal Logic (PPTL) formulae - a specification language for defining desirable properties. The object-oriented MSVL interpreter then executes the composite service program with the desired property formulas to determine if the service meets the properties. As an object-oriented language, MSVL facilitates the development of more structured and comprehensible applications, lowering potential errors. In addition, it permits the representation of numerous composition constructions, such as *Split*, *Join*, *Any-Order*, *If-Then-Else* and *Iterate*. However, this technique (like Petri nets) cannot execute real services to analyze their actual behaviour. Moreover, other than noting that the components of a composite service may be subject to specific pre-conditions, the authors do not mention verification of internal service limitations, leaving open the question of whether or not this system is relevant to our research.

In [4], Aggarwal, Verma, Miller, and Milnor suggest an alternative strategy for the constraint-driven composition of web services. Using a composition framework called METEOR-S (Managing End-to-End OpeRations for Semantic Web Services), this method allows the parts of an abstract process (a composite service) to be connected to concrete web services based on business and process constraints. This creates an executable process. This research uses BPEL4WS (Business Process Execution Language for Web Services) to create the abstract process. Then, it augments the process activities with service templates that define functional semantics and QoS specifications to aid the constraint analyzer and execution engine modules of METEOR-S in matching concrete services to abstract placeholders. In addition, this study proposes the usage of semantically annotated WSDL service descriptions

Table 3: Comparison of Research Works Concerning Internal Constraints (**Q1**)

Authors & Citations	**Year**	**Approach/Model/ Tool**	**Automated Composition**	**Internal Constraint Modeling**	**Simulation-based Verification**	**Execution-based Verification**
Aggarwal et al. [4]	2004	METEOR-S framework	+	+	−	+
Zhu and Du [7]	2010	Logical Petri Net	+	+	+	−
Wang et al. [56]	2014	Graph-search-based internal-constraint-aware composition	+	+	−	−
Cheng et al. [36]	2015	Fuzzy Predicate Petri Net	+	+	+	−
Wang and Yu [13]	2015	MSVL-PPTL verification	−	+	+	−
Laleh et al. [19–23]	2016-18	Planning-graph-based internal-constraint-aware composition	+	+	−	−
Gupta [3]	2019	Constraint-adjustment for context/constraint-aware using GIPSY	+	+	+	+

(+) Support, (-) No Support.

stored in an upgraded UDDI registry with the METEOR-S discovery engine module's interface to enable METEOR-S to locate relevant concrete services for matching. Finally, the QoS standards of a group of found prospective candidates for a process are used to select the ideal service. After the development, annotation, discovery, and composition steps have been completed, and the BPEL4WS web process can be executed on the BPWS4J engine. Unlike those previously discussed, this approach produces constraint-aware composite services that can be used to test the actual behaviour of real services. However, it needs the human construction of an abstract process, which becomes increasingly unfeasible as the composite service's complexity develops. In addition, this solution only supports execution-based verification of composite services; simulation-based testing is not possible.

In contrast, the Lucid/GIPSY combination that Gupta [3] suggested in their research work can employ either technique with equal ease, expanding the verification process's scope and, thus, increasing its dependability. In addition, a component of their solution is an improved version of the automated composition method defined by Laleh et al. [19–23], which incorporates a unique constraint-adjustment feature for reducing the rollback effort caused by runtime constraint-verification failures, thereby enhancing the overall efficiency of their simulation/execution process. Based on the study conducted to date, it can be concluded that only Gupta's solution [3] for verifying the internal limitations placed on composite services relies on simulation and execution. As the best solution, we tested scalability by running a large dataset on Gupta's simulation/execution [3]. We found some limitations and failures in generating a graph plan during our experiments. For example, the graph grows exponentially in the forward expansion phase, which turns into another problem in terms of failure to generate any solution for the user's request. Our motivation is to propose a solution to overcome this challenge, and we selected a genetic algorithm to generate a graph plan on large-scale datasets.

In order to answer **Q2**, we look at various scalable methods presented so far for different parts of composite web services. Although these solutions are not concerned with verifying internal service limits, contrasting them with our validation system allows for a more thorough analysis of its strengths and limitations. Using a Genetic algorithm (GA), the obstacle in the service composition optimization process is efficiently overcome. Canfora et al. [57] studied a genetic algorithm-based technique for resolving QoS-aware web service composition. Later, other researchers [58–61] enhanced this methodology. Yu and Lin [62] propose two models and two algorithms for this problem: 1) a combinatorial model and 2) a graph model. They provide a comparison of the four methods as well as usage settings. Schuller et al. [63, 64] examined this issue's unstructured orchestration patterns and stochastic QoS factors. Alrifai et al. [65] utilized skyline technology as a preprocessing step to exclude uninteresting candidates from the search space, hence decreasing the needed selection time. The authors use mixed integer programming to find the optimal decomposition of global QoS requirements into local QoS constraints and local selection to identify the best services that satisfy these local constraints for each abstract assignment [66]. X. Sun and J. Zhao also investigates the decomposition-based technique with global QoS guarantees [67]. Klein et al. [68] suggested a network-aware methodology to differentiate the cloud QoS of services from the network QoS. The selection technique based on this model is utilized to identify compositions with reduced latency. Wada et al. [69] suggested a system for optimizing service level agreement (SLA)-aware service composition termed E3. It can address several SLAs concurrently and efficiently produce a set of Pareto solutions by utilizing a multiobjective GA to maximize the entire QoS utility. Some services may be reconfigured and offer various QoS compromises. Ma et al. [70] addressed the problems of reconfigurable service modelling and effective service composition decision-making. Leitner et al. [71] investigated how to determine the ideal set of adjustments to reduce the total cost of SLA breaches, while Wang et al. [72] suggested a restriction-aware service composition technique based on the notions of service intension and service extension. As previously stated, approaches in the preceding category lack selection flexibility since they restrict candidates to only those concrete examples that have the functions provided in the composite service as a single abstract service. Lecue and Mehandjiev [73] attempted to broaden the selection scope by relaxing functional semantic matching, and they used a genetic algorithm to optimize QoS along with the global semantic matching degree.

Therefore, the additional gain in selection scope comes at the expense of semantic similarity. Some researchers consider all services to be stateless and arrange them in a dependency tree based on semantic inputs and outputs. The process plans of composite services are then automatically constructed by locating paths in this graph and selecting the path with the optimal QoS [74–78]. Utilized technologies for optimal path search include the worklist algorithm from [76] and the iterative deepening depth-first search from [78]. These methods dynamically generate process plans; as a result, their selection scope of service instances for service composition can be expansive. Nevertheless, most of them handle QoS and aggregation functions more simply and do not support QoS constraints from end to end. Moreover, when generating a specific path, only local optimization is applied if numerous candidates for specific functionality exist. Yilmaz et al. [79] presented an enhanced GA to address service composition with minimal global QoS and to enhance scalability. Using a backtracking-based approach and an extended genetic algorithm, Quanwang et al. [80] established a unique QoS-aware web service design method that optimizes

Table 4: Comparison of Research Works Concerning Genetic Algorithms (**Q2**)

Authors & Citations	**Year**	**Approach/Model/ Tool**	**Automated Composition**	**Internal Constraint Modeling**	**Scalability**
Yilmaz et al. [79]	2014	Enhanced GA	+	−	+
Kumar et al. [83]	2018	Multi-Tenant Cloud Service Composition using GA	+	−	+
Kashyap et al. [84]	2020	Service Composition in IoT using using GA	+	−	−
Hussain et al. [46]	2022	A multi-objective quantum-inspired genetic algorithm	+	−	+
Sefati et. al [85]	2022	Adaptive Penalty Function in GA	+	−	+

(+) Support, (-) No Support.

the overall QoS values. Ludwig [81] suggested a method based on clonal selection to solve workflow service selection with high-quality solutions.

Single-point crossover and mutation are the basis for all of these techniques. In addition, these strategies proposed adjustments to the crossover operator and parental chromosome selection to circumvent local optima. However, chromosomes become extremely lengthy if the number of abstract services and candidate services for each abstract service increases. Consequently, the GA technique leads to poor readability of the chromosome, crossover, and mutation, and it cannot forecast information on the semantics of services. Additionally, it leads to a low convergence rate and a low premature rate in local optima [82].

Genetic algorithms often use crossover and mutation operators to develop offspring (solutions) from the selected parents without considering search space global information utilizing the location information of the solutions obtained thus far. EA/G, unlike genetic algorithms, generates children using a probability mechanism that ensures the distribution of promising solutions in each generation from which offspring are sampled. Instead of directly utilizing the position values, EA/G changes the probability values for each generation based on global statistical information acquired from the population's members. To tackle the complementing aspect of GAs and EDAs (estimation of distribution algorithms), Zhang et al. [86] introduced a unique evolutionary algorithm with guided mutation (EA/G) that uses local knowledge of the solutions (like GAs) and global information about the search space (EDAs) when generating offsprings (solutions). EA/G employs a mutation operator known as a guided mutation, in which an offspring is created by integrating global statistical information about the search space with local information about the parent solution. These studies check for component ordering, deadlock, live-lock, reachability, and QoS properties. On the other hand, we are interested in validating the internal constraints that service providers put on their services. Kumar et al. [83] modeled service composition as an evolutionary optimization problem with a novel encoding representation and fitness evaluation technique. If changes occur during the execution of a composite service (e.g., partner service failure, uncertain request burden, QoS fluctuation, and environmental changes, etc.), the middleware is able to proactively recompose the tenant-required QoS-ensuring service execution plan. Kashyap et al [84], objective is to reduce service cost, execution time, and reliability while selecting candidates to do each service configuration activity. The empirical evaluation of the implemented algorithms on the real-world application data set demonstrated that the solutions generated by GA are more eloquent than those generated by PSO. Hussain et. al [46] showed the multi-objective Quantum-inspired Genetic Algorithm (MQGA) that can be used with quantum computing to reduce the time it takes to make something and the amount of energy it uses while still meeting the deadline. Sefati et. al [85], in the circumstances involving the development of larger services with higher functionalities, GA employs the penalty technique. In this study, GA uses the penalty of violation dependency and incompatibility constraints; however, this does not imply that the approach in cases of violation dependency and incompatibility constraints is discarded entirely. Instead, the GA chooses the right services based on user requirements. The ABC algorithm then assesses the services chosen by GA and merges them if suitable.

We proposed an innovative genetic approach for context/constraint-aware composite services for large repositories of component services. Our proposed method demonstrates the highest performance in terms of feasibility, scalability, and optimality solution for large component service repositories, as determined by an empirical evaluation.

2.5 Summary

The Lucid/GIPSY system's capacity to simulate and execute composite services with equal ease and an automated composition approach with its special constraint-adjustment technique. The inherent concurrency of Lucid's dataflow execution model. GIPSY can store execution results for their context in its warehouse for future reference. In addition, we utilized Gupta's [3] extensible and modular translation framework, which possesses all of the qualities that separate the system described in this dissertation from prior composite service verification methods. In this chapter, we analyzed the unique characteristics of Lucid and GIPSY in depth and compared our suggested solution to other comparable research undertaken in the field to date in order to acquire a comprehensive knowledge of its strengths and limitations. With this knowledge as a starting point, we will look at our research methodology in more depth in the next chapter.

Chapter 3

Genetic Service Composition

As was discussed in Contribution 3 (Section 1.3), the primary objective of this thesis is to optimize the simulation and execution of composite web services with internal constraints on large-scale datasets. However, to accomplish this objective, it is necessary first to construct such services based on a composition request and a set of services available for composition. This is an essential prerequisite for the simulation and execution (Section 1.5). In this thesis, the Execution/Simulation of Context/Constraint-aware Composite Services using GIPSY was borrowed from the research by Laleh et al. [19–23] and Gupta [3], and our novel genetic service composition technique was applied to it to facilitate *service discovery*. *Service discovery* is about proposing a specific set of services that satisfies a demand, which is exactly what the GA algorithm does. However, this *service discovery* is only used to narrow-down the size of possible services for service composition. This chapter discusses this unique composition methodology, the structure of the genetic composite services that it generates, the additions and modifications that we make to complete and optimize this technique, and the specific features that we introduce during its re-implementation to transform it into an independent, flexible, and maintainable application.

3.1 Composite Service Model

Understanding the basic entities and concepts involved in the service composition methodology is crucial. In order to comprehend these concepts we borrowed some definitions from Laleh's research. In this section, we present the formal definitions provided by Laleh in [19] for such entities and concepts.

Definition 1. *A **Service** is a tuple $S = \langle I, O, C, E, QoS \rangle$ where:*

- ***I*** *is the collection of ontology types corresponding to the service's input parameters.*
- ***O*** *is the collection of ontology types that represents the service's output parameters.*
- ***C*** *is the collection of constraint expressions that represent limitations on service capabilities.*
- ***E*** *is the collection of ontology types that represent parameters whose values are modified by the service's execution.*
- ***QoS*** *is the collection of quality parameters for a given service.*

For instance, the components of the Shipment service W_{509} that are presented in Table 6 could be expressed as follows:

- I $= \{var_{17}\}$
- $O = \{var_2\}$
- $C = \{var_{17} = Quebec\}$
- $E = \{var_2\}$
- $QoS = \{\}$

Within the scope of this thesis, we do not consider the QoS features. Consequently, in our implementation of the Service entity, we include a placeholder for QoS parameters; this is done solely to complete the Service structure, which will be needed to verify QoS constraints in the future. Presently, we are only concerned with the constraints imposed on services by their providers, known as internal constraints. Despite the fact that we already defined these constraints in Section 1.1, we now give a more formal definition for them that is based on Laleh's general definition of constraints.

Definition 2. *An **Internal Constraint** is an expression that is either true or false when evaluated. We restrict ourselves to expressions of the form:*
⟨feature⟩ ⟨operator⟩ ⟨literalValue⟩, for simplicity, in which:

- *The term ⟨feature⟩ represents a service input parameter, which is an ontology type.*
- *The term ⟨operator⟩ denotes operators such as* $=, <, >, \leq, \geq$.
- *The term ⟨literalValue⟩ refers to a single value or a collection of values that have the same data type as the expression feature.*

For instance, the internal constraint that was placed on the credit card brand input parameter of the Shipment service W_{508} that was listed in Table 5 would be expressed as $var_{17} = Quebec$.

As described in the Section 1.5 on research methods, a composition request specifying the user's requirements is also required to initiate automated service composition. The following describes this composition request:

Definition 3. *A **Service Composition Request** is represented by the tuple* $R = \langle I, O, QoS, C \rangle$ *in which:*

- ***I** is the collection of ontology types that represents the customer's input.*
- ***O** is the collection of ontology types that represents the customer's desired output.*

- ***QoS*** *is the collection of quality parameters expected by the customer from a service.*

- ***C*** *is the collection of constrains imposed by the service requester.*

Elements of a request to compose the services shown in Table 5 to build a web-based online shopping service, the one in Figure 2 might be similar to this:

- I $= \{var_1\}$

- $O = \{var_2\}$

- $QoS = \{\}$

- $C = \{\}$

Our current implementation of a composition request model does not process the requester's quality of service (QoS) and constraint requirements because, for this thesis, we only consider internal constraints. Currently, they are only included to make the composition model comprehensive. The composition approach used by Gupta generates a set of one or more solution plans in response to a composition request. These *solution plans* are workflows of component services that can produce the requested output by processing the given input while verifying the internal constraints placed on their components. These plans are known as constraint-aware plans and are characterized by the following definition:

Definition 4. *A* ***Constraint-Aware Plan*** *is a directed graph that is derived from the search graph is what is known as a constraint-aware plan. Each node is a service-node* $\langle C_S, service\rangle$*, that uses initial parameters (R.I). Then, the objective parameters are generated by sequentially applying the services of each node (R.O).*

In the preceding definition, search graph refers to the graph of search nodes created during the composition process's genetic algorithm stage. This graph acts as a repository for all possible solution plans that can be used to fulfil a specific composition request (explained further in Section 3.3). In a constraint-aware plan, the set of all service constraints that must be verified before a service can be executed is referred to as C_S for each service-node $\langle C_S, service \rangle$. The input parameters specified as a part of the composition request R are denoted by the notation $R.I.$ In contrast, the output parameters requested are denoted by the notation $R.O.$

Definition 5. *In a constraint-aware plan, the* ***predecessor*** *set of a service-node represents the set of service nodes that must be executed prior to the execution of the service node.*

For the Shipment service W_{508}, for example, the predecessor set would be as follows: $predecessors(W_{508}) = \{W_{590}, W_{904}\}$

3.2 Genetic Service Composition Model: Concepts

It is essential to comprehend the fundamental entities and concepts involved in genetic methodology. We provided some definitions from our research in order to comprehend these concepts.

Definition 6. *A set of web services (nodes in our graph) known as* ***genes*** *characterizes an individual solution. In this study, we defined a gene as a web service from the available web services inside the repository.*

Definition 7. *A* ***genome*** *(or chromosome) is a single solution that contains genetic information utilized by the GA. A genome comprises genes, each representing a single controlling factor represented by the GA. In our model, a genome is a set of web services.*

Definition 8. *The particular arrangement of genes in a solution is a* ***genotype****. In our model, the solutions in the computation space are represented and organized so that a computing system can easily understand and manipulate genotypes.*

Definition 9. *The genetic algorithms simulate the survival of the* ***fittest*** *among genomes of successive generations. The fitness function determines an individual's ability to advantageously compete with other individuals in the population.*

Definition 10. *A **phenotype** is a representation of a composite service in the real-world solution space in which solutions are represented and executed as they are in actual situations. Therefore, it can help us to visualize our graph plan and eventually observe their behavior at runtime.*

Definition 11. *In **uniform crossover**, each gene (web service) is randomly selected from among the corresponding genes of the parent chromosomes and flipped before they are inherited by the offspring.*

Definition 12. *In **single point crossover**, on the parent genome, a crossover point is chosen. All genes beyond that point are exchanged between the two parent genomes. Positional bias distinguishes genes.*

Definition 13. *Each gene has a set of constraints corresponding to the all variable inputs of the web service assigned to it, known as **gene constraints**.*

Definition 14. *If all gene constraints of a gene can match with the prdSet, then we mark this gene as a **composite gene**, and it will be added it to the planning graph.*

Definition 15. *If all gene constraints of a gene can not match with the prdSet, then we mark this gene as a **non composite gene**, and it won't be added it to the planning graph.*

Definition 16. *This technique employs a **penalty function** that reduces the fitness of infeasible solutions. The fitness decreases proportionally to the number of constraints that have been violated. In this manner, infeasible solutions will have a lower chance of surviving during evolution.*

Definition 17. *In a **K-way tournament selection**, k individuals are selected, and a tournament is held between them. Then, only the healthiest candidate is selected and passed on to the next generation. In this manner, many tournaments are held, and the final candidates for the next generation are chosen. It also has a selection pressure parameter, a probabilistic measure of a candidate's likelihood of participating in a tournament. When the tournament size increases, weak candidates have a lower chance of being chosen because they must compete with a stronger candidate.*

Definition 18. *There is a* ***connection*** *between two web services if there is any relation between their input(s) and output(s).*

3.3 Generic Service Composition Algorithms

The problem posed in Section 3.1 is fundamentally an optimization problem with constraints. Genetic Algorithms were utilized in order to overcome these problems. However, genetic Algorithms are only able to solve optimization problems that do not have any constraints attached to them. To handle constraints, they can incorporate some constraint handling methods, such as penalty function, which are introduced in Chapter 2 in detail. For this thesis, we used genetic algorithm to manage constraints on inter-service dependence and conflict by employing penalty function as our evaluator algorithm. This section presents and describes this algorithm alongside the others.

Algorithm 2 drives the service composition process, calling upon the other algorithms as necessary. It includes two major steps: (1) Genetic Algorithm is responsible for building a search graph based on a particular composition request and available services (line 2), (2) a *constructPlan* Algorithm 9 for building for translating solution plans into constraint-aware plans (line 3). The algorithm fails if it cannot make a plan that takes constraints into account for the given composition request.

Algorithm 3 is in charge of making a search graph for the composition request it is given (R) using Genetic Algorithms. Similar to Algorithm 1, a penalty-based GA follows the same phases. Genetic encoding, genetic operators, and a fitness function are the fundamental components of the penalty-based GA. Specifically, the fitness function of the penalty GA includes a penalty strategy that violates constraints on inter-service dependence and conflict (line 3 and line 14).

Algorithm 4 is accountable for the initialization procedure. Initialize an empty array of length equal to length of genome and perform a random assignment from SR as candidate gene. A web service instance is chosen randomly from the SR candidates and bound to the first gene. This procedure repeats until the final gene

Algorithm 2 ServiceComposition

```
Input: $R$ (composition request), $SR$ (set of available services).
Output: $plans$ (a set of constraint-aware plans, or failure).
1: $plans = \emptyset$
2: $searchGraph = GeneticAlgorithm(R, SR)$
3: $plan = constructPlan(searchGraph)$
4: if $(plans \neq \emptyset)$ then
5:     return $cnstr\_plans$
6: else
7:     return $failure$
8: end if
```

Algorithm 3 GeneticAlgorithm

```
Input: $R$ (composition request), $SR$ (set of available services)
Output: $population$ (search graph generated by genetic algorithm, or failure).
1: $searchGraph$ = null; $population$ = null;
2: $population = Initialization(SR)$
3: $fitness\_function = FitnessFunction(population)$
4: while $terminationcondition$ is not true do
5:     for start from $elitismOffset$ to $population.size$ do
6:         $parents = TournamentSelection(population)$
7:         $offsprings = Crossover(parents)$
8:         $population.saveGenomes(offsprings))$
9:     end for
10:    for start from $elitismOffset$ to $population.size$ do
11:        $offsprings = Mutation(offsprings)$
12:        $population.saveGenomes(offsprings))$
13:    end for
14:    $population = Evaluator(population)$
15: end while
16: if $population$ is not null then
17:     return $population$
18: end if
19: return $failure$
```

has been assigned.

Algorithm 5 is responsible for selecting a service from a population of available web services. For tournament selection, a small number of genomes are chosen randomly from the population and put through several *tournaments*. Then, these chosen candidates are given to the next generation. Finally, in a K-way tournament selection, k participants are chosen to compete in a tournament. Only the fittest

Algorithm 4 Initialization

```
Input: $SR$ (set of available services).
Output: $population$ (a set of constraint-aware plans, or failure).
1: $population = Population(populationSize)$
2: for $i = 0$ to $defualtPopulationSize$ do
3:     $genome = newGenome()$
4:     randomly select a web service from $SR$ and save it on $genome$
5:     $population.addGenome(genome)$
6: end for
7: return $population$
```

Algorithm 5 tournamentSelection

```
Input: $population$ (current population), $N$ (size of tournament).
Output: $selection$ (fittest genome among selection).
1: $t\_population$ = Population(N, intialize=false)
2: for $i = 0$ to $N$ do
3:     randomly select a genome from $population$ and save it on $t\_population$
4: end for
5: $selection = t\_population$.getFittest()
6: return $selection$
```

candidate is passed on to the following generation. In this way, many tournaments like this take place, and the best candidates are chosen to move on to the next generation.

Algorithm 6 is the crossover algorithm. Two genomes are chosen randomly from the *Tournament Selection* to cross over to make better children. For a genome with length n, there are *n-1* splitting points. However, selecting some as splitting points will result in genome composition genes after crossover; consequently, the composition instance genes should not be split in a genome. To overcome this problem, in this algorithm, we designed the three policies as follows:

- **Non-Composite Policy**: If parents do not have any composite gene or, in other words, any genes in our genome that have all their inputs will not match with *prdSet*, which means there is no connection among shared input-output nodes. Then, *Uniform Crossover* will apply to this policy. Each gene is randomly chosen from among the corresponding genes of the parent chromosomes.

Algorithm 6 Crossover

```
Input: $Genomes$ (selected genomes as parents for mating).
Output: $genomes$ (mutated genomes as offsprings).
1: if (policy is ‘non-composition’ then
2:     $crossover\_point$ = randomly select a split point from $Genomes$
3:     $genomes = Swap(Genomes, crossover\_point)$
4: else if policy is ‘composition’ then
5:     $crossover\_point$ = keep composition and randomly select a split point from
       other genes in $Genomes$
6:     $genomes = Swap(Genomes, crossover\_point)$
7: else if policy is ‘refusal’ then
8:     continue
9: end if
10: return $genomes$
```

Algorithm 7 Mutation

```
Input: $G$ (Genome).
Output: $genome$ (mutated).
1: $genome$ = randomly select a service instance from $G$ and mark its gene
2: let $i$ be the position of the marked gene
3: let $c$ be a new candidate randomly got from same type selected
4: if policy is ‘adoption’ then
5:     bind $c$ to the selected gene and unmark the corresponding genes
6: else if policy is ‘replace’ then
7:     replace $c$ to the selected gene and unmark the corresponding gene
8: else if policy is ‘refusal’ then
9:     continue
10: end if
11: return $genome$
```

- **Composite Policy**: If the genomes have a composite gene or, in other words, if all input of the single web services matches with the *prdSet*, then we will keep these genomes inside these parents in order to expand our graph and then *Single Point Crossover* will apply to this policy. A split point on the non-composite genes of the parent genome is chosen randomly. Then, the two parent organisms exchange all subsequent data in the non-composite genes.

- **Refusal Policy**: If all genes are composite ones or the genome is selected as the solution candidate, we skip the crossover operation because a solution found.

Algorithm 8 FitnessFunction

```
Input: $G$ (Genome); $GW$ (generation number).
Output: $fitness$ (The fitness value for genome).
1: for each $gene$ inside $G$; $i$ index of gene do
2:     if $gene$ contains invalid constraint then
3:         return $MIN\_VALUE$
4:     else
5:         $fitness = \sum CS_i \; - \; CW_i \; * \; n\_gen \; * \; p\_w$
6:     end if
7: end for
8: return $fitness$
```

Algorithm 7 represents the mutation algorithm. The next generation of a population is kept genetically diverse through mutation. In this algorithm, we designed the three policies as follows:

- **Adoption Policy**: If the genome is not a candidate solution and there are any composite genes inside the genome, we randomly select one gene and replace it with a random web service from SR.
- **Replace Policy**: If all genes inside the genome are composite genes and the genome is not the candidate solution, then we will randomly select one gene and replace it with the random web service from SR. In this policy, we must update *prdSet* after mutation because the connection might not be available in the list.
- **Refusal Policy**: If the genome is selected as the solution candidate, we skip the mutation operation because a solution was found.

Algorithm 8 represents the Fitness Function, a type of objective function that specifies the quality of a solution (that is, a chromosome) in a GA, allowing any chromosome to be ranked against any other candidate chromosome. It is also used to evaluate how well the model solution performs. The fitness of a composite service instance is dependent on its connection weight and the number of constraints that are met. The evolution toward constraint satisfaction is driven by the penalty method. The penalty is defined in line 5, and when the constraint is violated, it adds weight to the total constraints. Then, the fitness function is defined as the weighted sum of the connections (CS_i) and the penalty of a number of involved constraints in *prdSet* (CW_i), where the penalty weight (p_w) increases as the number of generations (n_gen) increases. Thus, in early generations, genomes that violate the constraints but have high utility values are still considered, whereas, in later generations, genomes that violate the constraints are severely punished. CS_i represent composite gene weight, or in other words, if the *prdSet* is a provider list, then all node inputs are satisfied as a consumer. If the gene is marked as the composite instance, then the score value of the genome will be increased. Also, If the connection is available among genes inside a genome, then the composite genes' outputs will be added to the *prdSet* list. So this list for each genome will be updated until the *prdSet* satisfies $R.O$. Then, if the $R.O$ are available in the *prdSet*, the genome is probably a candidate solution in this graph, and the fitness score increases the significant growth.

Algorithm 9 represents the Construct Plan process. It uses Gupta's service composition layered based approach. It includes four major steps: (1) forward expansion is responsible for building a search graph based on a given composition request and available services. (2) using backward search to extract solution plan sets from the search, (3) plan construction is responsible for removing unnecessary services from solution plan sets and organizing the remaining services into workflows or solution plans, (4) construction of constraint-aware plans to turn solution plans into constraint-aware plans with their constraint verification points moved to the best places. The algorithm fails if no constraint-aware plans can be generated for the specified composition request [3].

3.4 Genetic Service Composition Example

This section explains the step-by-step generation of constraint-aware plans for a given composition request and repository of available services through an example. For example, consider the following composition request (R) for the creation of an online shopping service similar to the one represented in Figure 2:

Algorithm 9 ConstructPlan

```
Input: $R$ (composition request), $SR$ (final population which contains a candidate
    solution genome), $R.O$, $R.I$.
Output: $plans$ (a set of constraint-aware plans, or failure).
1: $serviceSet = \emptyset; plans = \emptyset$
2: $searchGraph = ForwardExpansion(R, SR)$
3: repeat
4:   $l$ = maximum layer index in the search graph
5:   $ServiceSet$ = all services in layer $l$ of the search graph
6:   $serviceSet = BackwardSearch(searchGraph, ServiceSet, \emptyset, l)$
7:   $plan = constructPlan(serviceSet)$
8:   if $(plan \notin plans)$ then
9:     $plans = plans \cup plan$
10:  end if
11: until (no more plan can be added to the plans)
12: if $(plans \neq \emptyset)$ then
13:   for (each $plan \in plans$) do
14:     for (each $service \in plan$) do
15:       $serviceNode.service = service$
16:       $serviceNode.C_s = service.C$
17:       $cnstrAwarePlan = cnstrAwarePlan \cup serviceNode$
18:     end for
19:     $cnstrAwarePlan = adjustConstraint(cnstrAwarePlan)$
20:     $cnstr\_plans = cnstr\_plans \cup cnstrAwarePlan$
21:   end for
22:   return $cnstr\_plans$
23: else
24:   return $failure$
25: end if
```

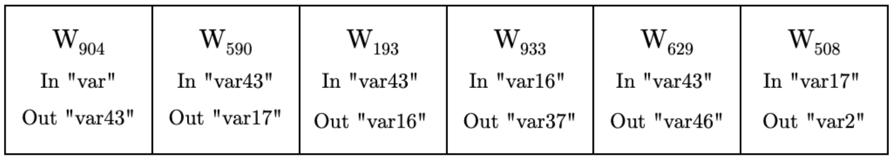
- $R.I = \{var_1\}$
- $R.O = \{var_2\}$
- $R.QoS = \{\}$
- $R.C = \{\}$

Table 5 provides information on the composition's available services (i.e., the repository's SR). The *prdSet* is initially created with the beginning parameters ($R.I$). Then, it gradually adds new connection genes as a result of the services provided by each genome that is added to the graph until it eventually reaches the state where

Table 5: Services Available for Composition of Shopping Application

Service	**Type**	**Input Parameters**	**Sample Input Values**	**Output Parameters**	**Sample Output Values**	**Internal Constraints**
W$_{904}$	Catalog	var$_{1}$	Book	var$_{43}$	ct-212	$C_{904} = \emptyset$
W$_{648}$	Catalog	var$_{1}$	Book	var$_{47}$	ct-765	$C_{648} = \emptyset$
W$_{840}$	Order	var$_{48}$	ct-543	var$_{50}$	od-543	$C_{840} = \emptyset$
W$_{952}$	Order/Payment	var$_{31}$	Bit Coin	var$_{37}$	od-987	C$_{952}$=var$_{31}$=Bit Coin
W$_{224}$	Payment	var$_{18}$	American Express	var$_{33}$	Complete	C$_{224}$=var$_{18}$=American Express
W$_{193}$	Payment	var$_{43}$	Master	var$_{16}$	Complete	C$_{193}$=var$_{43}$=Master
W$_{933}$	Order/Payment	var$_{16}$	Master	var$_{37}$	Complete	C$_{933}$=var$_{16}$=Master
W$_{590}$	Order/Payment	var$_{43}$	Visa	var$_{17}$	Complete	C$_{590}$=var$_{43}$=Visa
W$_{629}$	Order/Payment	var$_{43}$	American Express	var$_{46}$	Complete	C$_{629}$=var$_{43}$=American Express
W$_{356}$	Order	var$_{47}$	ct-126	var$_{4}$	Confirmed	$C_{356} = \emptyset$
W$_{29}$	Shipment	var$_{13}$	Brasil	var$_{32}$	Confirmed	C$_{29}$=var$_{13}$=Brasil
W$_{488}$	Shipment	var$_{38}$	Ontario	var$_{36}$	Confirmed	C$_{488}$=var$_{38}$=Ontario
W$_{813}$	Shipment	var$_{27}$	New York	var$_{32}$	Confirmed	C$_{813}$=var$_{27}$=New York
W$_{508}$	Shipment	var$_{17}$	Quebec	var$_{2}$	Confirmed	C$_{508}$=var$_{17}$=Quebec
W$_{683}$	Payment	var$_{4}$	Ali Pay	var$_{45}$	Confirmed	C$_{683}$=var$_{4}$=Ali Pay

it has all of the goal parameters. Our genetic encoding scheme aims to find solution and generate graph. Genomes represent graph connectivity linearly. (see Figure 10 as *Genotype*). In this example, we present our genetic encoding by representing one of the possible solutions generated by the genetic algorithm (The fittest genome of the last generation). Then we will explain our genetic operators: *crossover* and *mutation*. Each genome contains a list of connection genes, each referring to the connection of two node genes. Node genes provide a list of connectable inputs and outputs. Each connection gene contains information describing the in-node, the out-node, whether the connection gene is expressed (known as a composite). For example, W_{193} accepts var_{43} (produced by W_{904}) and generates var_{16} as output. Also, W_{590} and W_{93} consume var_{43} and generates var_{17} and var_{46}. Therefore, $predecessors = \{var_1, var_{43}, var_{17}, var_{16}, var_{46}\}$ within the search graph.



W$_{904}$	W$_{590}$	W$_{193}$	W$_{933}$	W$_{629}$	W$_{508}$
In "var"	In "var43"	In "var43"	In "var16"	In "var43"	In "var17"
Out "var43"	Out "var17"	Out "var16"	Out "var37"	Out "var46"	Out "var2"

Figure 10: Genome (Genotype)

Figure 11 (Phenotype) displays all such relationships inside the search graph of the Shopping application. W_{508} accepts var_{17} (produced by W_{590}) and produced var_2 and along with this, it will be added to $prdSet$. As $prdSet$ has variable which matches

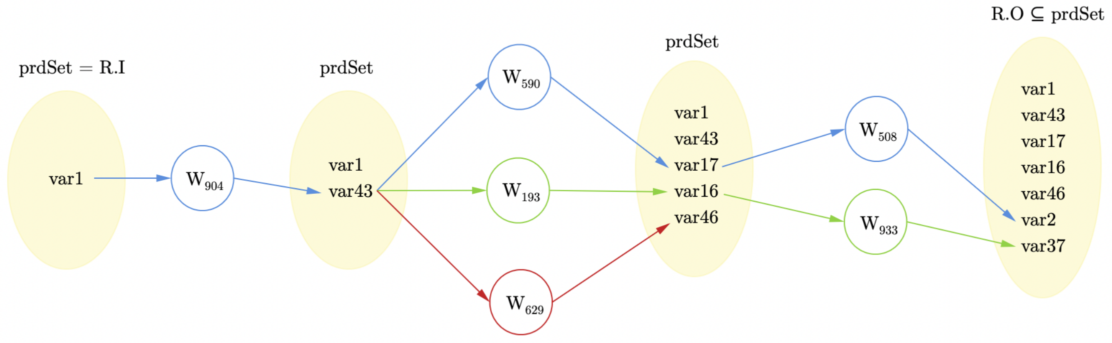


Figure 11: Generated Graph (Phenotype)

with $R.O$, this genome is probably a solution among all generated populations (we will check the validation with the *constructPlan* algorithm in the following steps). To generate such a solution as a genetic approach, we need to explore how genetic operators generated this solution. First we start with the *crossover* operation.

Parent 1	W_{904} var1 → var43 Composite	W_{629} var43 → var46 Composite	W_{193} var43 → var16 Composite	W_{840} var48 → var50 Non Composite	W_{224} var18 → var33 Non Composite	W_{488} var38 → var36 Non Composite
Parent 2	W_{648} var1 → var47 Composite	W_{356} var47 → var4 Composite	W_{683} var4 → var45 Composite	W_{590} var43→ var17 Non Composite	W_{952} var31 → var37 Non Composite	W_{813} var27 → var32 Non Composite

Offspring	W_{904} var1 → var43 Composite	W_{629} var43 → var46 Composite	W_{193} var43 → var16 Composite	W_{590} var43 → var17 Composite	W_{952} var31 → var37 Non Composite	W_{813} var27 → var32 Non Composite

Figure 12: Crossover (Composition Policy)

Consider $parent_1$ and $parent_2$ as two chromosomes selected among the population by the tournament selection. As we described the crossover algorithm in Figure 12, the algorithm will select the *Composition Policy* because we have composite web services inside the $parent_1$. So, it will keep the composite nodes as part of the solution path and select the split point among the non-composite genes randomly.

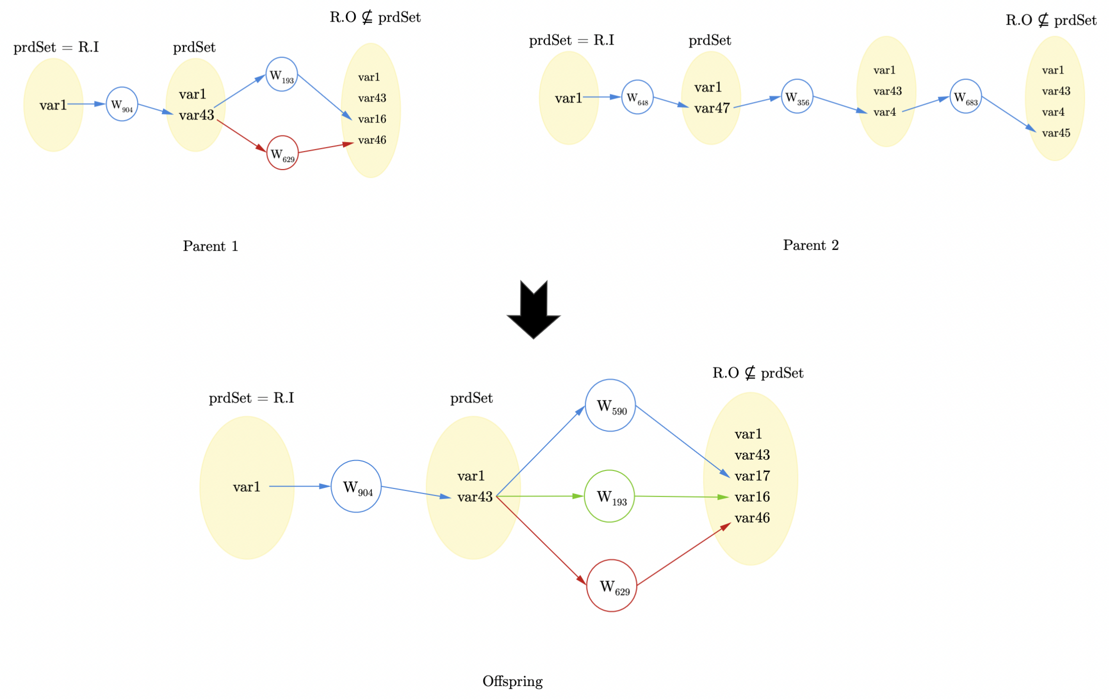


Figure 13: Crossover (Composition Policy) - Phenotype

Then, the newly generated offspring chromosome, as shown in Figure 12, consists of $parent_1$'s composite genes and the non-composite genes swapped from $parent_2$. After evaluation at the end of this generation, the fitness function found the new connection in the *offspring* genome and marked it as a composite gene. Therefore, as shown in the Figure 13, the *prdSet* will be: $predecessors = \{var_1,\ var_{43},\ var_{17},\ var_{16},\ var_{46}\}$ within the search graph.

The next step will be mutating the genomes among the population. Suppose the *offspring* chromosome, which the crossover algorithm has generated. As the chromosome has composite genes, the adoption policy applied for this mutation. As shown in Figure 14, the mutation algorithm replaced one of the non-composite genes with random web services from SR. This operator allows the search algorithm to improve its generation and facilitate to reach of the goal. After evaluation at the end of this generation, the fitness function found the new connection in the mutated genome and marked it as a composite gene. Therefore, as shown in the Figure 15, the *prdSet* will be: $predecessors = \{var_1,\ var_{43},\ var_{17},\ var_{16},\ var_{46},\ var_{37}\}$ within

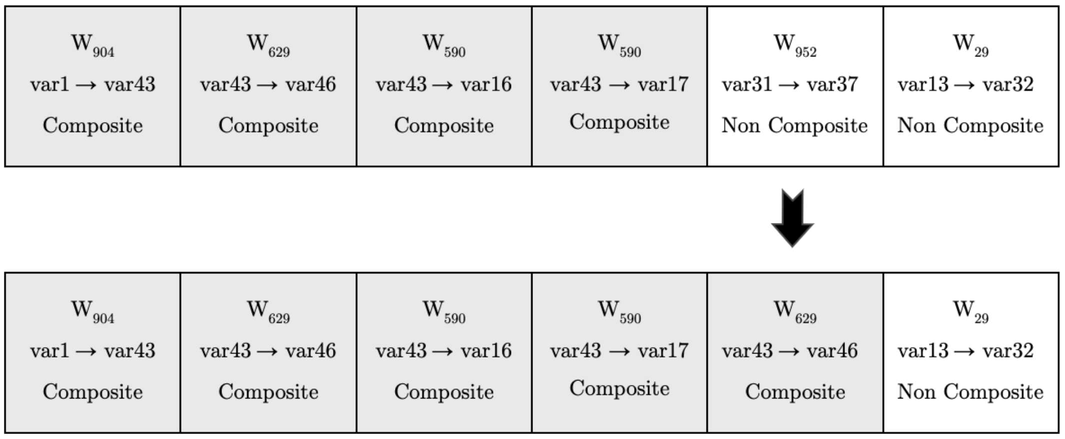

W_{904} var1 → var43 Composite
W_{629} var43 → var46 Composite
W_{590} var43 → var16 Composite
W_{590} var43 → var17 Composite
W_{952} var31 → var37 Non Composite
W_{29} var13 → var32 Non Composite
W_{904} var1 → var43 Composite
W_{629} var43 → var46 Composite
W_{590} var43 → var16 Composite
W_{590} var43 → var17 Composite
W_{629} var43 → var46 Composite
W_{29} var13 → var32 Non Composite

Figure 14: Mutation Adoption Policy - Genotype

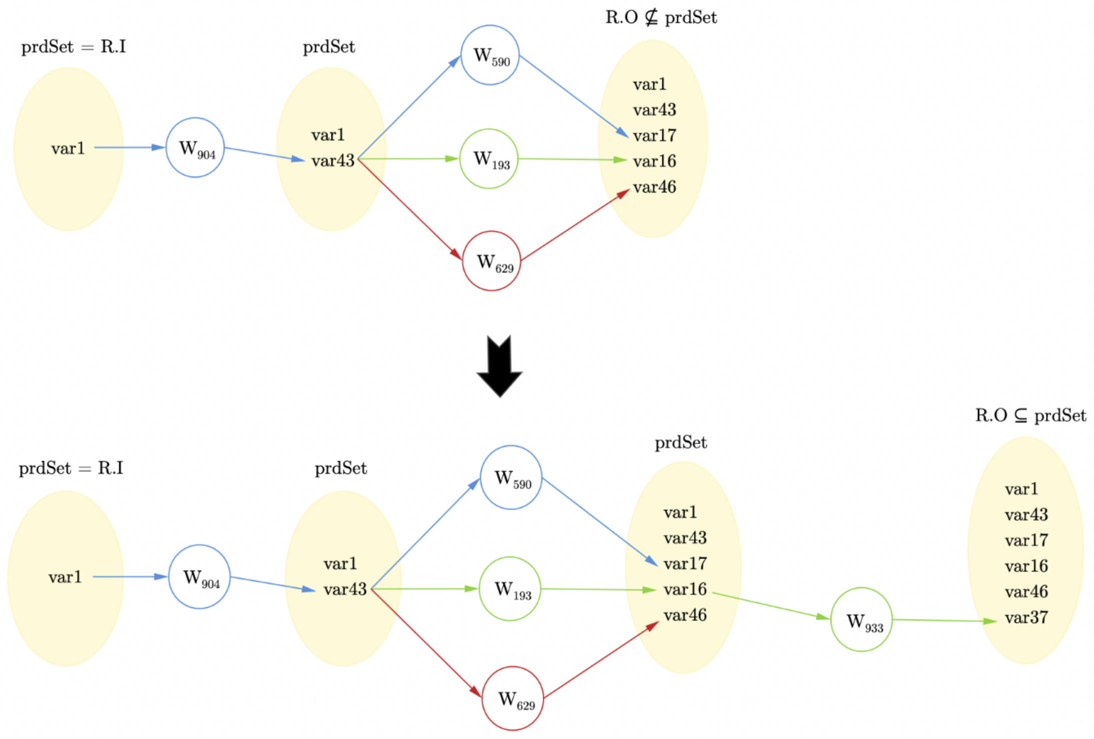
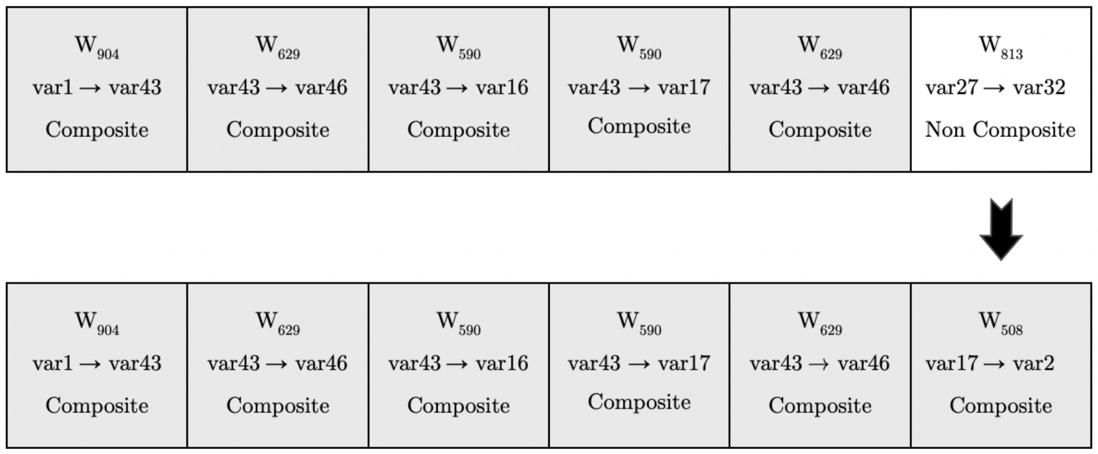


Figure 15: Mutation Adoption Policy - Phenotype

the search graph. We consider this mutated chromosome as *elitism* genome and we save it for the next genration.

In the next generation, this genome as *elitism* has a high probability of selecting as the candidate for mating with *crossover*. Then, the genome will pass to the mutation algorithm. Again, as the chromosome has composite genes, the adoption policy



W904 var1 → var43 Composite
W629 var43 → var46 Composite
W590 var43 → var16 Composite
W590 var43 → var17 Composite
W629 var43 → var46 Composite
W813 var27 → var32 Non Composite
W904 var1 → var43 Composite
W629 var43 → var46 Composite
W590 var43 → var16 Composite
W590 var43 → var17 Composite
W629 var43 → var46 Composite
W508 var17 → var2 Composite

Figure 16: Mutation Adoption Policy - Genotype

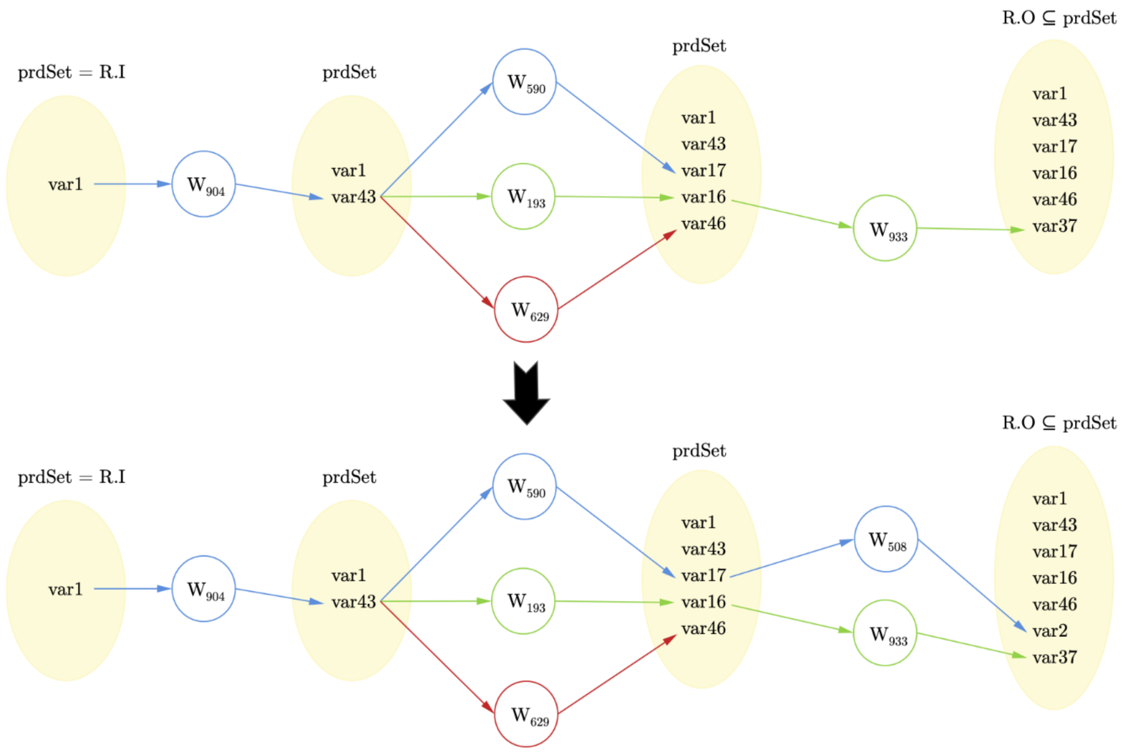


Figure 17: Mutation Adoption Policy - Phenotype

applied for this mutation. As shown in Figure 16, the mutation algorithm replaced a non-composite gene with web services from SR. After evaluation at the end of this generation, the fitness function found the new connection in the mutated genome and marked it as a composite gene. This time, the fitness function found var_2 in prdSet, produced by W_{508}. Given that $prdSet$ contains a variable corresponding to $R.O$, this genome is most likely a solution among all generated populations. Therefore, as shown in the Figure 17, the *prdSet* will be: $predecessors = \{var_1, var_{43}, var_{17}, var_{16}, var_{46}, var_{37}, var_2\}$ within the search graph. This search graph is utilized by the construct plan phase of the composition process to generate service sets that may be converted into solution plans.

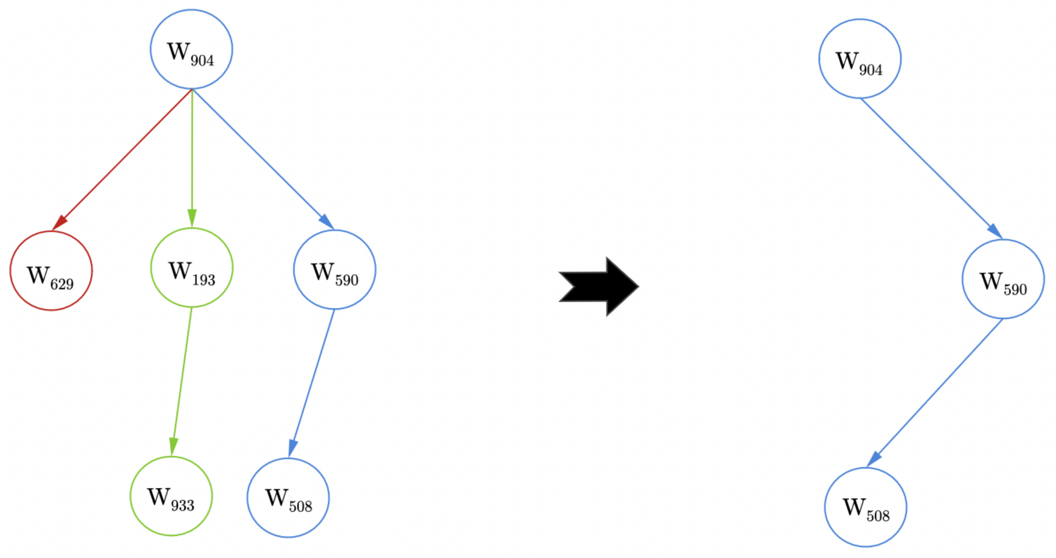


Figure 18: Shopping Service Solution Plan Set Construction

Figure 18 depicts the generation of such plan sets when backtracking begins from a goal associated with R.O. In the next step, the algorithm looks for web service(s) inside the genome, which provides W_{508}'s inputs. As W_{590} accepts var_{43} (produced by W_{904}) and generates var_{17} as output. So it will be added to the plan set. In the next step, W_{904}accepts var_1 and produce var_{43}. So, our solution is validated, and it will be added to the plan sets. These example strategies can be applied to all genomes, which are solution candidates after the final generation generated by the genetic algorithm.

3.5 Restriction on Service Composition

The planning-graph-based approach to service composition employed by the genetic algorithm (see Section 3.3) restricts the addition of services to a search graph, which has a substantial effect on the resulting graph and, consequently, on the final constraint-aware composition plans. We describe this limitation in this section, along with its justification and consequences.

A service may only be added to a genome during *initialization* or genetic operators (*crossover* and *mutation*) if it generates at least one output parameter that does not already exist in the *prdSet* at the time the service must be added to the genome. The justifications for this restriction are outlined below:

- **Avoiding numerous similar service instances to reduce execution effort**: Without this restriction, the same service, for instance, W_{904}, is added multiple times to the same genome when considering a hypothetical search graph. If this scenario results in a solution plan containing both instances of W_{904}, resources will be wasted by executing W_{904} twice to calculate the same set of parameters.

- **Preventing redundant genome construction for the same input**: Without this restriction, each time the repository is searched to find services for a new connection, the services that were included in the search graph in a previous iteration will be scored again and deemed eligible to increase their previously calculated fitness function value. This is merely redundant processing of services for the same input that has already been added to the *prdSet*, which accounts for a significant amount of processing time.

- **Preventing the limitless propagation of the genetic operations**: In the absence of this restriction, every eligible service in the service repository will be lost in every iteration, regardless of whether it is already present in the search graph. In other words, each iteration will add one or more services to the graph. As a result, the termination condition of the composition service process will never be met, resulting in the process entering an infinite loop once all possible component services for a composition request have been added.

3.6 Service Composition Implementation

The service composition algorithms that were presented in the previous section will be discussed in this section, along with their respective implementations. We need the composition application to work as a generic (not scenario-specific) tool that can take any valid composition request and a large-scale service repository and generate a set of possible constraint-aware solution plans in a format that can be used as input to Gupta's composite service translation framework [3]. In addition, our re-implementation includes some important optimizations implemented at every stage of the composition process to improve the quality of its products, reduce the processing work required, and increase its dependability and efficiency. Finally, validation checks, error logging and handling, an extensible framework for user's request, storage and reusing composed services, and a graphical interface have been implemented to make the composition application more robust, reliable, flexible, and maintainable from a software engineering standpoint. Section 3.6.2 and Section 3.6.4 describe these additional features in depth.

3.6.1 Assumptions

Before we can explore the qualities that differentiate our implementation of the service composition technique from the Laleh and Gupta approach [3], it is necessary to establish the assumptions upon which this implementation is based explicitly. They are as follows:

- Since such a solution does not qualify as a composition of (many) services, any intermediate (search graph, plan set, etc.) or final (constraint-aware plan) solution generated for a composition problem that consists of only one service from the service repository is deemed invalid.

- All service constraints are expected to be unique during constraint-aware plan creation. Although the restrictions represented by two unique `Constraint` objects (Java class specified in [3]) objects are similar, they are considered distinct. Therefore, they will be independently validated during the simulation and execution of composite services. Furthermore, in order to reduce redundancy, several copies of the same `Constraint` object cannot be associated with a single service node.

3.6.2 Validation Checks

As part of our approach, we include all Gupta's validation checks [3] at critical points in the composition process. In addition, we perform additional checks on a composition request received from a user to ensure that it includes all of the essential information in the anticipated format before enabling any services to be composed to resolve it. The process is terminated immediately if any of these tests fail. Following is a list of the significant design characteristics of this class, followed by the respective additional validation checks:

- Each genome must have a *prdSet* list with unique dependencies variables. If the *prdSet* contains the duplicated variable, they must be removed from the list.
- Solutions to the composition problem at any stage (search graph, plan set, etc.) that only involve one service must be disregarded as invalid because they do not satisfy the definition of a composition of services.

3.6.3 Optimizations

We re-implement the service composition algorithms and introduce several changes that can improve the efficiency, while lowering the processing effort required. This section describes the optimizations implemented at each stage of the process, their justification, or, in other words, their influence on the process. This modification is intended to permit the generation of additional alternative composition solutions.

3.6.3.1 Service Composition

Following are distinctions between Service Composition (Algorithm 2) and its implementation that serve to optimize the implemented process:

- We refactored Gupta's implementation and added methods which enable users to generate graphs by the genetic algorithm and then use Gupta's composition process to convert it to the layered-based composition services. This modification makes a new algorithm comparable to the previous method, which was based on layer-based techniques.

- If the composition process is successful, we can use Gupta's composition process to generate layer-based composited services that are consumable by translation of these services into various practical formats, such as Objective Lucid programs that can be simulated/executed on GIPSY, if these services are translated into these formats.

3.6.3.2 Service Composition Driver

The service composition driver is in charge of prompting the user for the inputs needed to execute the service composition process, activating the various phases involved in the process in the correct order, and showing the process's ultimate status (success/failure) on the console.

- We added the genetic algorithm as an argument on the console, so the user can run the with/without options as a choice in the start mode.

- Initiate the service composition process, sending the request parameters and logger objects produced using a genetic approach.

3.6.4 Additional Features

So that our service composition implementation is more flexible and can be used as a tool or application, we add some features that were not in the original implementation (by Gupta) of the composition algorithms shown in Section 3.3. This section describes these additional features and the architecture associated with them.

3.6.4.1 Plot Graph Tool

The graph tool module offers a `Graph` class and some algorithms that work on it. We used XChart, a lightweight and convenient library for plotting data that reduces the time required to go from data to chart and removes the guesswork from changing the chart style [87]. This tool has proven to be very helpful for the visualization of planning-graph-based service composition. For instance, Figure 19 illustrates a planning graph generated by this module.

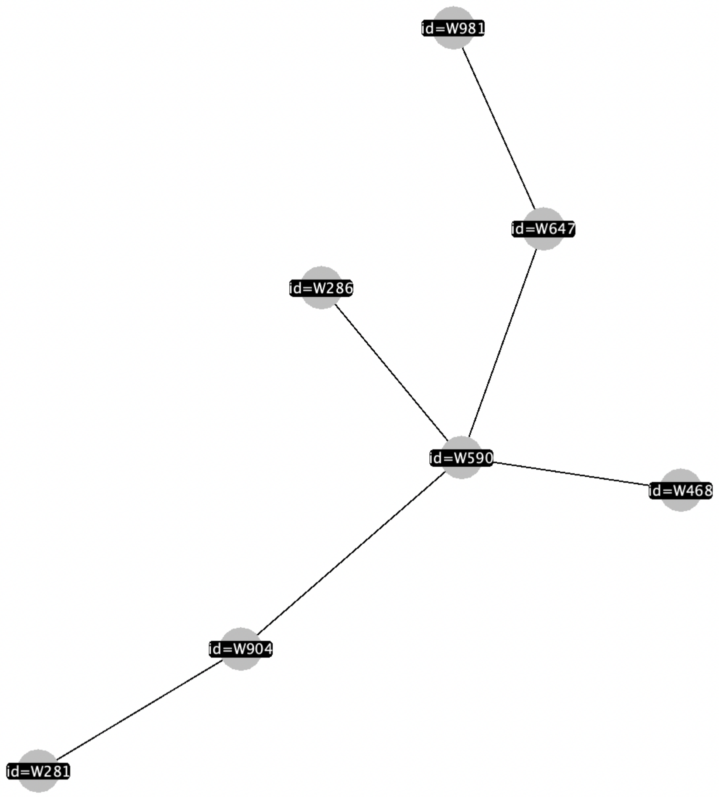


Figure 19: GA Graph Represented Using the Plot Graph Tool.

3.6.4.2 Service Composition Dataset Generator

We developed a tool that generates a service repository based on a specified statistical distribution of the features of the initial service population. For instance, the following statistical criteria must be considered after a graph is created:

- number of services in the repository and generated graph.
- minimum/maximum/average of inputs/outputs/effects/constraints per service.
- A percentage of variables having constraints.
- number of different variables are used in the entire repository and generated graph.

This dataset generator has been extensively used in our evaluation process.

3.7 Summary

In this chapter, we described the novel planning-graph-based service composition and constraint-adjustment method developed by genetic algorithm and the enhancements we introduce over Gupta and Laleh et al. This enhanced method is utilized in our study to generate constraint-aware composite services on large-scale repositories that can be converted into Objective Lucid applications suited for execution on GIPSY.

Chapter 4

Solution Evaluation

In Chapter 3, we have described the process of composing constraint- and context-aware services in response to a composition request using Genetic Algorithms. In this chapter, an evaluation of this web service composition technique is provided. In this chapter, we review the tests and analyses conducted on the proposed solution and the conclusions that can be derived from them to establish whether the solution fully satisfies the requirements for which it was built.

4.1 Simulation Settings

The experiments were carried out on a personal computer equipped with an Apple M1 chip, 8GB RAM, macOS Monterey, and Java Standard Edition V1.8.0 to assess the scalability and effectiveness of the proposed approaches. IntelliJ IDEA enables real-time monitoring of a process's performance statistics. We used the Profiler [88], which measures resource consumption, identifies resource-related bottlenecks, and determines how certain events impact program performance. The following metrics are available:

- **CPU:** The amount of CPU use by the selected process. Each procedure has a unique figure.
- **Heap memory:** The amount of memory currently being used and the

maximum heap size. When new objects of reference types are allocated, the heap size grows, and it shrinks when they are garbage-collected. The -Xmx option specifies the maximum heap size.

- **Threads:** the total number of threads (including daemon threads). The number after the slash represents the maximum number of threads since the process began.

- **Non-heap memory:** This type of memory is used to store JVM objects and structures that are required for the JVM to function. The first value is the current memory value, and the second is the maximum value since the charts began.

4.2 Service Composition Process Evaluation

As mentioned in Section 1.2, the primary objective of this thesis is to build an operational service composition mechanism capable of generating one or more constraint- and context-aware composite services as solutions to a valid composition problem based on the services available for usage as components during composition on a large-scale repository. In this section, we describe the evaluation technique used to confirm that this application satisfies all of its functional requirements, i.e., that it meets the first objective of our thesis. As described in Chapter 3, our technique for composing services consists of multiple phases: initialization, tournament selection, cross over, mutation, evaluation, and service composition. Each of these steps must satisfy a set of precise parameters and exhibit certain features during its processing to achieve its intended purpose. As part of our evaluation methodology, we compile detailed lists of all such distinguishing qualities and test each stage individually. The entire process guarantees that all requirements are met. While we recognize that scenario-based testing is not absolute proof of the absence of faulty behaviour and that its effectiveness is contingent on the thoroughness with which test cases are designed, the complexities of our composition process and time limitation prevent us from preparing a full-fledged mathematical proof evaluating each constituent operation and possible scenario. By using a diligent and methodical approach to the design and execution of test cases, however, we make every effort to ensure that all critical aspects of the composition process are adequately tested.

4.2.1 Scalability Evaluation

Scalability is one of the primary challenges for constraint-aware web service composition. This experiment was designed to test the scalability of the solution space by measuring the computation time and space required by each algorithm to find at least one solution. We designed eighteen tests for the evaluation of scalability in the solution space. So each test has statistical factors based on our repository generator shown in below tables. The factors include:

- The number of services in the repository
- The minimum/maximum inputs/outputs per service
- The percentage of variables having constraints
- The number of different variables used in the entire repository

The required computation time, space, the number of solutions and the success of service composition of the GA-based method(the results are an average of three times run for each test) and forward expansion are shown in the below tables.

Min-Input	**Max - Input**	**Min-Output**	**Max-Output**	**Number of Variables Used**	**Percentage of Variables Having Constraints**
1	1	1	1	10	50%

Table 6: Statistical Factors for Test 1

Number of Web Services	**Time - FE(ms)**	**Avg Time - GA(ms)**	**Solution(s) - FE**	**Solution(s) - GA - Min**	**Solution(s) - GA - Max**	**Max Memory Heap(MB) - FE**	**Max Memory Heap(MB) - GA**	**Service Composition - FE**	**Service Composition - GA**
10	109	326	1	1	1	15	15	Successful	Successful
100	511	416	4	1	4	255	15	Successful	Successful
1000	1573	432	52	9	19	413	28	Successful	Successful
10000	7248	1845	12	1	4	1437	80	Successful	Successful

Table 7: Evaluation Summary for Test 1

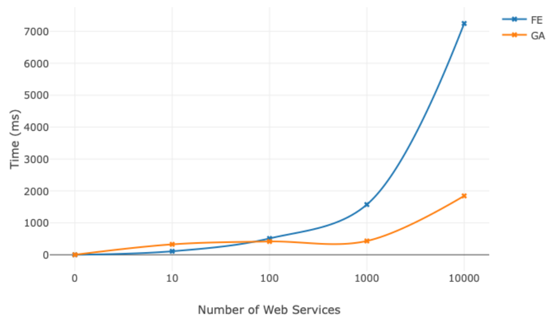


Figure 20: Time vs.
Number of Web Services

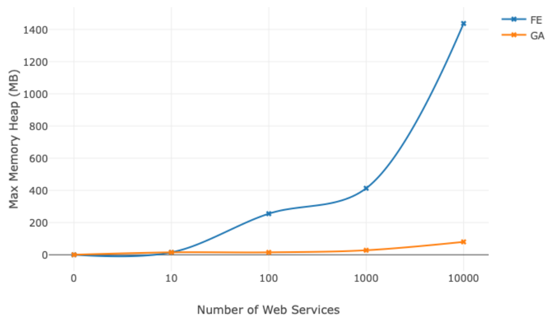


Figure 21: Memory Heap vs.
Number of Web Services

Min-Input	Max - Input	Min-Output	Max-Output	Number of Variables Used	Percentage of Variables Having Constraints
1	1	1	1	10	100%

Table 8: Statistical Factors for Test 2

Number of Web Services	Time - FE(ms)	Avg Time - GA(ms)	Solution(s) - FE	Solution(s) - GA - Min	Solution(s) - GA - Max	Max Memory Heap(MB) - FE	Max Memory Heap(MB) - GA	Service Composition - FE	Service Composition - GA
10	112	341	1	1	1	15	15	Successful	Successful
100	412	442	7	1	3	210	15	Successful	Successful
1000	-	594	-	1	5	1787	42	Failed	Successful
10000	-	1877	-	1	7	1787	90	Failed	Successful

Table 9: Evaluation Summary for Test 2

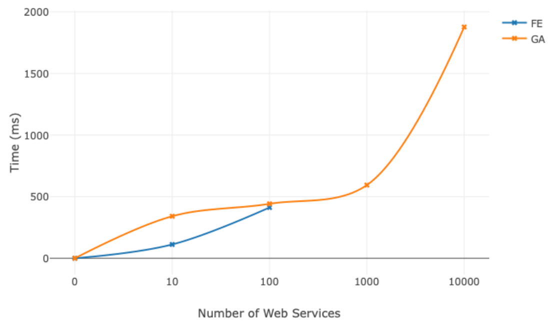


Figure 22: Time vs.
Number of Web Services

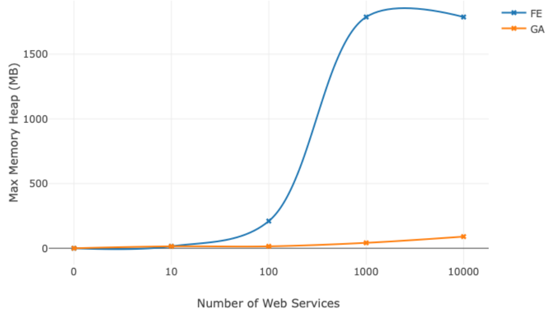


Figure 23: Memory Heap vs.
Number of Web Services

Min-Input	Max - Input	Min-Output	Max-Output	Number of Variables Used	Percentage of Variables Having Constraints
1	1	1	1	20	50%

Table 10: Statistical Factors for Test 3

Number of Web Services	Time - FE(ms)	Avg Time - GA(ms)	Solution(s) - FE	Solution(s) - GA - Min	Solution(s) - GA - Max	Max Memory Heap(MB) - FE	Max Memory Heap(MB) - GA	Service Composition - FE	Service Composition - GA
10	94	321	1	1	1	15	15	Successful	Successful
100	498	463	5	1	3	80	28	Successful	Successful
1000	-	764	-	11	16	1787	52	Failed	Successful
10000	-	1764	-	14	32	1787	100	Failed	Successful

Table 11: Evaluation Summary for Test 3

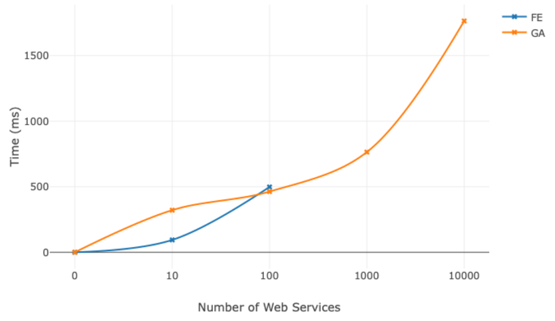


Figure 24: Time vs. Number of Web Services

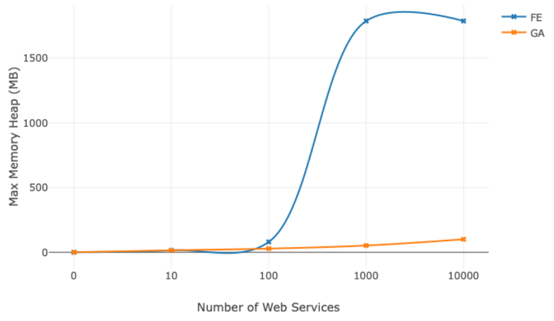


Figure 25: Memory Heap vs. Number of Web Services

Min-Input	Max - Input	Min-Output	Max-Output	Number of Variables Used	Percentage of Variables Having Constraints
1	1	1	1	20	100%

Table 12: Statistical Factors for Test 4

Number of Web Services	Time - FE(ms)	Avg Time - GA(ms)	Solution(s) - FE	Solution(s) - GA - Min	Solution(s) - GA - Max	Max Memory Heap(MB) - FE	Max Memory Heap(MB) - GA	Service Composition - FE	Service Composition - GA
10	91	373	1	1	1	15	15	Successful	Successful
100	410	527	5	1	5	72	28	Successful	Successful
1000	-	641	-	1	3	1787	60	Failed	Successful
10000	-	978	-	1	5	1787	90	Failed	Successful

Table 13: Evaluation Summary for Test 4

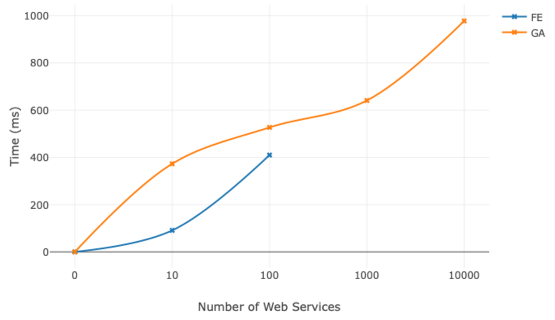


Figure 26: Time vs. Number of Web Services

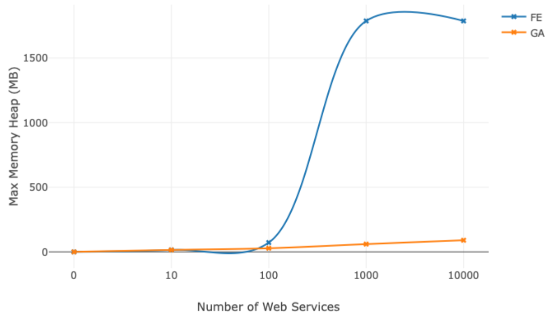


Figure 27: Memory Heap vs. Number of Web Services

Min-Input	Max - Input	Min-Output	Max-Output	Number of Variables Used	Percentage of Variables Having Constraints
1	1	1	1	50	50%

Table 14: Statistical Factors for Test 5

Number of Web Services	Time - FE(ms)	Avg Time - GA(ms)	Solution(s) - FE	Solution(s) - GA - Min	Solution(s) - GA - Max	Max Memory Heap(MB) - FE	Max Memory Heap(MB) - GA	Service Composition - FE	Service Composition - GA
10	109	402	1	1	1	15	15	Successful	Successful
100	399	452	6	2	6	80	28	Successful	Successful
1000	-	676	-	14	18	1787	58	Failed	Successful
10000	-	1079	-	1	1	1787	96	Failed	Successful

Table 15: Evaluation Summary for Test 5

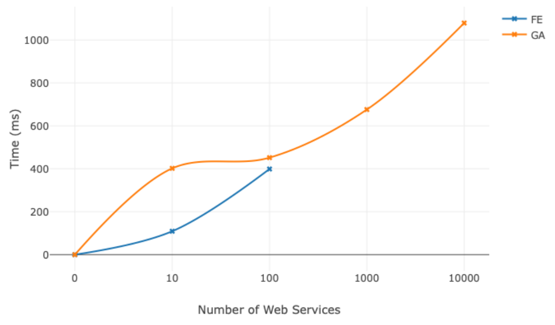


Figure 28: Time vs. Number of Web Services

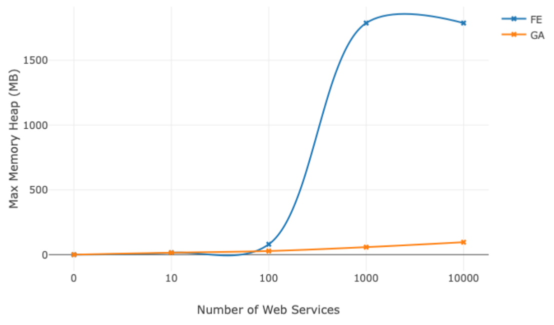


Figure 29: Memory Heap vs. Number of Web Services

Min-Input	Max - Input	Min-Output	Max-Output	Number of Variables Used	Percentage of Variables Having Constraints
1	1	1	1	50	100%

Table 16: Statistical Factors for Test 6

Number of Web Services	Time - FE(ms)	Avg Time - GA(ms)	Solution(s) - FE	Solution(s) - GA - Min	Solution(s) - GA - Max	Max Memory Heap(MB) - FE	Max Memory Heap(MB) - GA	Service Composition - FE	Service Composition - GA
10	99	412	1	1	1	15	15	Successful	Successful
100	431	502	8	3	4	64	30	Successful	Successful
1000	-	622	-	10	12	1787	60	Failed	Successful
10000	-	1644	-	22	32	1787	100	Failed	Successful

Table 17: Evaluation Summary for Test 6

Min-Input	Max - Input	Min-Output	Max-Output	Number of Variables Used	Percentage of Variables Having Constraints
1	5	1	5	10	50%

Table 18: Statistical Factors for Test 7

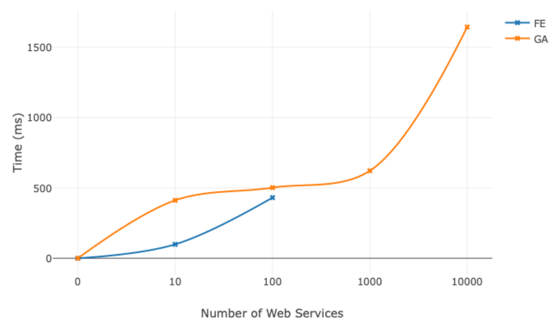


Figure 30: Time vs. Number of Web Services

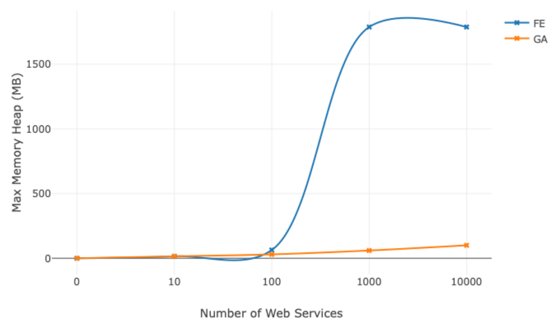


Figure 31: Memory Heap vs. Number of Web Services

Number of Web Services	Time - FE(ms)	Avg Time - GA(ms)	Solution(s) - FE	Solution(s) - GA - Min	Solution(s) - GA - Max	Max Memory Heap(MB) - FE	Max Memory Heap(MB) - GA	Service Composition - FE	Service Composition - GA
10	100	326	1	1	1	15	15	Successful	Successful
100	418	461	10	1	5	78	32	Successful	Successful
1000	-	665	-	1	10	1787	57	Failed	Successful
10000	-	1316	-	16	31	1787	94	Failed	Successful

Table 19: Evaluation Summary for Test 7

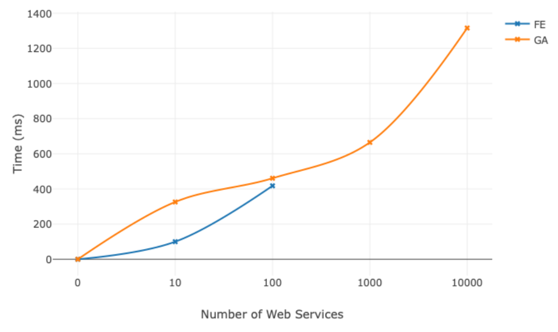


Figure 32: Time vs. Number of Web Services

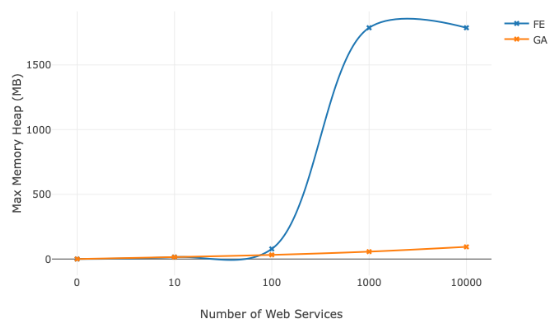


Figure 33: Memory Heap vs. Number of Web Services

Min-Input	Max - Input	Min-Output	Max-Output	Number of Variables Used	Percentage of Variables Having Constraints
1	5	1	5	10	100%

Table 20: Statistical Factors for Test 8

Number of Web Services	Time - FE(ms)	Avg Time - GA(ms)	Solution(s) - FE	Solution(s) - GA - Min	Solution(s) - GA - Max	Max Memory Heap(MB) - FE	Max Memory Heap(MB) - GA	Service Composition - FE	Service Composition - GA
10	112	344	1	1	1	15	15	Successful	Successful
100	464	455	11	4	8	57	31	Successful	Successful
1000	-	666	-	8	13	1787	65	Failed	Successful
10000	-	1419	-	19	46	1787	94	Failed	Successful

Table 21: Evaluation Summary for Test 8

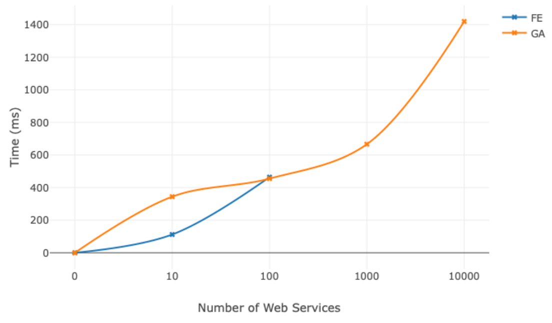


Figure 34: Time vs.
Number of Web Services

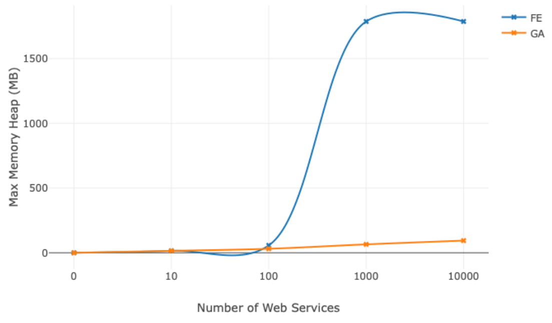


Figure 35: Memory Heap vs.
Number of Web Services

Min-Input	**Max - Input**	**Min-Output**	**Max-Output**	**Number of Variables Used**	**Percentage of Variables Having Constraints**
1	5	1	5	20	50%

Table 22: Statistical Factors for Test 9

Number of Web Services	**Time - FE(ms)**	**Avg Time - GA(ms)**	**Solution(s) - FE**	**Solution(s) - GA - Min**	**Solution(s) - GA - Max**	**Max Memory Heap(MB) - FE**	**Max Memory Heap(MB) - GA**	**Service Composition - FE**	**Service Composition - GA**
10	106	368	1	1	1	15	15	Successful	Successful
100	132	514	9	3	5	54	28	Successful	Successful
1000	-	638	-	6	14	1787	63	Failed	Successful
10000	-	1493	-	15	27	1787	99	Failed	Successful

Table 23: Evaluation Summary for Test 9

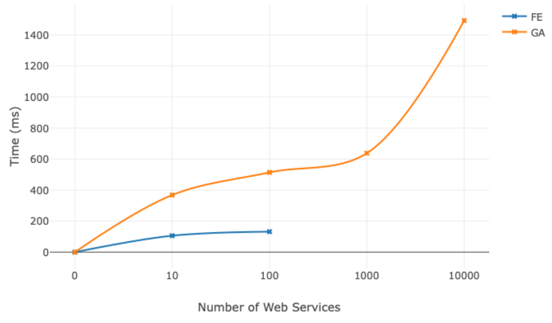


Figure 36: Time vs.
Number of Web Services

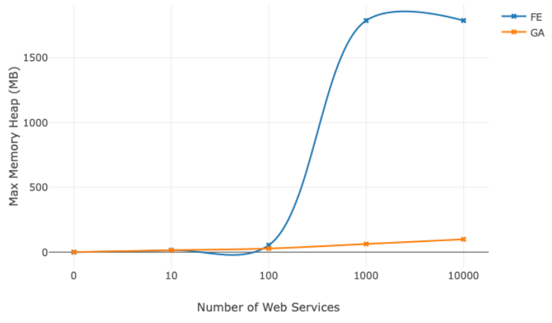


Figure 37: Memory Heap vs.
Number of Web Services

Min-Input	**Max - Input**	**Min-Output**	**Max-Output**	**Number of Variables Used**	**Percentage of Variables Having Constraints**
1	5	1	5	20	100%

Table 24: Statistical Factors for Test 10

Number of Web Services	Time - FE(ms)	Avg Time - GA(ms)	Solution(s) - FE	Solution(s) - GA - Min	Solution(s) - GA - Max	Max Memory Heap(MB) - FE	Max Memory Heap(MB) - GA	Service Composition - FE	Service Composition - GA
10	97	732	1	1	1	15	15	Successful	Successful
100	124	642	5	1	5	52	28	Successful	Successful
1000	-	748	-	1	12	1787	62	Failed	Successful
10000	-	1695	-	1	23	1787	96	Failed	Successful

Table 25: Evaluation Summary for Test 10

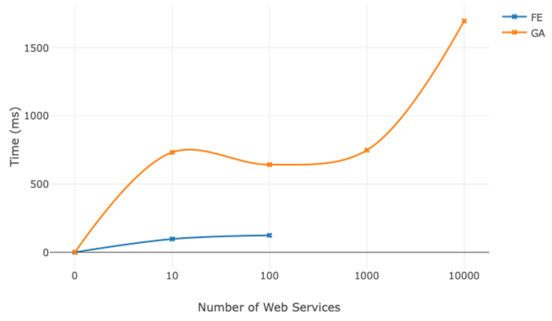


Figure 38: Time vs. Number of Web Services

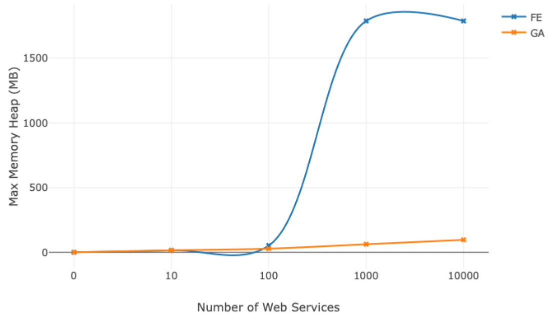


Figure 39: Memory Heap vs. Number of Web Services

Min-Input	Max - Input	Min-Output	Max-Output	Number of Variables Used	Percentage of Variables Having Constraints
1	5	1	5	50	50%

Table 26: Statistical Factors for Test 11

Number of Web Services	Time - FE(ms)	Avg Time - GA(ms)	Solution(s) - FE	Solution(s) - GA - Min	Solution(s) - GA - Max	Max Memory Heap(MB) - FE	Max Memory Heap(MB) - GA	Service Composition - FE	Service Composition - GA
10	119	688	1	1	1	15	15	Successful	Successful
100	154	724	5	1	3	72	44	Successful	Successful
1000	-	748	-	1	4	1787	78	Failed	Successful
10000	-	1471	-	1	3	1787	124	Failed	Successful

Table 27: Evaluation Summary for Test 11

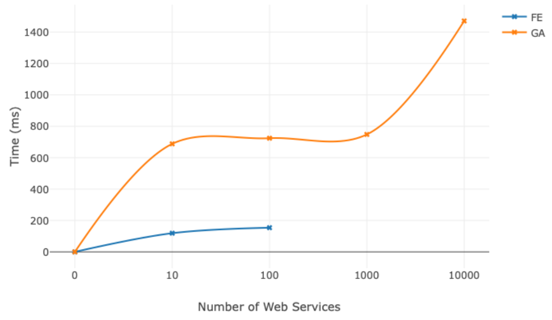


Figure 40: Time vs. Number of Web Services

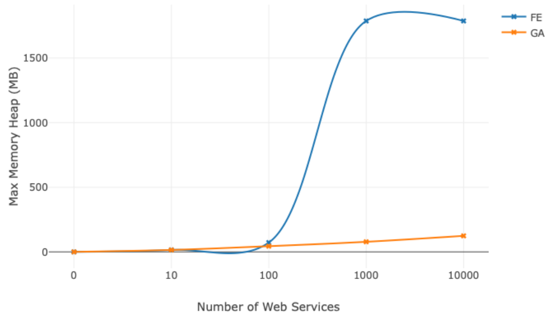


Figure 41: Memory Heap vs. Number of Web Services

Min-Input	Max - Input	Min-Output	Max-Output	Number of Variables Used	Percentage of Variables Having Constraints
1	5	1	5	50	100%

Table 28: Statistical Factors for Test 12

Number of Web Services	Time - FE(ms)	Avg Time - GA(ms)	Solution(s) - FE	Solution(s) - GA - Min	Solution(s) - GA - Max	Max Memory Heap(MB) - FE	Max Memory Heap(MB) - GA	Service Composition - FE	Service Composition - GA
10	115	758	1	1	1	15	15	Successful	Successful
100	161	774	7	1	3	75	48	Successful	Successful
1000	-	782	-	1	5	1787	81	Failed	Successful
10000	-	1778	-	1	4	1787	132	Failed	Successful

Table 29: Evaluation Summary for Test 12

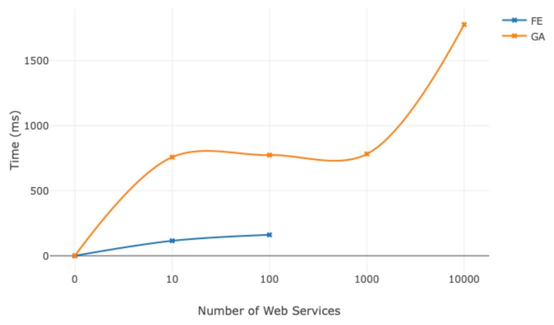


Figure 42: Time vs. Number of Web Services

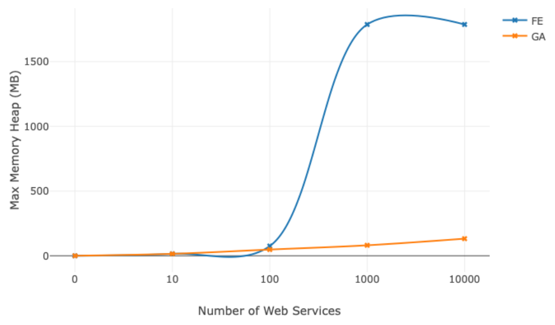


Figure 43: Memory Heap vs. Number of Web Services

Min-Input	Max - Input	Min-Output	Max-Output	Number of Variables Used	Percentage of Variables Having Constraints
1	10	1	10	10	50%

Table 30: Statistical Factors for Test 13

Number of Web Services	Time - FE(ms)	Avg Time - GA(ms)	Solution(s) - FE	Solution(s) - GA - Min	Solution(s) - GA - Max	Max Memory Heap(MB) - FE	Max Memory Heap(MB) - GA	Service Composition - FE	Service Composition - GA
100	-	774	-	1	5	1787	42	Failed	Successful
1000	-	812	-	1	4	1787	86	Failed	Successful
10000	-	1552	-	1	1	1787	142	Failed	Successful

Table 31: Evaluation Summary for Test 13

Min-Input	Max - Input	Min-Output	Max-Output	Number of Variables Used	Percentage of Variables Having Constraints
1	10	1	10	10	100%

Table 32: Statistical Factors for Test 14

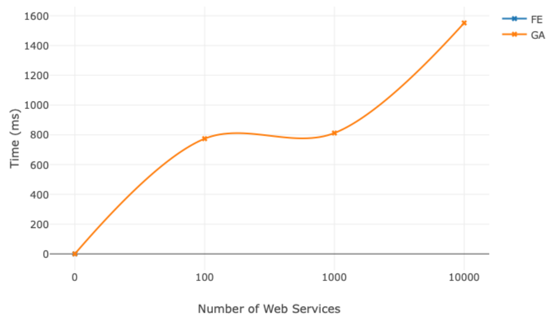


Figure 44: Time vs.
Number of Web Services

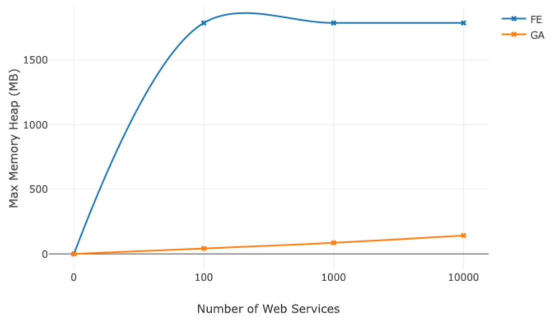


Figure 45: Memory Heap vs.
Number of Web Services

Number of Web Services	**Time - FE(ms)**	**Avg Time - GA(ms)**	**Solution(s) - FE**	**Solution(s) - GA - Min**	**Solution(s) - GA - Max**	**Max Memory Heap(MB) - FE**	**Max Memory Heap(MB) - GA**	**Service Composition - FE**	**Service Composition - GA**
100	-	722	-	1	4	1787	45	Failed	Successful
1000	-	743	-	1	9	1787	76	Failed	Successful
10000	-	1615	-	1	3	1787	132	Failed	Successful

Table 33: Evaluation Summary for Test 14

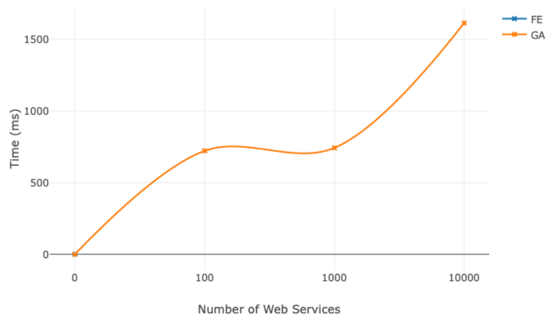


Figure 46: Time vs.
Number of Web Services

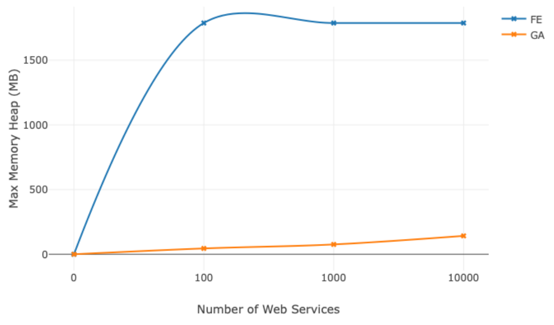


Figure 47: Memory Heap vs.
Number of Web Services

Min-Input	**Max - Input**	**Min-Output**	**Max-Output**	**Number of Variables Used**	**Percentage of Variables Having Constraints**
1	10	1	10	20	50%

Table 34: Statistical Factors for Test 15

Number of Web Services	**Time - FE(ms)**	**Avg Time - GA(ms)**	**Solution(s) - FE**	**Solution(s) - GA - Min**	**Solution(s) - GA - Max**	**Max Memory Heap(MB) - FE**	**Max Memory Heap(MB) - GA**	**Service Composition - FE**	**Service Composition - GA**
100	-	727	-	1	12	1787	42	Failed	Successful
1000	-	834	-	1	6	1787	84	Failed	Successful
10000	-	1676	-	1	8	1787	146	Failed	Successful

Table 35: Evaluation Summary for Test 15

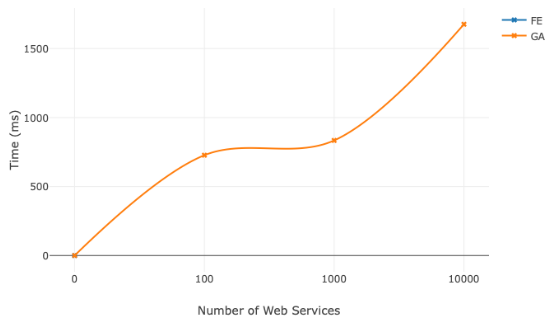


Figure 48: Time vs. Number of Web Services

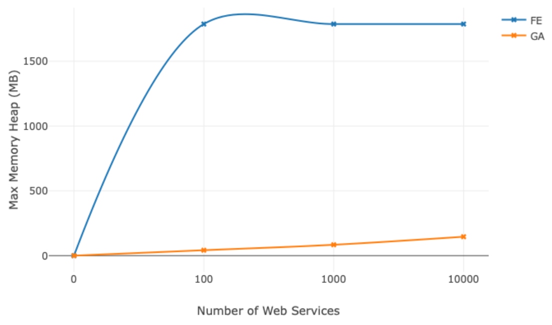


Figure 49: Memory Heap vs. Number of Web Services

Min-Input	**Max - Input**	**Min-Output**	**Max-Output**	**Number of Variables Used**	**Percentage of Variables Having Constraints**
1	10	1	10	20	100%

Table 36: Statistical Factors for Test 16

Number of Web Services	**Time - FE(ms)**	**Avg Time - GA(ms)**	**Solution(s) - FE**	**Solution(s) - GA - Min**	**Solution(s) - GA - Max**	**Max Memory Heap(MB) - FE**	**Max Memory Heap(MB) - GA**	**Service Composition - FE**	**Service Composition - GA**
100	-	712	-	1	6	1787	52	Failed	Successful
1000	-	855	-	1	10	1787	99	Failed	Successful
10000	-	1596	-	1	24	1787	224	Failed	Successful

Table 37: Evaluation Summary for Test 16

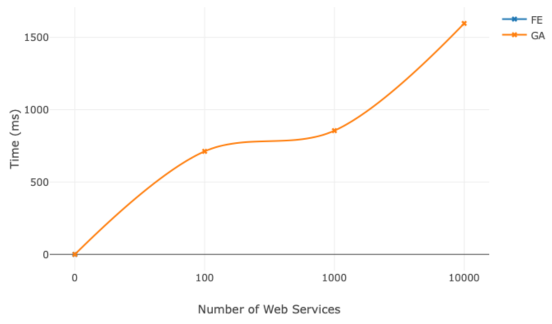


Figure 50: Time vs. Number of Web Services

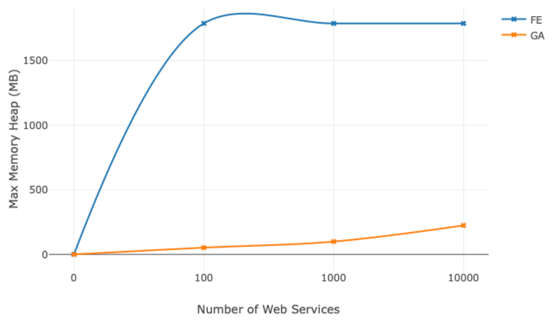


Figure 51: Memory Heap vs. Number of Web Services

Min-Input	**Max - Input**	**Min-Output**	**Max-Output**	**Number of Variables Used**	**Percentage of Variables Having Constraints**
1	10	1	10	50	50%

Table 38: Statistical Factors for Test 17

Number of Web Services	Time - FE(ms)	Avg Time - GA(ms)	Solution(s) - FE	Solution(s) - GA - Min	Solution(s) - GA - Max	Max Memory Heap(MB) - FE	Max Memory Heap(MB) - GA	Service Composition - FE	Service Composition - GA
100	-	746	-	1	6	1787	41	Failed	Successful
1000	-	861	-	1	9	1787	86	Failed	Successful
10000	-	1745	-	1	15	1787	252	Failed	Successful

Table 39: Evaluation Summary for Test 17

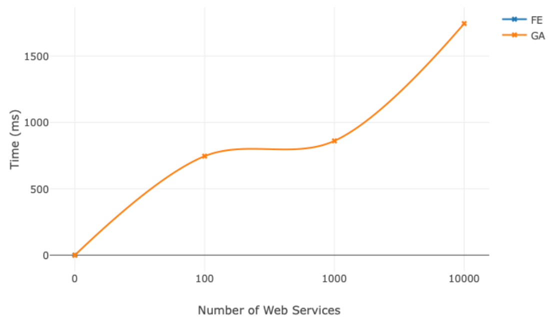


Figure 52: Time vs. Number of Web Services

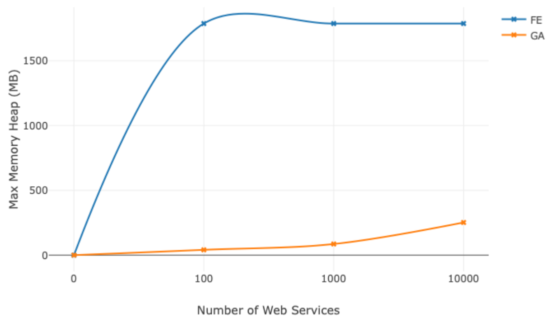


Figure 53: Memory Heap vs. Number of Web Services

Min-Input	Max - Input	Min-Output	Max-Output	Number of Variables Used	Percentage of Variables Having Constraints
1	10	1	10	50	100%

Table 40: Statistical Factors for Test 18

Number of Web Services	Time - FE(ms)	Avg Time - GA(ms)	Solution(s) - FE	Solution(s) - GA - Min	Solution(s) - GA - Max	Max Memory Heap(MB) - FE	Max Memory Heap(MB) - GA	Service Composition - FE	Service Composition - GA
100	-	812	-	1	8	1787	46	Failed	Successful
1000	-	910	-	1	19	1787	99	Failed	Successful
10000	-	1832	-	1	14	1787	310	Failed	Successful

Table 41: Evaluation Summary for Test 18

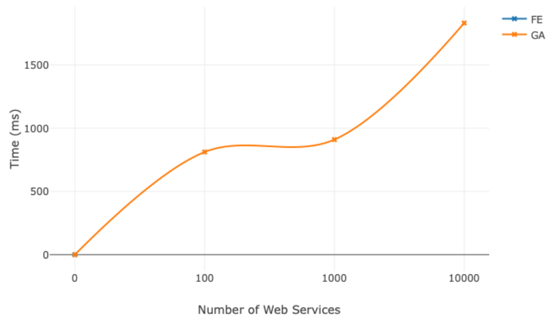


Figure 54: Time vs. Number of Web Services

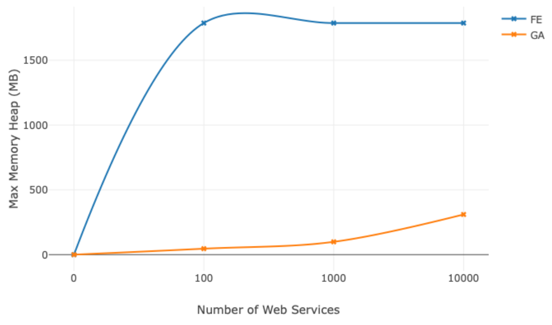


Figure 55: Memory Heap vs. Number of Web Services

The GA-based method works the same way when we scale the solution in the number of variables used or the percentage of variables having constraints. That the amount of time it takes increases slightly with the number of web services in the repository. The forward expansion algorithm, in contrast, demands more time and space as all factors increase. The forward expansion technique is deterministic and brute-force, but the GA-based method is nondeterministic and heuristic. As a result, the GA-based method performed substantially better in terms of computation time and space when scaling in solution space than the forward expansion methodology. For example, the Figure 20 shows time consumption for both methods when we gradually increased the number of web services from 10 to 10000 with min/max input/output set to 1, the number of variables used set to 10, and the percentage of having constraints set to 50%. Figure 21 illustrates max memory heap consumption for both methods when we gradually increased the number of web services from 10 to 10,000 with min/max input/output set to 1, the number of variables used set to 10, and the percentage of having constraints set to 50%. These results indicate that the GA-based method can be utilized for real-time or dynamic large-scale service composition. However, the forward expansion methodology can only be used when the solution size is very small.

4.2.2 Effectiveness Evaluation

The GA approach was compared with the forward expansion approach, which only marked solution genome instances as candidates instead of generating all possible plans. To this end, let us examine the generated plans that each approach obtained. As the number of web services and the variables used becomes larger, i.e., more fan-out instances are available in the generated graph. Therefore, the forward expansion algorithm searches using a brute-force approach to generate all possible solutions based on nodes with massive fan-out, and the graph grows wider. The corresponding repository factors are shown in Table 42. The result data for the forward expansion algorithm are shown in Table 44. As var_2 was requested as output in Listing 4.1 and var_1 was provided as output, as a result, as depicted in Table 44, the computation

Listing 4.1: Configuration Request

```
        <?xml version="1.0" encoding="UTF−8" standalone="no"?>
        <requestconfig>
        <inputs value="string : var1"/>
        <outputs value="string : var2"/>
        <qos value="COST"/>
        <constraints value="COST | &lt; | 100"/>
        <storecsflag value="N"/>
        </requestconfig>
```

time and space became so large. This exponential growth caused the application to be faced with an out-of-memory error and could not find any solution.

Number of Web Services	**Min-Input**	**Max - Input**	**Min-Output**	**Max-Output**	**Number of Variables Used**	**Percentage of Variables Having Constraints**
1000	1	5	1	5	10	50%

Table 42: Repository Information

Method	**Service Composition**	**Min - Solutions**	**Max - Solutions**	**Avg Time (ms)**	**Number of Nodes**
Forward Expansion	Failed	-	-	-	-
Genetic Algorithm	Successful	1	10	655	20

Table 43: FE and GA Results

Time(ms)	**CPU**	**Memory Heap(MB)**	**Threads**	**Non Heap Memory(MB)**
1000	40%	284	10	17
2000	49%	433	11	18
3000	86%	1486	11	18
4000	81%	1787	11	18

Table 44: FE Information

Time(ms)	**CPU**	**Memory Heap(MB)**	**Threads**	**Non Heap Memory(MB)**
500	20%	57	10	17

Table 45: GA Memory Usage Information

In contrast, we experimented on the same repository, and our proposed GA algorithm found a candidate solution. Table 46 demonstrates the fittest genome, which is the solution candidate generated by GA. Table 47 shows the statical information of each generation through building graph by GA. As var_2 was requested as output in Listing 4.1 and var_1 was provided as output, the solution candidate converted to the layered based constructed plan as depicted in Listing 4.2.

#	Service	Input Parameters	Output Parameters
1	W31	var1	var47
2	W882	var47	var2
3	W146	var1	var40
4	W653	var42	var5
5	W332	var20	var16
6	W634	var14	var13
7	W354	var35	var31
8	W15	var47	var4
9	W571	var29	var46
10	W778	var39	var21
11	W694	var36	var30
12	W798	var44	var36
13	W813	var27	var32
14	W952	var31	var37
15	W802	var7	var45
16	W554	var11	var1
17	W14	var9	var16
18	W418	var50	var38
19	W683	var4	var45
20	W903	var3	var16

Table 46: Fittest Genome as Solution candidate generated by GA

Listing 4.2: Constructed Plan (in Layer-Based)

```
        Plan 1:
        Layer 0: {} [] W31 {W882}
        Layer 1: {W31} [] W882 {}
```

Table 44 shows that the forward expansion algorithm's memory usage rises exponentially, which is why the application could not generate any solution. The forward expansion algorithm is deterministic and brute-force, whereas the GA-based method is nondeterministic and heuristic. As a result, the GA-based method outperformed the forward expansion algorithm in terms of computation time and space. Table 43 provided the comparative results for both methods. Next, let us examine the question of how many feasible solutions from composite web services

Generations	Fitness Score
10	960
20	1920
30	2160
40	6180
50	8200
60	9160
70	9880
80	9880
90	9880
100	9880

Table 47: Fittest Genome Score versus Generation

generated by a genetic algorithm can be found when the forward expansion can generate total solutions on the same request configuration and repository.

Number of Web Services	Min-Input	Max - Input	Min-Output	Max-Output	Number of Variables Used	Percentage of Variables Having Constraints
1000	1	1	1	1	10	50%

Table 48: Repository Information

Method	Service Composition	Min - Solutions	Max - Solutions	Avg Time (ms)	Number of Nodes
Forward Expansion	Successful	52	52	1573	194
Genetic Algorithm	Successful	9	19	432	14

Table 49: FE vs. GA

Time(ms)	CPU	Memory Heap(MB)	Threads	Non Heap Memory(MB)
400	25%	20	10	12
800	45%	413	10	17
1200	40%	397	10	17

Table 50: Forward Expansion Memory Usage

Time(ms)	CPU	Memory Heap(MB)	Threads	Non Heap Memory(MB)
400	41%	28	10	17

Table 51: Genetic Algorithm Memory Usage

Number of Web Services	Min-Input	Max - Input	Min-Output	Max-Output	Number of Variables Used	Percentage of Variables Having Constraints
1000000	1	1000	1	1000	50%	1000

Table 52: Repository Information

Method	**Service Composition**	**Min - Solutions**	**Max - Solutions**	**Avg Time (ms)**	**Number of Nodes**
Forward Expension	Failed	-	-	-	-
Genetic Algorithm	Partial-Solution	-	-	-	-

Table 53: Number of Web services time diffrences

To this end, we experimented on the repository with Table 48 statistical information and based on the Listing 4.1 request configuration. Table 49 provided the comparative results for both methods. Next, let us examine if the factors increased significantly and how each algorithm behaves in our tests. As Table 52 and Table 53 presented, the forward expansion failed to generate any plan, and GA generated a partial solution, which means that the final generation might have part of the solution but can not find a solution.

4.3 Summary

As mentioned in Section 4.2, the activities to be completed to achieve the objectives of this thesis (as specified in Section 1.2) are structured as a systematic process (described in Section 1.5) targeted at generating context- and constraint-aware composite services on a large-scale repository. The steps of this approach, which pertain to the development of composite services on a large-scale repository, have been successfully completed as part of this study. Furthermore, components of our overall verification system responsible for these duties have been planned and implemented, explained in Chapter 3 and thoroughly reviewed in Section 4.2, and determined to be capable of achieving all of their design goals (See `https://github.com/GIPSY-dev/ServiceCompositionRepo` for the implementation of the solution and the results of the tests). In this thesis, we propose a GA approach for generating context- and constraint-aware composite services on a large-scale repository, based on all the ideas presented thus far and our time, scope, and resource constraints. We accomplished the objectives of this thesis to the best of our abilities at the time by describing the development of all the necessary components that comprise it and demonstrating through a careful evaluation technique that they meet all of their requirements.

Chapter 5

Conclusion and Future Work

In this final chapter, we give an overview of all the discussions made so far in this thesis. Then, we describe the limitations we found in our suggested solution during this research and how we intend to address each of them in future work.

5.1 Conclusion

Composite web services have been extensively researched over the last two decades due to benefits such as clarity of structure, re-usability of components, broader options for users, and the freedom to specialize for providers. Nonetheless, based on a thorough review of the literature on the studies conducted in the field thus far (as presented in Section 2.4), we conclude that no web service has the universal aptitude and has mainly been overlooked in all existing research. Most of these studies need to consider that each service can only do its job well within certain limits. These limits are set by the service providers themselves and are called internal constraints. Internal constraints impact the functionality of a composite service because, when used as a component, the restricted context spaces of all such component services define the contextual boundaries of the composite service cohesively. However, to our knowledge, only Gupta's systems [3] cater to the specific verification and validation of internal constraints imposed on components of a composite service. This is due to the limited exposure this aspect of web service composition has received (discussed elaborately in

Section 1.1 and Section 2.4). In addition, we investigated the scalability of the solution space in this specific research, and we found some limitations when the complexity of the solution space increased. As the parameters of the current version of the service composition algorithm are increased, the experiment uses up an exponential amount of space. This issue causes failure to generate an execution plan before any complete solution can be found. This is because this algorithm generates the graph using a brute force method. The brute force approach ensures that all solutions are found by listing all possible candidate solutions to the user's request. This method for solving a problem relies more on compromising the computing power of a computer system than on a well-designed algorithm. So on a large-scale repository with the high complexity of the solution space factors, based on our experiments in Chapter 4, this method is inefficient. In an attempt to address these gaps and problems related to web service composition, we defined **three goals** in Section 1.2 for this thesis.

Our **first goal** was ***"To formally define and apply the new operational service composition mechanism to the problem of constraint-aware service composition".*** we propose a genetic algorithm for generating constraint- and context-aware composite web services in this thesis. The service composition methodology used in this thesis has been characterized as a collection of algorithms, which has a particular objective that must be met by carrying out a specific set of steps. These tests have been designed based on thoroughly examining the concepts and models for generation of constraint- and context-aware composite web services. In Section 4.2.2, we showed that our GA approach could successfully generate the constraint- and context-aware composite web service, and we illustrated its plan graph.

Then, our **second goal** was ***" To scale the solution scope for the execution context of services and the restrictions/constraints imposed on them so that, if possible, one solution to any valid composition request is generated".*** In Section 4.2.1, we experimented the computation time, space, the number of solutions, and success of service composition using the GA-based method (the results were an average of three runs for each test) and forward expansion to compare the result of both approaches to meet our second goal. When we scale the solution in terms of the number web services in the repository, the number of variables used, or the percentage of variables with constraints, the GA-based method works the same way. The time required increases slightly with the number of web services in the repository. In contrast, the forward expansion algorithm requires more time and space as all factors increase. Forward expansion is deterministic and brute-force, whereas the GA-based method is nondeterministic and heuristic. As a result, when scaling in solution space, the GA-based method outperformed the forward expansion methodology regarding computation time and space. These findings suggest that the GA-based method is suitable for real-time or dynamic large-scale service composition. On the other hand, the forward expansion methodology can only be used when the solution size is very small. The experiments showed that our proposed methodology had successful results during the service composition phase on the large-scale solution space and can overcome the forward expansion problem.

Our **final goal** of this thesis was ***"To allow our verification system to simulate and execute context- and constraint-aware composite web services on a large scale".*** In Section 4.2.1, we tested Gupta's verification system to simulate and execute context- and constraint-aware composite web services with our proposed methodology on the large-scale solution space. Taking our scope and time limitations into account, we can conclude that we have been able to effectively evaluate our composition solutions to the best of our abilities due to the meticulous study that we have conducted on the composition methodology, and the successful execution of all the tests that we have performed (refer to Chapter 4 for complete details of the assessment conducted).

Based on these discussions, we conclude that, given our time, scope, and resource limitation, we have proposed a scalable automatic service composition using a genetic algorithm, effectively described the development of all its constituent building blocks, and successfully demonstrated that they meet all their requirements, thereby achieving the objectives of this thesis (as defined in Section 1.2) to the best of our abilities.

5.2 Limitations and Future Work

We discovered several features that can be incorporated into our current solution to make it more comprehensive, maintainable, efficient, robust, reliable, and versatile but have not been included at this time due to time and scope restrictions. This section describes all of the features, enhancements, and incomplete tasks that can or will be integrated into the various units of our proposed solution in future extensions to this research. The following are the limitations discovered in and future work to be undertaken for the genetic algorithm stage of the service composition process:

- Advanced optimization techniques, such as NEAT algorithm [89] which can optimize and evolve neural network structure. With help of Neat Algorithm, a neural network evolves through a genetic algorithm rather than relying on a fixed structure. It employs direct encoding, and its representation is slightly more complex than a simple graph or binary encoding. It can be integrated into the search graph construction stage to improve the quality of composition solutions extracted later in the process. This can solve one of the complex problems of dynamic genetic algorithms, which is using a different genome length for genetic operators such as crossover and mutation.

- Future work will look into how to optimize the fitness function to generate more composite services. Furthermore, it will be beneficial to investigate how to extend other meta-heuristic algorithms to solve the proposed problem and compare their results with the proposed algorithm.

- Although GA can generate at least one solution, in this approach, we cannot determine that we generated all possible solutions inside the repository. Therefore, generating all solutions is out of the scope of this thesis.

- Another research direction is creating a knowledge-based operator [90] to improve search efficiency further. The knowledge-based genetic algorithm incorporates domain knowledge into its specialized operators, and some of them also use a local search technique.

- Federated learning [91] is one of the possible domains that can be integrated into this research topic. Federated learning (also known as collaborative learning) is a machine learning technique that trains an algorithm without exchanging data samples across multiple decentralized edge devices or servers holding local data samples. This approach differs from traditional centralized machine learning techniques, in which all local datasets are uploaded to a single server, and more traditional decentralized approaches, which frequently assume that local data samples are uniformly distributed. So we can consider federated learning to generate service composition in future studies.

Bibliography

[1] Z. Wu, S. Deng, and J. Wu, *Service Computing, Concepts, Methods and Technology*. Hangzhou, China: Morgan Kaufmann, Oct. 2014, ISBN: 978-0-12-802330-3.

[2] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. Addison-Wesley, 2012, ISBN: 978-0-13-214301-1.

[3] J. Gupta, "Execution/simulation of context/constraint-aware composite services using gipsy," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada.

[4] R. Aggarwal, K. Verma, J. Miller, and W. Milnor, "Constraint driven web service composition in METEOR-S," in *IEEE International Conference on Services Computing, 2004. (SCC 2004). Proceedings. 2004*, Sept 2004. doi: 10.1109/SCC.2004.1357986 pp. 23–30.

[5] G. Chafle, K. Dasgupta, A. Kumar, S. Mittal, and B. Srivastava, "Adaptation in web service composition and execution," in *2006 IEEE International Conference on Web Services (ICWS'06)*, Sept 2006. doi: 10.1109/ICWS.2006.22 pp. 549–557.

[6] S. Youcef, M. U. Bhatti, L. Mokdad, and V. Monfort, "Simulation-based response-time analysis of composite web services," in *2006 IEEE International Multitopic Conference*, Dec 2006. doi: 10.1109/INMIC.2006.358190 pp. 349–354.

[7] C. Zhu and Y. Du, "Application of logical petri nets in web service composition," in *2010 IEEE International Conference on Mechatronics and Automation*, Aug 2010. doi: 10.1109/ICMA.2010.5589966. ISSN 2152-744X pp. 913–918.

[8] M. Chen, T. H. Tan, J. Sun, Y. Liu, and J. S. Dong, "VeriWS: A tool for verification of combined functional and non-functional requirements of web service composition," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014. doi: 10.1145/2591062.2591070. ISBN 978-1-4503-2768-8 pp. 564–567. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591070

[9] K. T. Huynh, T. T. Quan, and T. H. Bui, "Fast and formalized: Heuristics-based on-the-fly web service composition and verification," in *2015 2nd National Foundation for Science and Technology Development Conference on Information and Computer Science (NICS)*, Sept 2015. doi: 10.1109/NICS.2015.7302186 pp. 174–179.

[10] V. Shkarupylo, "A simulation-driven approach for composite web services validation," in *Central European Conference on Information and Intelligent Systems*. Faculty of Organization and Informatics Varazdin, Sept 2016, p. 227.

[11] S. Narayanan and S. A. McIlraith, "Simulation, verification and automated composition of web services," in *Proceedings of the 11th International Conference on World Wide Web*, ser. WWW '02. New York, NY, USA: ACM, 2002. doi: 10.1145/511446.511457. ISBN 1-58113-449-5 pp. 77–88. [Online]. Available: http://doi.acm.org/10.1145/511446.511457

[12] S. Narayanan and S. McIlraith, "Analysis and simulation of web services," *Computer Networks*, vol. 42, no. 5, pp. 675 – 693, 2003. doi: https://doi.org/10.1016/S1389-1286(03)00228-7 The Semantic Web: an evolution for a revolution. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128603002287

[13] X. Wang and S. Yu, "A novel method for verification of composite web services," in *2015 2nd International Conference on Information Science and Control Engineering*, April 2015. doi: 10.1109/ICISCE.2015.17 pp. 37–40.

[14] C. Dechsupa, W. Vatanawood, and A. Thongtak, "Formal verification of web service orchestration using colored petri net," in *Proceedings of the International MultiConference of Engineers and Computer Scientists*, vol. 1, March 2016.

[15] X. Fu, T. Bultan, and J. Su, "Analysis of interacting BPEL web services," in *Proceedings of the 13th international conference on World Wide Web*. ACM, 2004, pp. 621–630.

[16] J. Paquet and P. G. Kropf, "The GIPSY architecture," in *Proceedings of Distributed Computing on the Web*, ser. Lecture Notes in Computer Science, P. G. Kropf, G. Babin, J. Plaice, and H. Unger, Eds., vol. 1830. Springer Berlin Heidelberg, 2000. doi: 10.1007/3-540-45111-0_17 pp. 144–153.

[17] J. Paquet and A. H. Wu, "GIPSY – a platform for the investigation on intensional programming languages," in *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*. CSREA Press, Jun. 2005. ISBN 1-932415-75-0 pp. 8–14.

[18] J. Paquet, "Distributed eductive execution of hybrid intensional programs," in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*. IEEE Computer Society, Jul. 2009. doi: 10.1109/COMPSAC.2009.137. ISBN 978-0-7695-3726-9. ISSN 0730-3157 pp. 218–224.

[19] T. Laleh, "Constraint verification in web service composition," Ph.D. dissertation, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Feb. 2018.

[20] T. Laleh, J. Paquet, S. Mokhov, and Y. Yan, "Constraint verification failure recovery in web service composition," *Future Generation Computer Systems*,

vol. 89, pp. 387 – 401, 2018. doi: https://doi.org/10.1016/j.future.2018.06.037 http://www.sciencedirect.com/science/article/pii/S0167739X17320629.

[21] ——, "Predictive failure recovery in constraint-aware web service composition," in *Proceedings of the 7th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER*, INSTICC. SciTePress, 2017. doi: 10.5220/0006313802410252. ISBN 978-989-758-243-1 pp. 241–252.

[22] ——, "Constraint adaptation in web service composition," in *2017 IEEE International Conference on Services Computing (SCC)*, June 2017. doi: 10.1109/SCC.2017.27. ISSN 2474-2473 pp. 156–163.

[23] T. Laleh, J. Paquet, S. A. Mokhov, and Y. Yan, "Efficient constraint verification in service composition design and execution (short paper)," in *CoopIS*. Springer, 2016, pp. 445–455.

[24] L. Zhang, H. Dou, H. Wang, Y. Peng, S. Zheng, and C. Zhang, "Neural network optimized by genetic algorithm for predicting single well production in high water cut reservoir," in *2021 3rd International Conference on Intelligent Control, Measurement and Signal Processing and Intelligent Oil Field (ICMSP)*, 2021. doi: 10.1109/ICMSP53480.2021.9513395 pp. 297–306.

[25] Z. yi Chai, M. meng Du, and G. zhi Song, "A fast energy-centered and qos-aware service composition approach for internet of things," *Applied Soft Computing*, vol. 100, p. 106914, 2021. doi: https://doi.org/10.1016/j.asoc.2020.106914. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1568494620308528

[26] H. T. Khosrowshahi and M. Shakeri, "Relay node placement for connectivity restoration in wireless sensor networks using genetic algorithms," *International Journal of Electronics and Communication Engineering*, vol. 12, no. 3, pp. 161–170, 2018.

[27] Amazon, "Aws compute optimizer documentation," [online], 2022, https://docs.aws.amazon.com/compute-optimizer/index.html.

[28] kubecost, "Kubecost," [online], 2022, https://guide.kubecost.com/hc/en-us/articles/4407595950359.

[29] datadog, "datadog," [online], 2022, https://www.datadoghq.com/dg/ccm/aws-cloud-cost-management/?utm_source=advertisement&utm_medium=search&utm_campaign=dg-google-cloudcostmgmt-na-aws&utm_keyword=aws%20cost%20management%20tools&utm_matchtype=p&utm_campaignid=18599012112&utm_adgroupid=141830245825&gclid=CjwKCAiAvK2bBhB8EiwAZUbP1OV_kHdOex7kbD_it6_MzRJ4vkJLW8nWIqK1vAIFiD-8LFI4rBbuFxoCo4kQAvD_BwE.

[30] T. I. Authors, "Istio," [online], 2023. [Online]. Available: https://istio.io/latest/docs/ops/deployment/performance-and-scalability/

[31] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge, *Multidimensional Programming.* London: Oxford University Press, Feb. 1995, ISBN: 978-0195075977.

[32] W. W. Wadge and E. A. Ashcroft, *Lucid, the Dataflow Programming Language*. London: Academic Press, 1985.

[33] P. Rondogiannis and W. W. Wadge, "Intensional programming languages," in *Proceedings of the First Panhellenic Conference on New Information Technologies (NIT'98), Athens, Greece*, 1998, pp. 85–94.

[34] B. Han, "Towards a multi-tier runtime system for GIPSY," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2010.

[35] J. Paquet, "Scientific intensional programming," Ph.D. dissertation, Department of Computer Science, Quebec City, Canada, 1999.

[36] J. Cheng, C. Liu, M. Zhou, Q. Zeng, and A. Yla-Jaaski, "Automatic composition of semantic web services based on fuzzy predicate petri nets," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 2, pp. 680–689, April 2015. doi: 10.1109/TASE.2013.2293879

[37] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber, "The petri net markup language: Concepts, technology, and tools," in *Applications and Theory of Petri Nets 2003*, W. M. P. van der Aalst and E. Best, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. ISBN 978-3-540-44919-5 pp. 483–505.

[38] "Pnml.org - PNML reference site," http://www.pnml.org/, viewed in August 2018.

[39] S. Juan and W. Hao, "Performance analysis for web service composition based on queueing petri net," in *Software Engineering and Service Science (ICSESS), 2012 IEEE 3rd International Conference on.* IEEE, 2012, pp. 501–504.

[40] R. Jagannathan and C. Dodd, "GLU programmer's guide," SRI International, Menlo Park, California, Tech. Rep., 1996.

[41] R. Jagannathan, C. Dodd, and I. Agi, "GLU: A high-level system for granular data-parallel programming," in *Concurrency: Practice and Experience*, vol. 1, 1997, pp. 63–83.

[42] S. A. Mokhov, "Towards hybrid intensional programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Oct. 2005, ISBN 0494102934; online at http://arxiv.org/abs/0907.2640.

[43] S. Mokhov and J. Paquet, "Objective Lucid – first step in object-oriented intensional programming in the GIPSY," in *Proceedings of the 2005 International*

Conference on Programming Languages and Compilers (PLC 2005). CSREA Press, Jun. 2005. ISBN 1-932415-75-0 pp. 22–28.

[44] Sun Microsystems, Inc., "The Java web services tutorial (for Java Web Services Developer's Pack, v2.0)," [online], Feb. 2006, http://download.oracle.com/docs/cd/E17802_01/webservices/webservices/docs/2.0/tutorial/doc/.

[45] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN 0201157675

[46] M. Hussain, L.-F. Wei, F. Abbas, A. Rehman, M. Ali, and A. Lakhan, "A multi-objective quantum-inspired genetic algorithm for workflow healthcare application scheduling with hard and soft deadline constraints in hybrid clouds," *Applied Soft Computing*, vol. 128, p. 109440, 2022. doi: https://doi.org/10.1016/j.asoc.2022.109440. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1568494622005610

[47] D. Min, Z. Song, H. Chen, T. Wang, and T. Zhang, "Genetic algorithm optimized neural network based fuel cell hybrid electric vehicle energy management strategy under start-stop condition," *Applied Energy*, vol. 306, p. 118036, 2022.

[48] A. A. Khan, A. A. Shaikh, Z. A. Shaikh, A. A. Laghari, and S. Karim, "Ipm-model: Ai and metaheuristic-enabled face recognition using image partial matching for multimedia forensics investigation with genetic algorithm," *Multimedia Tools and Applications*, pp. 1–17, 2022.

[49] y. Eiben, A. E. and Smith, J. E.", title=ntroduction to Evolutionary Computing. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-662-05094-1. [Online]. Available: https://doi.org/10.1007/978-3-662-05094-1_1

[50] L. Davis, *Handbook of Genetic Algorithms*, ser. VNR Computer Library VNR Computer Library. Van Nostrand Reinhold, 1991. ISBN 9780442001735. [Online]. Available: https://books.google.ca/books?id=Kl7vAAAAMAAJ

[51] C. Z. Janikow and Z. Michalewicz, "An experimental comparison of binary and floating point representations in genetic algorithms," in *ICGA*, 1991.

[52] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs (2nd, Extended Ed.).* Berlin, Heidelberg: Springer-Verlag, 1994. ISBN 3540580905

[53] J. R. Koza, "Evolution and co-evolution of computer programs to control independently-acting agents," in *n Proceedings of the first international conference on simulation of adaptive behavior on From animals to animats*, 1990.

[54] T. Blickle and L. Thiele, "A mathematical analysis of tournament selection," in *Proceedings of the 6th International Conference on Genetic Algorithms.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995. ISBN 1558603700 p. 9–16.

[55] W. Banzhaf and C. Reeves, *Foundations of Genetic Algorithms*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN 1558605592

[56] P. Wang, Z. Ding, C. Jiang, and M. Zhou, "Constraint-aware approach to web service composition," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, no. 6, pp. 770–784, June 2014. doi: 10.1109/TSMC.2013.2280559

[57] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani, "An approach for qos-aware service composition based on genetic algorithms," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '05. New York, NY, USA: Association for Computing Machinery, 2005. doi: 10.1145/1068009.1068189. ISBN 1595930108 p. 1069–1075. [Online]. Available: https://doi.org/10.1145/1068009.1068189

[58] Y. Ma and C. Zhang, "Quick convergence of genetic algorithm for qos-driven web service selection," *Comput. Netw.*, vol. 52, no. 5, p.

1093–1104, apr 2008. doi: 10.1016/j.comnet.2007.12.003. [Online]. Available: https://doi.org/10.1016/j.comnet.2007.12.003

[59] Q. Wu, Q. Zhu, and X. Jian, "QoS-Aware Multi-granularity Service Composition Based on Generalized Component Services, booktitle=Service-Oriented Computing, year=2013." Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-642-45005-1 pp. 446–455.

[60] W. Liu, B. Liu, D. Sun, Y. Li, and G. Ma, "Study on multi-task oriented services composition and optimisation with the 'multi-composition for each task' pattern in cloud manufacturing systems," *International Journal of Computer Integrated Manufacturing*, vol. 26, no. 8, pp. 786–805, 2013. doi: 10.1080/0951192X.2013.766939

[61] Q. Wu, Q. Zhu, X. Jian, and F. Ishikawa, "Broker-based sla-aware composite service provisioning," *Journal of Systems and Software*, vol. 96, pp. 194–201, 2014. doi: https://doi.org/10.1016/j.jss.2014.06.027. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121214001459

[62] T. Yu and K.-J. Lin, "Service selection algorithms for composing complex services with multiple qos constraints," in *Service-Oriented Computing - ICSOC 2005*, B. Benatallah, F. Casati, and P. Traverso, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005. ISBN 978-3-540-32294-8 pp. 130–143.

[63] D. Schuller, A. Polyvyanyy, L. García-Bañuelos, and S. Schulte, "Optimization of complex qos-aware service compositions," in *ICSOC*, 2011.

[64] D. Schuller, U. Lampe, J. Eckert, R. Steinmetz, and S. Schulte, "Cost-driven optimization of complex service-based workflows for stochastic qos parameters," 06 2012. doi: 10.1109/ICWS.2012.50 pp. 66–73.

[65] M. Alrifai, D. Skoutas, and T. Risse, "Selecting skyline services for qos-based web service composition," in *Proceedings of the 19th International Conference on World Wide Web*. New York, NY, USA: Association for Computing Machinery,

2010. doi: 10.1145/1772690.1772693. ISBN 9781605587998 p. 11–20. [Online]. Available: https://doi.org/10.1145/1772690.1772693

[66] M. Alrifai, T. Risse, and W. Nejdl, "A hybrid approach for efficient web service composition with end-to-end qos constraints," *ACM Trans. Web*, vol. 6, no. 2, jun 2012. doi: 10.1145/2180861.2180864. [Online]. Available: https://doi.org/10.1145/2180861.2180864

[67] S. X. Sun and J. Zhao, "A decomposition-based approach for service composition with global qos guarantees," *Information Sciences*, vol. 199, pp. 138–153, 2012. doi: https://doi.org/10.1016/j.ins.2012.02.061. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020025512001892

[68] A. Klein, F. Ishikawa, and S. Honiden, "Sanga: A self-adaptive network-aware approach to service composition," *IEEE Transactions on Services Computing*, vol. 7, no. 3, pp. 452–464, 2014. doi: 10.1109/TSC.2013.2

[69] H. Wada, J. Suzuki, Y. Yamano, and K. Oba, "A multiobjective optimization framework for sla-aware service composition," *IEEE Trans. Service Comput.*, vol. 99, pp. 1–14, 01 2012.

[70] H. Ma, F. Bastani, I.-L. Yen, and H. Mei, "Qos-driven service composition with reconfigurable services," *IEEE Transactions on Services Computing*, vol. 6, no. 1, pp. 20–34, 2013. doi: 10.1109/TSC.2011.21

[71] P. Leitner, W. Hummer, and S. Dustdar, "Cost-based optimization of service compositions," *IEEE Transactions on Services Computing*, vol. 6, no. 2, pp. 239–251, 2013. doi: 10.1109/TSC.2011.53

[72] P. Wang, Z. Ding, C. Jiang, and M. Zhou, "Constraint-aware approach to web service composition," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, no. 6, pp. 770–784, 2014. doi: 10.1109/TSMC.2013.2280559

[73] F. Lécué and N. Mehandjiev, "Seeking quality of web service composition in a semantic dimension," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 6, pp. 942–959, 2011. doi: 10.1109/TKDE.2010.237

[74] S. A. McIlraith and T. C. Son, "Adapting golog for composition of semantic web services," in *Proceedings of the Eights International Conference on Principles of Knowledge Representation and Reasoning.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002. ISBN 1558605541 p. 482–496.

[75] Y. Yan, B. Xu, Z. Gu, and S. Luo, "A qos-driven approach for semantic service composition," *2009 IEEE Conference on Commerce and Enterprise Computing*, pp. 523–526, 2009.

[76] T. W. Wei Jiang, "Qos-aware automatic service composition: A graph view," *Journal of Computer Science and Technology*, vol. 26, no. 5, p. 837, 2011. doi: 10.1007/s11390-011-0183-2

[77] P. Bartalos and M. Bielikova, "Qos aware semantic web service composition approach considering pre/postconditions," 07 2010. doi: 10.1109/ICWS.2010.90 pp. 345–352.

[78] F. Wagner, F. Ishikawa, and S. Honiden, "Qos-aware automatic service composition by applying functional clustering," 07 2011. doi: 10.1109/ICWS.2011.32 pp. 89–96.

[79] A. E. Yilmaz and P. Karagoz, "Improved genetic algorithm based approach for qos aware web service composition," in *2014 IEEE International Conference on Web Services*, 2014. doi: 10.1109/ICWS.2014.72 pp. 463–470.

[80] Q. Wu, F. Ishikawa, Q. Zhu, and D.-H. Shin, "Qos-aware multigranularity service composition: Modeling and optimization," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 46, no. 11, pp. 1565–1577, 2016. doi: 10.1109/TSMC.2015.2503384

[81] S. A. Ludwig, "Clonal selection based genetic algorithm for workflow service selection," in *2012 IEEE Congress on Evolutionary Computation*, 2012. doi: 10.1109/CEC.2012.6256465 pp. 1–7.

[82] Y. Ma and C. Zhang, "Quick convergence of genetic algorithm for qos-driven web service selection," *Computer Networks*, vol. 52, no. 5, pp. 1093–1104, 2008. doi: https://doi.org/10.1016/j.comnet.2007.12.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128607003489

[83] S. Kumar, R. Bahsoon, T. Chen, K. Li, and R. Buyya, "Multi-tenant cloud service composition using evolutionary optimization," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, 2018. doi: 10.1109/PADSW.2018.8644640 pp. 972–979.

[84] N. Kashyap, A. C. Kumari, and R. Chhikara, "Service composition in iot using genetic algorithm and particle swarm optimization," *Open Computer Science*, vol. 10, no. 1, pp. 56–64, 2020. doi: doi:10.1515/comp-2020-0011. [Online]. Available: https://doi.org/10.1515/comp-2020-0011

[85] S. S. Sefati and S. Halunga, "A hybrid service selection and composition for cloud computing using the adaptive penalty function in genetic and artificial bee colony algorithm," *Sensors*, vol. 22, 06 2022. doi: 10.3390/s22134873

[86] Q. Zhang, J. Sun, and E. Tsang, "An evolutionary algorithm with guided mutation for the maximum clique problem," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 2, pp. 192–200, 2005.

[87] K. Inc., "Xchart, simple java charts," [online], 2019. [Online]. Available: https://knowm.org/open-source/xchart/

[88] Jetbrains, "Intellij idea, profilier, cpu and memory live charts," [online], 2022. [Online]. Available: https://www.jetbrains.com/help/idea/cpu-and-memory-live-charts.html

[89] K. O. Stanley and R. Miikkulainen, “Evolving neural networks through augmenting topologies,” *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002.

[90] R. Sarkar, D. Barman, and N. Chowdhury, “Domain knowledge based genetic algorithms for mobile robot path planning having single and multiple targets,” *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 7, pp. 4269–4283, 2022. doi: https://doi.org/10.1016/j.jksuci.2020.10.010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1319157820304900

[91] “Advances and open problems in federated learning,” *Foundations and Trends® in Machine Learning*, vol. 14, no. 1–2, pp. 1–210, 2021. doi: 10.1561/2200000083. [Online]. Available: http://dx.doi.org/10.1561/2200000083