QoS-Aware Web Service Composition using Genetic Algorithms

By

Lifeng Ai

A thesis submitted to Faculty of Science and Technology
Queensland University of Technology
in partial fulfillment of requirements for the degree of
Doctor of Philosophy

June 2011
Keywords

Quality of service, web services, composite web services, optimisation, genetic algorithms
Abstract

Web service technology is increasingly being used to build various e-Applications, in domains such as e-Business and e-Science. Characteristic benefits of web service technology are its inter-operability, decoupling and just-in-time integration. Using web service technology, an e-Application can be implemented by web service composition — by composing existing individual web services in accordance with the business process of the application. This means the application is provided to customers in the form of a value-added composite web service. An important and challenging issue of web service composition, is how to meet Quality-of-Service (QoS) requirements. This includes customer focused elements such as response time, price, throughput and reliability as well as how to best provide QoS results for the composites. This in turn best fulfils customers’ expectations and achieves their satisfaction. Fulfilling these QoS requirements or addressing the QoS-aware web service composition problem is the focus of this project.

From a computational point of view, QoS-aware web service composition can be transformed into diverse optimisation problems. These problems are characterised as complex, large-scale, highly constrained and multi-objective problems. We therefore use genetic algorithms (GAs) to address QoS-based service composition problems. More precisely, this study addresses three important subproblems of QoS-aware web service composition; QoS-based web service selection for a composite web service accommodating constraints on inter-
service dependence and conflict, QoS-based resource allocation and scheduling for multiple composite services on hybrid clouds, and performance-driven composite service partitioning for decentralised execution. Based on operations research theory, we model the three problems as a constrained optimisation problem, a resource allocation and scheduling problem, and a graph partitioning problem, respectively. Then, we present novel GAs to address these problems. We also conduct experiments to evaluate the performance of the new GAs. Finally, verification experiments are performed to show the correctness of the GAs.

The major outcomes from the first problem are three novel GAs: a penalty-based GA, a min-conflict hill-climbing repairing GA, and a hybrid GA. These GAs adopt different constraint handling strategies to handle constraints on inter-service dependence and conflict. This is an important factor that has been largely ignored by existing algorithms that might lead to the generation of infeasible composite services. Experimental results demonstrate the effectiveness of our GAs for handling the QoS-based web service selection problem with constraints on inter-service dependence and conflict, as well as their better scalability than the existing integer programming-based method for large scale web service selection problems.

The major outcomes from the second problem has resulted in two GAs; a random-key GA and a cooperative coevolutionary GA (CCGA). Experiments demonstrate the good scalability of the two algorithms. In particular, the CCGA scales well as the number of composite services involved in a problem increases, while no other algorithms demonstrate this ability.

The findings from the third problem result in a novel GA for composite service partitioning for decentralised execution. Compared with existing heuristic algorithms, the new GA is more suitable for a large-scale composite web service program partitioning problems. In addition, the GA outperforms existing heuristic
algorithms, generating a better deployment topology for a composite web service for decentralised execution.

These effective and scalable GAs can be integrated into QoS-based management tools to facilitate the delivery of feasible, reliable and high quality composite web services.
Contents

Keywords i
Abstract iii
Contents vi
List of Figures xiii
List of Tables xvi
Statement of Original Authorship xvii
Acknowledgement xix
List of Publications xxi

1 Introduction 1
   1.1 Research Motivation .......................... 1
   1.2 Research Problems ............................. 7
       1.2.1 Problem One — QoS-Based Web Service Selection with Inter Service Dependencies and Conflicts 8
       1.2.2 Problem Two — QoS-Based Resource Allocation and Scheduling for Multiple Composite Web Service on Hybrid Clouds 11
       1.2.3 Problem Three — Performance-Driven Composite Web Service Partitioning for Decentralised Execution 14
   1.3 Major Contributions ............................ 15
   1.4 Thesis Outline ................................. 21

2 Background 23
   2.1 Basic Concepts of Web Service and Web Service Composition 24
       2.1.1 What is a Web Service? ...................... 24
2.1.2 What is Web Service Composition? .......................... 27
2.2 QoS with Web Services ........................................... 30
  2.2.1 QoS Properties for Web Services ......................... 31
  2.2.2 QoS-Based Management for Web Service-Based Systems 33
  2.2.3 QoS-Based Configuration Methods for Composite Services 36
  2.2.4 QoS-Based Composite Services Optimisation ............ 40
2.3 Overview of Genetic Algorithms .............................. 42
  2.3.1 What is a Genetic Algorithm? ............................ 43
  2.3.2 Components of a Genetic Algorithm .................... 44
  2.3.3 Constraint Handling Techniques for GAs ................ 48
  2.3.4 Techniques for Solving Multi-Objective Problems ....... 50
  2.3.5 Cooperative Coevolutionary Genetic Algorithms ........ 51
2.4 Summary .......................................................... 52

3 QoS-Based Web Service Selection with Inter-Service Dependencies and Conflicts 55
  3.1 Introduction ..................................................... 56
  3.2 Problem Formulation .......................................... 62
  3.3 Related Work .................................................. 64
  3.4 New Methods for Service Selection ......................... 69
    3.4.1 A Penalty-Based Genetic Algorithm .................... 70
    3.4.2 A Min-Conflict Hill-Climbing Repairing Based Genetic Approach .................. 75
    3.4.3 A Hybrid Genetic Algorithm ............................ 78
  3.5 Evaluation ..................................................... 84
    3.5.1 Experimental Setting ..................................... 85
    3.5.2 Simulation Model ......................................... 87
    3.5.3 Experimental Results ..................................... 92
  3.6 Verification ................................................... 106
  3.7 Summary ........................................................ 109

4 QoS-Based Resource Allocation and Scheduling for Multiple Composite Web Services on Hybrid Clouds 111
  4.1 Introduction ..................................................... 112
  4.2 Problem Description .......................................... 117
  4.3 Related Work .................................................. 121
  4.4 New Genetic Algorithms ..................................... 124
    4.4.1 The Random-Key Genetic Algorithm .................... 125
    4.4.2 The Cooperative Coevolutionary Genetic Algorithm .. 133
  4.5 Experiments .................................................... 144
    4.5.1 Simulation Model ......................................... 144
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5.2</td>
<td>Experiments on the Number of Composite Services</td>
<td>148</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Experiments on the Number of Abstract Services in Each Composite Service</td>
<td>152</td>
</tr>
<tr>
<td>4.5.4</td>
<td>Experiments on the Number of Candidate Cloud Services for Each Abstract Service</td>
<td>155</td>
</tr>
<tr>
<td>4.5.5</td>
<td>Experiments on the Constraint Density</td>
<td>156</td>
</tr>
<tr>
<td>4.6</td>
<td>Verification</td>
<td>160</td>
</tr>
<tr>
<td>4.7</td>
<td>Summary</td>
<td>165</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>168</td>
</tr>
<tr>
<td>5.2</td>
<td>Problem Formulation</td>
<td>173</td>
</tr>
<tr>
<td>5.3</td>
<td>Related Work</td>
<td>177</td>
</tr>
<tr>
<td>5.4</td>
<td>A GA for the BPEL Partitioning Problem</td>
<td>180</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Genetic Encoding</td>
<td>184</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Initial Population Generation</td>
<td>185</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Genetic Operators</td>
<td>187</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Fitness Function</td>
<td>187</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Precedence Constraints Handling</td>
<td>189</td>
</tr>
<tr>
<td>5.4.6</td>
<td>Control Dependency Constraints Handling</td>
<td>194</td>
</tr>
<tr>
<td>5.4.7</td>
<td>Local Optimiser</td>
<td>197</td>
</tr>
<tr>
<td>5.4.8</td>
<td>Complexity Analysis</td>
<td>199</td>
</tr>
<tr>
<td>5.5</td>
<td>Experiments</td>
<td>203</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Test Problems</td>
<td>204</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Parameters for the Cost Model and the GA</td>
<td>209</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Test Environment</td>
<td>211</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Results for Problems Without Control Dependency Constraints</td>
<td>211</td>
</tr>
<tr>
<td>5.5.5</td>
<td>Results for Problems With Control Dependency Constraints</td>
<td>215</td>
</tr>
<tr>
<td>5.6</td>
<td>Verification</td>
<td>218</td>
</tr>
<tr>
<td>5.7</td>
<td>Summary</td>
<td>221</td>
</tr>
<tr>
<td>6.1</td>
<td>Summary</td>
<td>223</td>
</tr>
<tr>
<td>6.2</td>
<td>Major Contributions</td>
<td>226</td>
</tr>
<tr>
<td>6.3</td>
<td>Future Work</td>
<td>236</td>
</tr>
</tbody>
</table>

Bibliography 238
# List of Figures

1.1 Category of the QoS-based web service composition subproblems .......................... 5  
1.2 Example of a workflow with service bindings and constraints on conflicts .................. 9  
1.3 Hybrid clouds ....................................................................................................... 13  
1.4 Example of BPEL program partitioning for a ‘Loan Approval’ program .................... 15  

2.1 The web service model ............................................................................................ 25  
2.2 BPEL-WS process flow. The specification supports structured activities to manage the overall process flow as well as basic activities that involve interactions with services external to the process itself. ................. 31  
2.3 QoS-broker based web service architecture ............................................................ 35  
2.4 Evaluation phase of a cooperative coevolutionary genetic algorithm ....................... 54  

3.1 Basic composition patterns .................................................................................... 59  
3.2 Encoding and decoding ......................................................................................... 72  
3.3 Crossover and mutation ......................................................................................... 73  
3.4 Knowledge-based crossover .................................................................................. 80  
3.5 Comparison of the average fitness progresses during the evolution of the penalty GA with a mutation rate of 0.15, and the penalty GA with a mutation of 0.08 ............................................. 88  
3.6 Comparison of the best fitness progresses during the evolution of the penalty GA with a mutation rate of 0.15, and the penalty GA with a mutation of 0.08 ................................. 89  
3.7 Workflow building block ....................................................................................... 91  
3.8 The effect of the number of abstract web services on computation time ............... 93  
3.9 The effect of the number of concrete web services per abstract web service on the computation time ................................................................. 98  
3.10 The effect of constraint density on the computation time ...................................... 99
3.11 Comparisons of the computation time of the integer programming-based method versus the genetic algorithms for problems of different complexities ............................................. 105
3.12 QoS-based web service selection tool .............................................. 108

4.1 Hybrid cloud ................................................................. 115
4.2 Random-key encoding method ................................................ 128
4.3 Example of uniform crossover for the random-key based algorithm 131
4.4 Generational transition ....................................................... 131
4.5 Random key method for chromosome representation of each subpopulation ................................................................. 142
4.6 Example of the uniform crossover for a subpopulation in the cooperative coevolutionary genetic algorithm ..................................................... 143
4.7 Building blocks for test problems with different numbers of composite services ................................................................. 149
4.8 Building blocks for the second kind of test problems with different numbers of abstract services in each composite service ....... 150
4.9 Number of composite services versus the computation time of the RGA and the CCGA ......................................................... 153
4.10 Problems with different number of composite services versus the converge speeds of the RGA and the CCGA .............................. 154
4.11 Number of abstract services versus the computation time of the RGA and the CCGA ................................................................. 156
4.12 Comparison of the convergence speeds of the RGA and the CCGA for the problem with five abstract services ............................. 157
4.13 Number of candidate cloud services versus the computation time of the RGA and the CCGA ................................................................. 159
4.14 Comparison of the convergence speeds of the RGA and the CCGA for the problem with 25 candidate cloud services ............................. 160
4.15 Comparison of the results of the RGA and the CCGA for the 'Low' deadline constraint problem ................................................................. 161
4.16 Comparison of the results of the RGA and the CCGA for the 'Medium' deadline constraint problem ................................................................. 161
4.17 Comparison of the results of the RGA and the CCGA for the 'High' deadline constraint problem ................................................................. 162
4.18 A test problem for verification .................................................. 163

5.1 Centralised execution mode for a ‘Loan Approval’ BPEL program 169
5.2 Decentralised execution mode for a ‘Loan Approval’ BPEL program 170
5.3 Program dependency graph for the ‘Loan Approval’ BPEL program 173
5.4 A feasible partitioning and the chromosome encoding for the ‘Loan Approval’ example 182
5.5 Precedence dependency graph for the ‘Loan Approval’ example; an infeasible partitioning plan violating precedence dependency constraints; and the partitions dependency graph for this plan 183
5.6 An infeasible partitioning for the ‘Loan Approval’ example violating control dependency constraints 184
5.7 Initialization step for the ‘Loan Approval’ example 185
5.8 Genetic operators used in the partitioning algorithm 186
5.9 Topological sorting on a new precedence dependency graph to calculate the penalty for precedence constraints violation 192
5.10 Control flow graph for the ‘Loan Approval’ example 196
5.11 Polymorph workflow example 204
5.12 Polymorph workflow test instance one 205
5.13 Polymorph workflow test instance two 206
5.14 Polymorph workflow test instance three 207
5.15 Polymorph workflow test instance four 208
5.16 Travel plan workflow and the single-level control condition problem with one building block 209
5.17 The single-level control condition problem with two building blocks 209
5.18 The two-level control condition problems with one building block 210
5.19 The two-level control condition problems with two building blocks 210
5.20 Comparisons of the computation time of the genetic algorithm versus the MDU and PGM heuristic algorithms for partitioning polymorph workflows of different sizes 211
5.21 Comparisons of the computation time of the genetic algorithm versus the MDU and PGM heuristic algorithms for problems of different sizes with single-level control dependency constraints 214
5.22 Comparisons of the computation time of the genetic algorithm versus the MDU and PGM algorithms for problems of different sizes with two-level control dependency constraints 216
5.23 Measured throughput of the single-level one-building block Travel Plan workflow under partitioning topologies found by the genetic algorithm, and the MDU and PGM algorithms 220
List of Tables

2.1 Summary of QoS properties ........................................ 32
3.1 QoS aggregation model ............................................ 58
3.2 GA parameters setting ............................................ 87
3.3 Comparison results of the PGA, the RGA and the HGA for test problems with different numbers of abstract web services .... 94
3.4 T-test results for the pair of the PGA and the RGA, the pair of the PGA and the HGA, and the pair of the RGA and the HGA, for test problems with different numbers of abstract web services .... 95
3.5 Comparison results of the PGA, the RGA and the HGA for test problems with different number of concrete web services for each abstract web service ........................................ 97
3.6 T-test results for the pair of the PGA and the RGA, the pair of the PGA and the HGA, and the pair of the RGA and the HGA, for test problems with different number of concrete web services for each abstract web service ........................................ 98
3.7 Comparison results of the PGA, the RGA and the HGA for test problems with different constraint densities ......... 100
3.8 T-test results for the pair of the PGA and the RGA, the pair of the PGA and the HGA, and the pair of the RGA and the HGA, for test problems with different constraint densities ......... 101
3.9 Comparison results of the integer programming-based method, the three GAs for problems of different complexities .... 104
4.1 Simulation parameters for the experiments ..................... 145
4.2 Comparison results of the RGA and the CCGA for deadline constraint problems with different numbers of composite services 151
4.3 Comparison results of the RGA and the CCGA for cost constraint problems with different numbers of composite services 151
4.4 Comparison results of the RGA and the CCGA for deadline constraint problems with different numbers of abstract web services 155
4.5 Comparison results of the RGA and the CCGA for deadline constraint problems with different numbers of candidate cloud services for each abstract service .................................................. 158
4.6 QoS values of candidate cloud web services for each task involved in the problem ................................................................. 163
4.7 Results of the RGA and the CCGA on the test problem ........... 165

5.1 Test results for the genetic algorithm, against the MDU and PGM heuristic algorithms, for polymorph search workflow partitioning, showing the algorithms’ execution times and the throughputs of the solutions found ............................................. 212
5.2 Test results for the genetic algorithm, and the MDU and PGM heuristic algorithms, for partitioning workflows with single-level control conditions, showing the algorithms’ execution times and the throughputs of the solutions found .................................. 215
5.3 Test results for the genetic algorithm, and the MDU and PGM algorithms, for partitioning workflows with two-level control conditions, showing the algorithms’ execution times and the throughputs of the solutions found ............................................. 217
Statement of Original Authorship

The work contained in this thesis has not been previously submitted for a degree or diploma at any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Signed:  

Date:  

xvii
Acknowledgement

I would like to express my deep gratitude to my principal supervisor Dr Maolin Tang for accepting me as his PhD student so that I had an opportunity to study in Faculty of Science and Technology at Queensland University of Technology (QUT). Many encouragements, numerous supports, valuable suggestions and patient guidance by Dr Maolin Tang led to my successful study at QUT.

I would also like to express my sincere gratitude to my associate supervisor Professor Colin Fidge, for his invaluable expertise, encouragement, rigorous discussions and patience throughout the course of my Ph.D. work.

Also, I would like to thank Dr Frederic Maire and Dr Chun Ouyang for their valuable comments about my work.

Special thanks to QUT and the Cooperative Research Center on Smart Services, and the China Scholarship Council for their support in the form of a Top-Up Scholarship and a Postgraduate Scholarship, respectively.

Finally, I wish to extend my deepest appreciation to my wife, Xiaoyue Zhang, for her love and support during the difficult times. Without her encouragement and patience, the completion of thesis would not have been possible.
List of Publications

The following publications were made during the course of the thesis study:

Journal Papers


Conference Papers


xxi
Chapter 1

Introduction

This dissertation focuses on how to use genetic algorithms for Quality-of-Service (QoS)-based web service composition. QoS-based web service composition raises many challenges, which can be transformed into different optimisation problems. This research uses genetic algorithms to address web service composition problems. The purpose of this research is to develop scalable and effective genetic algorithms to facilitate the delivery of high quality composite web services in a responsive way. The remainder of this chapter elaborates on the motivation, research problems, and main contributions of this research.

1.1 Research Motivation

Web service technology provides a promising way to implement e-Applications, such as e-Business, e-Government, and e-Science. Benefits of web service technology are its inter-operability, decoupling and just-in-time integration. Using web service technology, various e-Applications with complex functions can be
implemented by web service composition, namely, by composing individual web services in accordance with the business process of the e-Applications. The benefit of web service composition originates from the added value generated by the possible interactions and by their large scale rather than by the capabilities of its individual web services separately. The web service paradigm creates tremendous opportunities in various verticals including finance, government, and media.

However, for the success of the web service paradigm in the real world, there is a need to address some critical issues involved in the paradigm. One of the issues is how to manage the QoS for composite web services, such that the delivered composite services to customers can not only meet the customers’ functional requirements, but also satisfy their QoS requirements, which in turn fulfils their expectations and achieves their satisfaction.

Over the past few years, QoS management for composite web services has experienced considerable interest in both industrial and academic fields. The work covers a wide range of areas, i.e. QoS modeling for identifying QoS attributes for web services [71, 19, 89], QoS specification languages [63, 103] for web services to enable the exchange of QoS information between web services, QoS-based service registering and discovery [89, 34], QoS-capable frameworks [72, 116] to support QoS-aware web service composition, and QoS monitoring [120] to check the compliance of the QoS promised to customers. These achievements as
1.1 Research Motivation

Due to the fundamental mechanisms mentioned above mature, QoS-based service composition has become an important issue. In practice, it is very common for customers to expect that they can consume services with an optimal QoS. For example, customers expect to use services with minimal response time, lowest cost, highest reliability, or highest throughput. It is also likely that customers might demand services with multiple QoS criteria being optimised simultaneously. QoS-based web service composition is a problem that studies how to compose web services to create value-added composite web services with the best QoS results and to best satisfy customers’ QoS-related goals.

The QoS-based web service composition problem can be divided into diverse subproblems (see Figure 1.1), focusing on configuring either the external aspect components of composite web services (namely the web service components involved in web service composition) or the internal aspect of components of composite web services (namely the workflow components that define the usage pattern of a collection of web service components). Focusing on configuring the web service components, researchers have put forward the QoS-based web service selection problem [118] and the QoS-based web service allocation and scheduling problem [114]. Focusing on configuring the internal workflow components of composite web services, researchers have introduced the composite web service partitioning problem [72].
The QoS-based web service selection problem is a study of assigning a proper external web service component in terms of QoS to each task of the workflow that specifies a composite web service, such that the aggregated QoS for the composite is the best. It is a planning problem by its nature. The QoS-based web service allocation and scheduling problem is an extension of the QoS-based web service selection problem, by not only considering the planning of web services for composite web services, but also considering scheduling of web services (determining a proper execution time for each selected web service for fulfilling a task). The planning and scheduling problem is more sophisticated than the planning problem. The study of the planning and scheduling problem could result in composite web services with more accurate QoS results. However, this problem is only applicable to build composite services based on the web services deployed on grid-based platforms or cloud-based platforms [42], where scheduling is supported using advanced reservation techniques [95, 94]. The planning problem, albeit less sophisticated, has wider applicability. The composite web service partitioning problem, is a study focused on finding the best decentralised topology for the internal workflow component, in order to enhance the QoS of the composite web service. This subproblem is a supplement of the two other subproblems, by considering the impact of the internal workflow components on the QoS results for the composite web services, which are ignored by the two other subproblems. This subproblem is particularly important for
building communication-intensive composite web services, whose QoS results are closely correlated with the goodness of internal topology of the composite service. The centralised topology and poor designated decentralised topology tend to lead to poor performance for the composite service.

From computational points of view, the above subproblems can be transformed into diverse optimisation problems. These problems are very challenging in the following three respects:

1. Firstly, the problems are large in scale. For example, for the QoS-based web service selection problem addressed in this project, the challenge is how to select a web service for each task involved in a composite web service to generate an execution plan for the composite. Suppose that a composite service has 20 tasks and each task has 30 candidate web services,
then there will be $30^{20}$ possible web service selection solutions. Making the optimal web service selection decision from such a large number of possible solutions is computationally intractable.

2. Secondly, the problems face hard constraints. For example, resource allocation for composite services is an important optimisation problem in the field of QoS-based management for composite services. It is essential to consider the precedence dependence constraints between different tasks. The QoS-based web service selection problem is constrained by the complex inter service dependence and conflict between the candidate services of different tasks. Handling these constraints is necessary to guarantee feasibility of the solution, but is very difficult.

3. Thirdly, the problems may have many conflicting QoS-related objectives that must be handled simultaneously. For example, minimising the response time and maximising the cost of a composite service might be desired by a customer simultaneously. However, the use of short response time individual web services on the one hand, can help to minimise the response time of the composite web service, but on the other hand could lead to a rise in the overall cost for the composite web service, because of the use of high performance but costly individual web services.

It is conjectured that these QoS-based web service composition subproblems
are NP-hard [44], requiring efficient algorithms to determine satisfactory, perhaps sub-optimal solutions. This research investigates how to use genetic algorithms (GA) to address QoS-based service composition problems. GAs are inspired by the process of natural evolution and they work on principles of natural evolution such as inheritance, mutation, selection, and crossover, to generate satisfaction, but do not always provide optimal solutions to optimisation and search problems. GAs have been successfully applied in many problem domains (i.e. computer-automated design [22], electronic circuit design [62], wireless sensor networks [55], etc.) to address complex, large-scale, constrained and multi-objective optimisation problems. These successful experiences also motivate our choice of GAs to address the QoS-based web service composition problems, which are also often characterised as complex, large scale, constrained and multi-objective optimisation problems. The objective of this research is to develop effective and scalable genetic algorithms for QoS-based composite web service optimisation, with the goal of facilitating the delivery of feasible, reliable and high quality composite web services to customers.

1.2 Research Problems

This research focuses on three important subproblems of QoS-based web service composition. Below is an elaboration of the three research subproblems.
1.2.1 Problem One — QoS-Based Web Service Selection with Inter Service Dependencies and Conflicts

This problem arises when building a composite service based on the individual web services provided by many external web service providers. A composite web service is built based on the following process:

1. First of all, we design a workflow for the composite web service. Figure 1.2 is an example of workflow, which consists of 10 tasks W1, W2, W3, . . . , W10.

2. Then, we obtain the information about all available web service implementations (or candidate web services) for each of the tasks in the workflow. This can be done by using a web service discovery tool or a web service broker. The information includes their URLs, the inter-dependencies and mutual conflicts between the web service implementations [106] (an example of mutual conflict between web services is illustrated in Figure 1.2), as well as their QoS values of interest.

3. Finally, we use a web service composition tool to select the best combination of web service implementations accommodating the constraints, which forms the problem to be addressed.

A formal statement of the problem is as follows:
1.2 Research Problems

Given the abstract specification of a composite web service, which is a workflow process, how can we select a web service implementation for each of the tasks in the abstract specification, so that the overall QoS of the composition is optimal, whilst accommodating constraints imposed on the selected web service implementation to guarantee the correctness of the composite service?

The problem is a typical combinatorial optimisation problem [82], and the number of possible combinations of web service implementations for a composite web service grows explosively as the number of tasks involved in the composite service and the number of web service implementations for each task increases. Finding an optimal solution, namely the optimal execution plan for a composite service, is very hard. Moreover, the complex constraints on inter-service dependence and conflict between web service implementations may make even finding a feasible solution very difficult. Therefore, intelligent web
service selection algorithms are required in order to find a feasible and satisfactory solution. However this solution may be a suboptimal solution in terms of the composite service in a reasonable time.

This thesis investigates the problem by addressing the following crucial issues:

- **Problem Model** — Previous research has proposed different problem models for the QoS-based web service selection problem, such as the multi-dimension multi-choice 0-1 knapsack model [49, 117] and the multi-constraint optimal path model [117]. However, none of these models consider constraints on inter service dependence and conflict in the problem. This research defines a problem model with consideration of these constraints.

- **Algorithm Design** — Web service selection algorithms must address two challenges, the optimality challenge and the correctness challenge. The optimality challenge means the requirement of optimising the QoS for a composite service. The correctness challenge means the requirement of satisfying all constraints on inter service dependence and conflict between the selected web service implementations for a composite web service to ensure its feasibility. Most existing web service selection algorithms [119, 117, 18, 88] only address the optimality challenge. This research emphasises how to use genetic algorithms to address both of these
challenges. Since genetic algorithms have different strategies to handle constraints, such as penalty functions, repairing methods, and knowledge-based operators which employ domain knowledge to handle constraints, this research investigates how to use different constraint handling strategies to handle constraints on inter service dependence and conflict.

- Evaluation — This research identifies the impact of the simulation parameters on the efficiency of the proposed algorithms. Based on these parameters, different test instances are constructed. The new genetic algorithms are tested on these test instances, in order to provide precise findings about the strengths and weaknesses of the algorithms.

- Verification — Whether the QoS results of composite web services generated by the proposed GAs are correct is verified on realistic problems.

1.2.2 Problem Two — QoS-Based Resource Allocation and Scheduling for Multiple Composite Web Service on Hybrid Clouds

This problem arises when building composite web services on cloud web services from hybrid clouds [6] (see Figure 1.3). A hybrid cloud is a cloud computing environment in which an organisation provides and manages some resources in-house and has others provided externally. All resources on a hybrid cloud are provided in the form of web services, which are also called cloud web services.
Cloud services can be further categorised into public cloud services and private cloud services. Private cloud services refer to the services provided by internal data centres of a business company or organisation. We use them because they are fast and can be controlled by the organisation, but the number of private cloud services is very limited and there might be competition for the same private cloud service by multiple tasks of composite services. Public cloud services refer to the services provided by public cloud data centres. We use them because their number can be expanded according to our demands, but we have to pay the public cloud service providers.

In such an environment, delivering high quality composite services must address two problems. One is assigning resources to each of the tasks in the composite web service. The other is scheduling the allocated resources when each resource may be used by more than one task and may be needed at different points in time. In addition, this resource allocation and scheduling problem must consider QoS issues, such as execution time, running cost, as well as constraints, such as execution time constraints and running cost constraints for the resulting composite web services.

Based on the above requirements, a formal statement of the problem is presented below:

*How can we effectively allocate and schedule resources for many composite web services simultaneously, such that the overall QoS for all composite services*
are optimal, while satisfying customers’ requirements on deadline constraints or cost constraints imposed on each composite service?

From the perspective of operations research theory, the problem is a typical resource allocation and scheduling problem. It is conjectured that the problem is NP-hard. This research investigates how to use genetic algorithms to address the problem. A characteristic of the problem is that its problem size could be very large, as either the size or number of composite services involved in the problem increases to a certain amount. Therefore, special emphasis for the problem in this research will be on designing genetic algorithms that can scale well as the problem size increases, and will always procedure acceptable, albeit sometimes suboptimal solutions.
1.2.3 Problem Three — Performance-Driven Composite Web Service Partitioning for Decentralised Execution

This problem is significant for communication-intensive web service-based applications or other kinds of web service-based applications that tend to suffer single-server bottlenecks, resulting in the poor performance of these applications. Decentralised execution for these applications would be an alternative to overcome the single-server bottleneck problem [67], but it raises the issue of how to partition a composite service program into subprograms for decentralised execution. Because a composite web service is usually implemented in the Business Process Execution Language (BPEL) [69], this research investigates how to optimally partition a composite service program that is implemented in BPEL into sub BPEL programs for decentralised execution, such that the composite service can achieve satisfactory performance. An illustrative example for a ‘Loan Approval’ BPEL program is shown in Figure 1.4. To improve the performance of the ‘Loan Approval’ composite service, its corresponding BPEL program (on the left of Figure 1.4) can be partitioned into five subprograms (on the bottom right). Then the subprograms can be deployed on separate servers, D0 to D4, and pairs of servers can communicate via asynchronous messaging. The topology of the decentralised execution mode for the partitioning plan is shown on the top right of Figure 1.4. The statement of the problem is presented below:

How can we partition a composite service implemented in a BPEL program
1.3 Major Contributions

In general, this research tackles the three important subproblems of the QoS-based web services composition problem, by developing new problem models.
Introduction

and novel algorithms for the problems, which advance the knowledge of QoS management in web services. More precisely, this project contributes to web service management by developing effective and scalable genetic algorithms for addressing the QoS-based web service selection problem, the QoS-based resource allocation and scheduling problem on hybrid clouds, and the performance-driven composite web service partitioning. The new genetic algorithms can be used by QoS-based web service management tools, to facilitate the automatic delivery of composite services to customers in a responsive and efficient way.

Besides the contribution to web service research, this thesis also contributes to genetic algorithm research. During the development of genetic algorithms, we introduce new genetic operators, new novel local optimizers in order to address the QoS-based web service composition problem. We also provide comprehensive evaluation of the new genetic algorithms, which adds substantially to our understanding of using genetic algorithms to address the QoS-aware web service composition problem. The following section elaborates on the major contributions of this thesis:

1. QoS-based web service selection problem (Chapter 3)

This project supplements the research on constrained QoS-aware web service composition by considering constraints on inter service dependence and conflict and addressing the scalability issues. Although QoS-aware
Web service composition has been intensively studied and many successful approaches have been proposed in the past few years, existing approaches are not scalable to handle large-scale Web service composition problems and do not consider constraints on inter service dependence and conflict. The poor scalability of existing approaches leads to a prohibitive decision time on web service selection for composite services, which is unacceptable for real-world web service provision. Even worse, ignoring constraints on inter service dependence and conflict might lead to the generation of incorrect composite web services.

This study provides a new problem model for the QoS-based web service selection problem that contains constraints on inter service dependence and conflict. The problem is addressed using three new genetic algorithms. They are a penalty-based genetic algorithm, a min-conflicts hill-climbing genetic algorithm, and a hybrid genetic algorithm. To the best of our knowledge, this is the first attempt to use genetic algorithms to address the problem. The penalty-based genetic algorithm contains the formulation of a penalty function to handle constraints on inter service dependence and conflict which has not been investigated before in web service research. The min-conflicts hill-climbing genetic algorithm explores a novel repairing strategy to handle the constraints, resulting in a fast repairing procedure
to quickly repair infeasible individuals. The hybrid genetic algorithm introduces a new knowledge-based operator to handle the constraints, with a local optimiser to further improve solution quality. Experimental results reveal the good scalability and effectiveness of these genetic algorithms, as well as the better scalability of these genetic algorithms than the existing integer programming-based method.

The algorithms could be applied in semantically-rich environments, such as e-science applications where there are growing numbers of functionally equal but technically incompatible web services, and e-Business applications in which business preferences cause complex dependence relationships among web services. The scalable algorithms are suitable to facilitate the management of large scale web service-based systems.

2. QoS-based resource allocation and scheduling for multiple composite web services on hybrid clouds (Chapter 4)

The resource allocation and scheduling problem for composite web services on hybrid clouds is a completely new problem. It presents two main distinguishing features, compared with previous work on resource allocation problems for composite web services. First, in contrast to previous research considering allocation and scheduling for only one
1.3 Major Contributions

composite service, this research considers multiple composite services, which is a more realistic problem. Second, the problem of allocation and scheduling of resources built on hybrid clouds is considered, while most previous research considers grid computing. Hybrid clouds comprise both single-user, small-scale resources (private cloud services) and multi-user, large-scale resources (public cloud services). In the context of grid computing, however, all services are single-user in nature, so applying grid computing algorithms to hybrid cloud problems will not necessarily lead to good use of public services.

This research presents a new formal model for the problem, which is essentially a constrained resource allocation and scheduling problem. To the best of our knowledge, this problem has not been addressed before. This study results in two genetic algorithms for the problem, a random-key genetic algorithm and a cooperative coevolutionary genetic algorithm (CCGA). Both of the algorithms are scalable, making them suitable for large scale problems where the sizes of composite services and the numbers of considered composite services are large. This is especially so for the CCGA. Because of its capability to parallel search on a large problem domain by using a cooperative coevolution model, it shows a clear advantage for solving large scale problems, in terms of solution quality
and convergence speed, compared with traditional genetic algorithms. An extra contribution in this research is that the application of the proposed two algorithms is not limited to the resource allocation and scheduling on hybrid clouds. The algorithms can also be applied to the resource allocation and scheduling problems in grid computing, because from an operations research point of view, the problem in the context of grid computing can be considered as a simplified kind of resource allocation and scheduling problem compared with these problems in the context of cloud computing.

3. The composite web service partitioning problem (Chapter 5)

This research presents an effective and scalable genetic algorithm for the BPEL program partitioning problem. Compared with existing heuristic algorithms, the new GA is more suitable for large scale composite web service program partitioning problems. In addition, the performance of the GA outperforms existing heuristic algorithms, which generates a better deployment topology for a composite web service for decentralised execution so that the composite can have a better performance. In addition, this research also constructs a comprehensive evaluation model for testing the performance of the algorithms for the BPEL program partitioning problems, which was missing before.
1.4 Thesis Outline

Chapter 2 provides the background information of this research. It firstly introduces the basic concepts of web services and QoS. Then, it overviews support concepts and technologies for QoS-based web service management. Besides the basic knowledge of web services, the basic concepts of genetic algorithms, as well as some common extensions of simple genetic algorithms that are used in this research are also introduced in this chapter.

Chapter 3 investigates QoS-based web service selection problem. Since inter-service dependence and conflict could affect the correctness of web service selection but have been largely ignored by most researchers, this chapter addresses the QoS-based web service selection problem with constraints on inter-service dependence and conflicts. This chapter formulates the new problem as a constrained combinatorial problem and presents three new GAs for the problem. Each GA adopts a different constraint handling technique. The first algorithm is a penalty-based GA that uses the penalty function method to handle constraints. The second algorithm is a repairing GA, adopting a fast repairing operator for constraints handling. The last algorithm is a hybrid GA, with a knowledge-based operator to handle constraints. The performance of the three GAs and the integer programming-based method are compared using experiments on a large number of test instances. Finally, a verification experiment is presented to validate the
correctness of the results generated by the GAs.

Chapter 4 studies QoS-based resource allocation and scheduling for multiple composite web services on hybrid clouds. This chapter firstly introduces a formal description of the problem. Then, a random-key GA and a cooperative coevolutionary genetic algorithm (CCGA) are presented to address the problem. The major feature of the random-key GA is its random-key representation, which is used to overcome the feasibility issues involved in the problem. The major feature of the CCGA is a cooperative coevolution model to deal with the increasing complexity involved in the problem. Finally, experiments are presented in this chapter to compare the performance of the two GAs for the problem.

Chapter 5 investigates the composite web service partitioning problem for decentralised execution. This chapter presents a new GA for the problem, the core idea behind which are novel strategies for handling data dependence constraints and control data dependence constraints, as well as a local optimizer for improving solution quality. The chapter presents a comprehensive evaluation model in order to fully test the performance of the proposed GA and existing heuristic algorithms for the problem. The obtained experimental results are presented to demonstrate the effectiveness and scalability of the algorithms. The chapter also presents the verification experiment to validate the correctness of the simulation results.

Chapter 6 contains a summary of the results, recommendations for future research, and the conclusion of the study.
Chapter 2

Background

Web services are emerging technology that enables disparate applications running on different machines to exchange data. No additional proprietary third-party software or hardware to integrate with one another is required. The long term goal of web service technology is to enable distributed applications that can be dynamically assembled via web service composition according to changing business needs. When discussing web service technology, quality-of-service (QoS) is a significant concern, as it is a critical factor which directly determines the success or failure of a web service-based application. Among QoS-related issues in the context of web services, the provision of high quality composite web services is a central one. However, the dynamic, unpredictable and large-scale features of composite services make the realisation of high quality composite services a difficult task. Genetic algorithms, i.e. population-based meta-heuristic optimisation algorithms, are candidate solutions which can be applied to alleviate the difficulties involved in the provision of high quality composite services.
In this chapter, background information on web services and genetic algorithms are introduced. Firstly, the concepts of web service and web service composition are presented. Then, QoS modeling for web services and the major challenges in the provision of high quality composite services are described. Finally, the basic concepts of genetic algorithms, as well as some advanced genetic algorithms which will be used in this thesis are presented.

2.1 Basic Concepts of Web Service and Web Service Composition

2.1.1 What is a Web Service?

The term “web service” is defined in different ways. A widely accepted definition is specified by the World Wide Web Consortium (W3C). They define a web service as “a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols” [3].

This definition reveals how a web service should work — it could be defined, described, and discovered. A more clear demonstration of the work mechanism of a web service is the web service model [104] (see Figure 2.1). In the model, there are three kinds of entities; service providers, service broker and service consumers. The role of service providers is to provide software applications for
specific needs as services. Service providers publish and update their services so that they are available on the Internet. From a business perspective, service providers are the owners of the services. An architectural perspective, they are the platforms that hold the implementations of the services. The role of a service broker is to provide a searchable repository of service descriptions. Service providers publish their services; consequently service requesters find services and obtain binding information. The service consumer retrieves the information from the broker and uses the service description obtained to bind and invoke the web service.

![Diagram of the web service model]

The web service model is enabled by a number of XML-based web service standards. The core standards include the Web Service Description Language (WSDL) [24], the Universal Description Discovery and Integration standard (UDDI) [9], and the Simple Object Access Protocol (SOAP) [48]. These three standards are intended to support the activities of service description, discovery,
and invocation, respectively. Besides the three standards, there are also some complementary standards, such as WS-Security, WS-Policy and WS-Trust [3]. All these standards, plus the three core standards, comprise the minimum infrastructure required by the web service paradigm, which is also called the web services technology stack [3].

Currently, web service technology is being used by many enterprises and organisations to implement their software applications as web services for different needs. The web services can be built based on different types of application modules. For example, a web service could be [83]:

- a self-contained business task, such as a funds withdrawal or funds deposit service;
- a full-fledged business process, such as the automated purchasing of office supplies; or
- a service-enabled resource, such as computation-resources, storage resources and network-resources.

Web service-based software and applications exhibit many benefits compared to traditional applications [104]:

1. **Easy and fast deployment**, as new web services can be developed by reusing and/or combing existing ones.
2. Inter-operability, as any web service can interact with other web services, thanks to the set of web service standards. Various applications can be deployed as web services and be inter-operative with each other. Moreover, the collaboration of web services is truly platform and language independent.

3. Decoupling and just-in-time integration, based on the notion of building applications by discovering and orchestrating network-available services.

4. Reduced complexity by encapsulation. Consumers don’t care about the implementation, instead they only care about the type of behaviour a service.

2.1.2 What is Web Service Composition?

An attractive feature of web services is that existing web services can be integrated together to create value-added composite web services that satisfy users’ needs. The process of creating a composite web service is called web service composition [119]. For example, a travel planner web service can be built by aggregating a flight booking web service, a travel insurance web service, an accommodation booking web service, a car rental web service, and an itinerary planning web service.

Web service composition is addressed by different approaches, which are
Background

classified by Nikola et al. [75] into the following five types: the workflow-based approach [69], the algebraic process composition approach, the model checking-based approach [15], the web component-based approach [113], and the semantic web-based approach [4]. Among these approaches, the workflow-based approach is the most popular one, and has been standardised and widely adopted to build web service-based applications in the communities of e-commerce, e-science, e-business and so on. In this thesis, all the research problems are investigated with the assumption that composite services are implemented using the workflow-based approach, specifically using the BPEL-based workflow approach which is introduced later in this section.

In the workflow-based approach, a workflow is used to describe the usage pattern of a collection of web services. The composition of those services provides the functionality needed to achieve a certain business objective. The workflow is used to: 1) Describe how activities, implemented as web services are combined. 2) Specify the order in which these steps are executed, and the decision points where steps may or may not have to be performed, namely the control flow for the specified composite service. 3) Specify the passing of data items between the steps involved, namely the dataflow for the composite service.

Many web service composition languages have been proposed for implementing workflow-based web service composition. Typical languages include WSFL [61], XLANG [99], WS-BPEL [69] and WSCI [5] (a complete list of ex-
isting web service composition languages can be found elsewhere [69]), among which WS-BPEL has emerged as a de facto standard. This thesis also uses WS-BPEL as the research basis. The following part of this section is a brief introduction to WS-BPEL.

WS-BPEL [69] (Business Process Execution Language for Web Services, also called BPEL4WS or BPEL) was proposed by IBM, Microsoft and BEA jointly as a specification for modelling the behaviour of web services in a business process (namely a workflow) interaction. The main feature of the specification is that it provides an XML-based grammar for describing the control logic required to coordinate web services participating in a process flow. Furthermore, this grammar can be interpreted and executed by an orchestration engine, which is controlled by one of the participating parties. The engine coordinates the various activities in the process, and compensates the system for errors [84]. Each BPEL process (also called a BPEL program) can be deployed as a web service (namely a composite service) that can be invoked by external clients.

A composite web service implemented in WS-BPEL primarily contains two kinds of activities; basic activities and structured activities. A basic activity is an instruction that interacts with external web services. Example of basic activities are **Invoke, Send, Receive**. Structured activities manage the overall process flow, namely, specifying the execution structures for the invoked web services. WS-BPEL contains a variety of structured activities (i.e. **Sequence**, **Sequence**,
If, While, Pick, Flow, etc.), to support execution structures such as sequential flow, conditional flow, iterative flow and conditional flow. Supporting these structured activities makes WS-BPEL expressive to build a composite service with complex business logic. Besides the two main kinds of activities, there are some other kinds of activities, i.e. dataflow-related activities (such as Variable and Assign) used for managing dataflow, event handling activities used for handling message-related and time related events, etc. Figure 2.2 provides a more intuitive description of a BPEL-WS process flow. Currently, there are more than 20 execution engines supporting the execution of such BPEL process flows. The most popular engines include Oracle BPEL Process Manager [69], IBM WebSphere Business Integration Server Foundation [90], IBM BPWS4J [28], and Microsoft BizTalk [111].

2.2 QoS with Web Services

Every application built by web service composition techniques provides a certain functionality. Conversely the application also exhibits a non functional behaviour. This behaviour is called the Quality-of-Service (QoS). For example, an application will produce a certain response time for each customer request, and will charge each request a certain amount of money. The length of the response time, and the amount of the cost, as well as the value of some other QoS parameters for services and products, will directly determine customers’ satisfaction with the service. In
2.2 QoS with Web Services

Figure 2.2: BPEL-WS process flow. The specification supports structured activities to manage the overall process flow as well as basic activities that involve interactions with services external to the process itself.

... and this will impact the success of web service-based applications and products in such domains of e-science and e-business. Therefore, when delivering services to customers, we consider not only the functionality of the services, but also their QoS. To this aim, one important requirement is the management of QoS for web service-based applications — making use of appropriate and effective control methods to create quality services and products to fulfill customers’ expectations and satisfaction [19].

2.2.1 QoS Properties for Web Services

The first issue for managing QoS for web service-based systems is to identify proper QoS properties for web services. The concept of QoS for web service is
inspired by the application of QoS in networking [27], real-time applications [25] and middleware [78]. In the past few years, modelling QoS for web service has been a big concern for researchers [71, 89, 81, 79]. At the moment, there is no standard QoS modelling technique and different organisations and researchers put forward different QoS categories for web services. The three commonly used QoS models are presented in Table 2.1.

Although each of the three models provides a comprehensive description of the QoS attributes for web services, only Zeng [119]’s model covers how to calculate the QoS for composite web services in terms of a QoS aggradation formula, which is essential when discussing QoS-aware web service composition. Now, Zeng’s QoS aggradation formula is widely adopted in the area of QoS-aware web service composition, especially when discussing QoS-related optimisation problems for web service composition. This thesis also follows Zeng’s model and considers how to optimise the QoS properties in this model.

The meanings of the QoS properties in this QoS model are as follows:

Table 2.1: Summary of QoS properties

<table>
<thead>
<tr>
<th>UML QoS-Profile [79]</th>
<th>Ran’s QoS Model [89]</th>
<th>Zeng’s QoS Model [119]</th>
</tr>
</thead>
<tbody>
<tr>
<td>throughput</td>
<td>throughput</td>
<td>throughput</td>
</tr>
<tr>
<td>latency</td>
<td>response time</td>
<td>response time</td>
</tr>
<tr>
<td>efficiency</td>
<td>cost</td>
<td>cost</td>
</tr>
<tr>
<td>availability</td>
<td>availability</td>
<td>availability</td>
</tr>
<tr>
<td>reliability</td>
<td>reliability</td>
<td>reliability</td>
</tr>
<tr>
<td>security</td>
<td>security</td>
<td>reputation</td>
</tr>
</tbody>
</table>
2.2 QoS with Web Services

- **Response Time** — Response time measures the expected delay in seconds between the moment when a request is sent and the moment when the results are received.

- **Cost** — Cost represents the amount of money the service requester has to pay for invoking a service.

- **Reliability** — Reliability represents the ability of a service to function correctly and consistently and provide the same service quality despite system or network failures.

- **Availability** — Availability represents the probability that a service is available.

- **Reputation** — Reputation is a measure of a service’s trustworthiness. It mainly depends on end users’ experience of using the service.

- **Throughput** — Throughput represents the number of web service requests served during a given time period.

2.2.2 QoS-Based Management for Web Service-Based Systems

Having chosen the QoS properties for web services, the problem now is how to guarantee these QoS criteria when delivering services. This entails the development of effective QoS-based management methods for web service-based systems. Existing work towards this aim ranges from exploring the web
service frameworks [72, 116, 100] to support QoS-capable service provision, to developing QoS-related enabling techniques [63, 103, 89, 34, 120] for the proposed QoS-capable web service frameworks.

Regarding the QoS-capable web service frameworks proposed by previous researchers, the most common one is the QoS-broker based framework depicted in Figure 2.3. In this model, a service provider needs to publish its service before the service can be invoked by customers. When sending a registry request (denoted by P1 in Figure 2.3) by a service provider, QoS information must be provided by the service provider, and the QoS service broker acknowledges (P2) the registry request once the service is registered.

The execution phase consists of the four steps listed below [72]:

1. B1: A customer sends a request, containing QoS information desired by the customer.

2. B2: A QoS-broker negotiates on behalf of service providers with the customer regarding the QoS. This negotiation can result in QoS requests being accepted, rejected, or in counter offers (for degraded QoS levels) being made to the consumer, who can accept or reject the counter offer.

3. B3: Once the offer is accepted, the customer can invoke the service.

4. B4: Service providers provide services in a way that the QoS can be guaranteed.
### 2.2 QoS with Web Services

![QoS-broker based web service architecture](image)

Figure 2.3: QoS-broker based web service architecture

Such a QoS-capable framework raises many technical issues. Below is a summary of the major research issues and existing solutions for these issues:

- **Enabling the exchange of QoS information between all parties (namely service providers, a QoS broker and customers)** the first issue must be addressed. Many QoS specification languages have been proposed to solve this issue. Typical ones include IBM’s Web Service Level agreement (WSLA) language [63], HP’s Web Service Management Language (WSML) [2] and the Web Service Offering Language (WSOL) proposed by Tosic et al [102]. With these QoS specification languages, customers, brokers and service providers can communicate about QoS requirements for web services.

- **QoS-based service registry and discovery** are also necessary. Many researchers have developed QoS brokers to take the role of the UDDI
repository, to support QoS-based service registry and discovery [89, 33].

This work can be further classified into developing an integrated QoS broker by extending a UDDI repository and developing a dedicated QoS broker separating from a UDDI repository. An example of the former kind of QoS brokers is Ran’s new UDDI model [89], and an example of the latter kind is Degwekar’s QoS broker [33]. Whatever the type of a QoS-broker, it is responsible for establishing the interaction between customer and service providers with QoS information.

- Effective configuration mechanisms for web service composition are also imperative, to ensure the promised QoS on the composite services delivered to customers. However, this issue has not yet been well addressed and is the emphasis of research discussed in detail in Section 2.2.3.

- QoS monitoring is also necessary to monitor all available offers and the current client requirements in order to check the compliance of offers. Representative work in this area is Tian’s [100] QoS framework that was proposed to with the consideration of QoS monitoring.

### 2.2.3 QoS-Based Configuration Methods for Composite Services

To develop QoS-based configuration methods for composite services, a
fundamental requirement is to identify what factors have impact on the QoS of composite services. When a composite service resides in a complex environment which contains a workflow engine, external web service components and the network itself, all of the elements constituting the composite web service can potentially directly or indirectly affect certain QoS properties. Below are some illustrative examples of how these components affect the QoS of a composite service.

- For a workflow-based composite web service, its QoS relates to the QoS of all individual web service components that participate in web service composition, and the workflow structure. More precisely, the QoS of a composite service is the aggregation of the QoS of all individual web service components that participate in the composition, which is revealed by the QoS aggregation model for workflow-based composite services [50, 19]. (An introduction to the model can be found in Section 3.1.)

- The performance of the workflow execution engine component hosting a composite service can also affect the performance of the composite when the engine is suffering a high load [67]. In this case, the performance of the composite tends to be degraded.

- The network component takes the role of transferring data between the workflow execution component and web service components. Data
propagation delays on the network contribute to the response time of a composite service [20].

Considering the impact of these elements on the QoS of a composite service, different QoS-based configuration methods have been proposed [35, 119, 20]. The configuration actions of these methods are taken on different layers of a service-based application ranging from the physical layer to the application layer. A typical example of the physical-level configuration method is the dispatching strategy to dispatch customers’ requests to different cluster servers hosting the same composite service, to avoid the single server bottleneck problem of the workflow execution engine [35]. Different from physical-level configuration methods, application-level configuration methods concentrate on configuration of the software components, such as the BPEL program for a composite service [20], the external web service components [119] and so forth. In this thesis, only application-level configuration methods are discussed. As introduced in the previous chapter, three typical application-level configuration methods are investigated as listed below:

1. QoS-based web service selection for a composite service — This is done to configure a composite web service via the selection of external web service components based on their QoS, such that the aggregated QoS for the composite can ensure customers’ requirements.
2. QoS-based web service allocation and scheduling — This is a more complicated configuration strategy, which not only considers how to select external web service components for the tasks involved in a composite service, namely the allocation. However, also considers when the allocated web service may be used by different tasks at different points of time, namely their scheduling. This configuration mechanism is applied to build composite services based on the web services deployed on grid-based platforms or cloud-based platforms [42] where scheduling is supported by using advanced reservation techniques [95, 94].

3. Performance-driven decentralised execution of a composite service —
   This is a configuration method put forward to overcome the single-server bottleneck issue. This issue was raised by the high load on the workflow execution engine for a composite service and/or by the high load on the network. This method overcomes the single server bottleneck by partitioning the BPEL program for a centralised composite service into sub BPEL programs for decentralised execution. It is believed that a good partitioning plan can help to improve the performance of a composite service significantly.
2.2.4 QoS-Based Composite Services Optimisation

Using the QoS-based configuration methods alone is insufficient to ensure satisfactory composite services. QoS-based composite services optimisation is also required to best fulfill customers’ QoS requirements. To this aim, we need to draw on optimisation techniques to produce the best configuration decisions.

As introduced in the previous section, We consider three different configuration methods, each of which introduces a QoS-based optimisation problem.

An optimisation problem, from the operations research point of view, is a problem of finding the best solution from all feasible solutions. Formally, an optimisation problem $A$ is a quintuple $(I, f, m, g, c)$, where

- $I$ is a set of instances;
- given an instance $x \in I$, $f(x)$ is the set of feasible solutions;
- given an instance $x$ and a feasible solution $y$ of $x$, $m(x, y)$ denotes the measure of $y$, which is usually a positive real;
- $g$ is the goal function, and is either min or max; and
- $c$ is the constraint set.

The goal is then to find for some instance $x$ an optimal solution, that is, a feasible solution $y$ with $m(x, y) = g\{m(x, y') | y' \in f(x)\}$, satisfying all constraints in $c$.
In general, QoS-based optimisation problems for composite services have the following features:

- The objective is to minimise or maximise a QoS criterion for a composite service, or minimise or maximise multiple QoS criteria simultaneously, or minimise or maximise QoS criteria for multiple composite services simultaneously.

- The candidate solution space is usually large. For example, for the QoS-based web service selection problem, the solution space relates to the size of the workflow process of a composite service in terms of the number of tasks, and the number of candidate web service components for each task. For the resource allocation and scheduling problem, the candidate solution space increases as the number of composite services involved in the problem increases, the size of each composite service increases, as well as an increase in the number of candidate services for each task. For the BPEL program partitioning problem, the candidate space relates to the BPEL program size.

- Various constraints might exist, such as QoS constraints, constraints on inter-service dependency and conflict. Precedence constraints between tasks involved in a composite service is the problem for resource allocation and scheduling. For the BPEL program partitioning problem, there are data
dependency constraints and control dependency constraints.

This thesis uses genetic algorithms (GA) to provide feasible solutions to these problems. Although a genetic algorithm will not always produce an optimal solution, the primary advantage of GA over other optimisation techniques is its robustness in non-polynomial search \[30\]. As is demonstrated in the following section, a GA works through function evaluation, not through differentiation or other means. Because of this approach, a GA does not care what type of problem it is asked to solve, only that it is properly coded in the GA’s chromosome. Therefore, a GA is a promising method to solve a wide range of problems, and is suitable for QoS-based web service composition problems. Furthermore, a GA has techniques suitable for problems characterised as multi-objective, with complex constraints and large scale difficulties, like the problems studied in this thesis.

### 2.3 Overview of Genetic Algorithms

This section is an introduction to genetic algorithms. It defines the terms used in the remainder of the dissertation, and describes how a typical genetic algorithm works. In addition, it introduces some advanced features of genetic algorithms. Such features include constraint handling techniques, strategies for multiple tasking and conflicting problems and the cooperative co-evolution models, which are needed for solving QoS-based web service composition problems.
2.3 Overview of Genetic Algorithms

2.3.1 What is a Genetic Algorithm?

Genetic algorithms (GAs) [45] are search methods which were developed by John Holland and are based on the principles of natural selection, a biological process in which stronger individuals are likely the winners in a competitive environment. GAs are usually used for producing acceptable solutions to optimisation problems when their search space cannot be traversed efficiently by traditional optimisation methods, such as gradient descent methods, or heuristic-based methods.

A GA operates on a population of individuals. Each individual in the population represents a possible solution to the optimisation problem. Individuals are evaluated depending upon their fitness. The fitness level indicates how well an individual of the population can solve the optimisation problem. A GA usually starts with a randomly generated population. Then, an evolution process of the population takes place, in the manner of a transition from one population to the next via the application of the genetic operators: selection, crossover, and mutation. Through the selection process, the fittest individuals will be chosen to go to the next population. Crossover exchanges the genetic material of two individuals, creating two new individuals. Mutation arbitrarily changes the genetic material of an individual. The application of the genetic operators upon the individuals of the population continues until a sufficient solution of the optimisation problem is found. The solution is usually achieved upon a predefined
Background

A stopping condition, i.e., a certain number of generations is reached, or the amount of variation of individuals between different generations is agreed, or a predefined value of fitness. The result may not be an optimal solution, but the algorithm can be calibrated so that it always produces a feasible result that satisfies certain criteria. The pseudocode of a classical genetic algorithm is shown in Algorithm 1.

**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. RECOMBINE pairs of parents
   6. MUTATE the resulting offspring
   7. EVALUATE each candidate solution
4. end

### 2.3.2 Components of a Genetic Algorithm

In general, a GA contains a number of components. The essential components are: representation, fitness function, population, parent selection mechanism, genetic operators such as crossover and mutation, and elitism. Below is an elaboration of these components.

**Representation**

A GA works directly on a coding space that can be understood by the algorithm rather than a solution space for solving a problem. Representation means transforming a solution into a certain code that can be processed by the GA.
Representation links “real world” to the “GA world”, to set up a bridge between the original problem context and the problem-solving space where evolution takes place [37].

An individual (or chromosome) is a single solution, which contains pure ‘genetic’ information that the GA uses. A chromosome consists of genes, each of which is the GA’s representation of a single factor for a control factor.

The encoding of the chromosome representation may vary according to the nature of the problem itself. In general, a bit string encoding [29] is the classic approach, because of its simplicity and traceability. However, a string-based representation may pose difficulties for, and sometimes unnatural obstacles to, some optimisation problems, e.g., the graph coloring problem [98]. The use of other encoding techniques, such as real number representation [52], order-based representation [29] for bin-patching and graph colouring, embedded lists for factory scheduling problems, variable element lists [74] for semiconductor layout, and even LISP S-expressions [58], have therefore been explored.

**Fitness Function**

The objective function of a problem is the main way of providing the mechanism for evaluating the status of each chromosome. This is an important link between the GA and the system.

A fitness function is a particular type of objective function that prescribes
the quality of a solution (that is, a chromosome) in a GA so that that particular chromosome may be ranked against all the other candidate chromosomes. Good candidate chromosomes are allowed to breed and mix their datasets by any of several techniques, producing a new generation that will be superior.

**Population**

A population is a collection of individuals.

**Parent Selection Mechanism**

During each successive generation, a proportion of the existing population is selected to breed a new generation. Individual solutions are typically selected through a fitness-based process, where fitter solutions (as measured by a fitness function) are typically more likely to be selected. Certain selection methods rate the fitness of each solution and preferentially select the best solutions. Other methods rate only a random sample of the population, as this process may be very time-consuming. Typical parent selection mechanisms include the rank-based selection, the proportional selection e.g., a roulette wheel selection is a popular proportional selection, and the tournament selection [12].

**Genetic Operators**

Crossover is the process by which two parent chromosomes recombine to create a new offspring chromosome. The idea behind crossover is that by mating two
individuals with different but desirable features, we can produce an offspring that combines these desirable features. It is analogous to reproduction and biological crossover, exchanging of genetic material between homologous chromosomes.

Typical crossovers include one-point crossover, two-point crossover, and uniform crossover [7]. In addition, we could also design a specific crossover operator by incorporating some domain knowledge for a specific problem.

Mutation is a genetic operator that alters one or more gene values in a chromosome from its initial state. This can result in entirely new gene values being added to the gene pool. With these new gene values, the GA may be able to arrive at a better solution than was previously possible. Mutation is an important part of the genetic search as it helps to prevent the population from stagnating at local optima. Mutation occurs during evolution according to a user-definable mutation probability. This probability should usually be set fairly low (0.01 is a good first choice). If it is set too high, the search will turn into a primitive random search.

**Elitism**

Elitism is the process of selecting the better individuals, or more to the point, selecting individuals with a bias towards the better ones. Elitism is important since it allows the solutions to get better over time.
2.3.3 Constraint Handling Techniques for GAs

In practice, many problems are constrained and ignoring the constraints of these problems will lead to a generation of invalid solutions. A typical GA however, cannot handle constraints directly, because the major operators of classical GAs (i.e. crossover and mutation) are “blind” to constraints [23]. In other words, two feasible parents might generate infeasible offspring via typical genetic operators. As a result, the feasibility of a final solution cannot be guaranteed using a typical GA.

To enable GAs to handle constraints, constraint handling techniques are required. Existing constraint handling techniques for GAs can be roughly classified into the following four types [26]:

1. Penalty-based methods — These methods use a penalty function that reduces the fitness of infeasible solutions. The fitness is reduced in proportion to the number of constraints violated, or to the distance from the feasible region. In this way, the infeasible solutions will have less chance to ‘survive’ during evolution.

2. Methods based on preserving the feasibility of solutions — These methods construct a problem specific representation and suitable initialisation, crossover, and mutation operators such that the feasibility of a solution is always satisfied.
3. Methods which make a clear distinction between feasible and infeasible solutions — Repairing GAs fall into this category. They take infeasible solutions and “repair” them. The repairing procedure is a local search to reduce or remove the constraint violation, rather than to simply improve the value of the fitness function, as is done by a penalty-based GA.

4. Other hybrid methods — These methods combine genetic search procedures with other methods, such as memetic algorithms [80], “ant colony” and local search algorithms [54, 121, 96]. These algorithms usually provide a possibility of conducting an efficient search of the feasible search space.

It is difficult to say which of the above constraint handling methods is the best, because each presents some strength or weakness for a specific problem. In general, penalty-based methods are the most commonly adopted because they are the simplest technique to implement, only requiring a straightforward modification of the objective function. However for many constrained problems, repairing methods, preserving feasibility methods, or hybrid methods outperform penalty-based methods in terms of solution quality. This is because a penalty-based method usually uses little or no domain knowledge, while the other methods can use knowledge that has proven useful to improve search quality.
2.3.4 Techniques for Solving Multi-Objective Problems

GAs for multi-objective problems can be categorised as plain aggregation, population-based non-Pareto and Pareto-based approaches [31, 41].

- **Plain aggregating approaches:** The approaches under this scheme combine $m$ objective functions into a single objective function using aggregating operators. These methods transform multiple objectives into a single one. There are different aggregating operators for the transformation, such as the weighted sum method, goal programming, and goal attainment. In this thesis, we primarily use the weighted sum method for solving the research problems with multiple objectives.

- **Population-based non-Pareto approaches:** A typical approach is the vector evaluated GA. In this approach, the total population is divided into a number of populations equal to the number of objective functions to be optimised. Each population is used to optimise each objective function independently. The populations are then shuffled together followed by conventional crossover and mutation operators.

- **Pareto-based approaches:** The methods under this scheme evaluate the fitness of the evolved solutions using the dominance relationship. The non-dominant solutions with respect to the members of the current populations
2.3 Overview of Genetic Algorithms

are considered to the fittest.

2.3.5 Cooperative Coevolutionary Genetic Algorithms

How to apply GAs effectively to solve increasingly complex problems in the real world is a big challenge. Many researchers have explored the extension of traditional genetic algorithms to solve problems with a large problem domain. One such effort is the introduction of the coevolution model into a GA [21, 85].

The coevolution model is inspired by the biological coevolution process in nature, “the change of a biological object (or a species) triggered by the change of a related object”. The coevolution model in GAs simulates the biological coevolution process to aid the generation of solutions to a range of difficult problems. The main features of a coevolution model are described below [85]:

- There are multiple subpopulations for evolution in a GA, unlike traditional GAs with a single population for evolving.
- Each subpopulation can be treated as a distinct species, representing a potential component to the greater problem to be solved. These subpopulations evolve in isolation to one another, using genetic operators adopted in traditional GAs. Each is only responsible for solving a specific subproblem of the target problem.
- Subpopulations must also interact with each other periodically to encourage
cooperation in solving the target problem, by rewarding individuals for their joint effort toward solving the entire problem.

An illustrative example of the coevolution model is depicted in Figure 2.4. As can be seen from the figure, there are three species in the model. Each species evolves in its own population and adapts to the environment through the repeated application of a GA. The figure shows the fitness evaluation phase of the GA from the perspective of each of the three species. Although most of the implementations of this model utilise a sequential pattern of evaluation, where the complete population of each species is evaluated in turn, the species could also be evaluated in parallel. To evaluate individuals from one species, collaborations are formed with representatives from each of the other species.

The advantage of this model is that it permits effective function decomposition. Each subpopulation is effectively solving a smaller, more tractable problem.

2.4 Summary

This chapter presented the background information on web services and genetic algorithms, which is helpful for the readers who may not be familiar with some of the material needed to follow this thesis. Firstly, the concepts of web service and web service composition were presented. Then, it overviewed support concepts and technologies for QoS-based web service management. This chapter also presented the QoS modeling for web services and the major challenges in the
provision of high quality composite services. Finally, the basic concepts of genetic algorithms, as well as some advanced genetic algorithms which will be used in this thesis were presented.
Figure 2.4: Evaluation phase of a cooperative coevolutionary genetic algorithm
Chapter 3

QoS-Based Web Service Selection with Inter-Service Dependencies and Conflicts

Web service composition is a process for building a web service-based application. Introducing QoS in web service composition raises many challenges. For instance, QoS modelling for composite web services and designing a QoS based framework for real-time monitoring of web services. Additionally, a further challenge is how to select a web service for each task involved in a composite service, such that the overall QoS for the composite service is optimal. The latter challenge is the so-called QoS-based web service selection problem. In practice, QoS-based web service selection is usually not a stand-alone operation. The selection should consider constraints on dependency and conflict between web services to guarantee the correctness of the composite web service. This chapter investigates the QoS-based web service selection problem with constraints on inter-service dependency and conflict, a hard computational problem. It
introduces three new genetic algorithms (GAs) for the problem, each of which adopts a different constraint handling strategy for the problem.

3.1 Introduction

As introduced in Chapter 2, most web service-based applications are built through web service composition. When developing web service-based applications, QoS-aware web service composition is a must. Jager [51] in his thesis has identified the main steps of QoS-aware web service composition, which are summarised below:

1. **Design of composition**: A workflow designer defines the workflow process for a composite web service using a web service composition language (i.e. WS-BPEL [69]). The outcome of this step is a composite web service program that is used to specify the composite service. A sample workflow process is shown in Figure 1.2, in which a composite web service contains ordered abstract web services (or tasks) defined based on a specific business logic.

2. **Discovery of concrete web services**: A service broker discovers functionally suitable concrete web services (or implementations) for each abstract web service in the composite web service.

3. **QoS-based web services selection**: According to a customer’s QoS preference, this step involves selecting a concrete service for each of the
abstract Web services in the abstract specification. The outcome of this step is an execution plan for the composite web service.

4. **Binding web services and execution**: Based on the execution plan, this step involves binding the selected concrete web services for the composite service, and executing the composite. Traditionally, web service binding happens at design time, which is not favourable for QoS-aware web service composition. Customers’ QoS requirements are only known when they send requests for a service, and design-time binding of web services might not be able to guarantee the QoS requirements. Thanks to the late-binding technique [66], the binding of a web service can be deferred to execution time, which enables more flexible web service composition in order to satisfy customers’ QoS requirements.

Among these steps, QoS-based web services selection is a critical step for success or failure in guaranteeing QoS for the composite. The selection decision directly determines the QoS values of the composite web service and is addressed in this chapter.

There are two challenging issues involved in QoS-based web service selection, the optimality issue and the correctness issue. As has been demonstrated in the previous chapter, an optimal QoS for the composite web service is always expected by customers. The ultimate goal of QoS-based web service selection is
therefore to find the best execution plan for the composite web service. This is the so-called optimality issue. In practice, web service selection is not a standalone operation, as it is also restricted by inter-service dependencies and conflicts [106].

The meaning of inter-service dependence and conflict will be explained later. An infeasible composite web service is the result of two conflicting web services being selected for a composite service. Careful selection of the web services for a composite web service is required to avoid generating an infeasible occurrence. This is the so-called correctness issue. Summing up the two aspects of challenges, the problem could be described as follows:

*Given the abstract specification of a composite web service, how can we select a web service implementation for each of the tasks in the abstract specification, so that the overall QoS of the composition is optimal, whilst accommodating constraints on inter-service dependence and conflict to guarantee the correctness of the composite service?*

The problem presents a number of features as listed below:

<table>
<thead>
<tr>
<th>QoS Attr.</th>
<th>Sequence</th>
<th>Conditional</th>
<th>Parallel</th>
<th>Iterative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (T)</td>
<td>$\sum_{i=1}^{m} T(t_i)$</td>
<td>$\sum_{i=1}^{n} p_{ai} \times T(t_i)$</td>
<td>$\max{T(t_i)_{i \in {1,...,p}}}$</td>
<td>$k \times T(t)$</td>
</tr>
<tr>
<td>Cost (C)</td>
<td>$\sum_{i=1}^{m} C(t_i)$</td>
<td>$\sum_{i=1}^{n} p_{ai} \times C(t_i)$</td>
<td>$\sum_{i=1}^{p} C(t_i)$</td>
<td>$k \times C(t)$</td>
</tr>
<tr>
<td>Availability (A)</td>
<td>$\prod_{i=1}^{m} A(t_i)$</td>
<td>$\sum_{i=1}^{n} p_{ai} \times A(t_i)$</td>
<td>$\prod_{i=1}^{p} A(t_i)$</td>
<td>$(t)^k$</td>
</tr>
<tr>
<td>Reliability (R)</td>
<td>$\prod_{i=1}^{m} R(t_i)$</td>
<td>$\sum_{i=1}^{n} p_{ai} \times R(t_i)$</td>
<td>$\prod_{i=1}^{p} R(t_i)$</td>
<td>$(t)^k$</td>
</tr>
</tbody>
</table>
3.1 Introduction

- QoS of a composite web service correlates with QoS of the individual web services that participate in service composition, as well as with the workflow structure of the composite. As demonstrated in the previous chapter, web service composition languages offer flexible composition patterns for defining a composite web service, such as sequential, parallel, conditional, and iterative patterns, as is shown in Figure 3.1. Researchers study how to measure the QoS of a composite web service and have put forward QoS aggregation models to reveal the correlations among the QoS of a composite, QoS of individual services, and the composition patterns. Table 3.1 presents parts of the QoS aggregation model proposed by Cardoso [19]. From the table we can see that, for a sequential pattern
of tasks \{t_1, \ldots, t_m\}, the Time and Cost aggregation functions are additive, while Availability and Reliability aggregation functions are multiplicative.

The conditional pattern of Cases 1, \ldots, n, with probabilities \(p_{a1}, \ldots, p_{an}\) such that \(\sum_{i=1}^{n} P_{ai} = 1\), and tasks \(\{t_1, \ldots, t_m\}\) respectively, is always evaluated as a sum of the attribute value of each task, times the probability of the Case to which it belongs. The aggregation functions for the flow pattern, are essentially the same as those for the sequential pattern, except for the Time attribute where this is the maximum time of the parallel tasks \(\{t_1, \ldots, t_p\}\). Finally, a Loop pattern with \(k\) iterations of task \(t\) is equivalent to a sequential pattern of \(k\) copies of \(t\).

- Usually, a QoS broker [116] could discover many functionally equal web services for each of the tasks in a composite web service, although the QoS of these candidate web services for a task may be different. Therefore, when we select web services for combination to generate an execution plan for a composite service, there are a large number of possible composition plans. Different plans will lead to significantly different QoS results for the composite service. We should carefully select the composition plan from the large amount of possible solutions.

- Few customers are only concerned about one QoS criterion for the delivered composite service. A more realistic situation is that a customer would
have multiple QoS requirements for a composite service, such as minimal response time and price, maximal reputation, availability, and reliability simultaneously. However, from the perspective of the composite service provided, some of these QoS criteria are conflicting. In practice, an individual web service with high performance (i.e. a low response time, high reliability), usually has an expensive price. The use of such web service will reduce the response time or increase the reliability of the composite, although will raise the total cost of the composite. In addition, different customers have different preferences for these QoS parameters, which can be expressed in a service-level-agreement (SLA) in terms of a weighting for preferences. The composite service provider must have a strategy to cope with conflicting objectives, in order to achieve an overall acceptable QoS of the composite service.

- There are constraints on dependence and conflict between web services [106, 1]. These constraints are caused by business preferences and incompatibility of techniques. When building a travel booking web service, if we select a particular flight booking web service implementation that does not accept deposits made by MasterCard, then we must not select an implementation for the payment web service that supports MasterCard. An example of a business preference leading to the constraints between web services. Web
service selection must consider constraints on inter-service dependence and conflict in order to guarantee the correctness of the composite service.

This is a typical combinational optimisation problem. It is conjectured that the problem is NP-hard. A number of possible combinations, namely the number of web service composition plans, will increase exponentially as the number of abstract web services and the number of concrete web services for an abstract service increase. For example, for a composite web service with 30 tasks, each of which has 10 concrete web services, the total number of composition plans is $10^{30}$. Obviously, finding an optimal composition plan from so many composition plans is impractical. Therefore, scalable web service selection methods are necessary in order to find a good quality composition solution in a short time. Additionally, constraints on inter-service dependence and conflict further complicate the problem. Even finding a feasible solution could be very difficult. Thus, an additional requirement for the web service selection methods is the ability to handle these constraints. To this aim, genetic algorithms are used to solve the problem. A formal description of the problem and a review of related work are presented.

### 3.2 Problem Formulation

According to the requirements introduced in the previous section, the problem is formulated as follows:
Given

(1) a set of abstract web services (or tasks) $A$ involved in a web service composition, where $A = \{A_1, A_2, \ldots, A_n\}$ and $n$ is the total number of abstract web services in the composition;

(2) a set of implementations (or concrete web services) $C_i$, for each of the abstract web services $A_i$, where $C_i = \{c_{i1}, c_{i2}, \ldots, c_{im_i}\}$ and $m_i$ is the total number of concrete web services for abstract Web service $A_i$;

(3) the QoS values $w_{1ij}$, $w_{2ij}$, $w_{3ij}$, $w_{4ij}$, and $w_{5ij}$ for response time, price, reputation, reliability, and availability, respectively, for concrete Web service $c_{ij}$;

(4) the weights for the QoS criteria, $W_1$, $W_2$, $W_3$, $W_4$, and $W_5$, for response time, price, reputation, reliability, and availability, respectively, where $\sum_{k=1}^{5} W_k = 1$;

(5) a set of dependencies between concrete Web services $D = \{(d_{i1j1}, d_{i2j2})\}$ if abstract Web service $i_1$ uses the $j_1^{th}$ concrete Web service, then abstract Web service $i_2$ must use the $j_2^{th}$ concrete Web service}; and

(6) a set of conflicts between concrete Web services $C = \{(c_{i1j1}, c_{i2j2})\}$ if abstract Web service $i_1$ uses the $j_1^{th}$ concrete Web service, then abstract Web service $i_2$ must not use the $j_2^{th}$ concrete Web service}
where $1 \leq i \leq n$, $1 \leq j \leq m$, find $X = (x_1, x_2, \cdots, x_n)$, meaning abstract Web service $A_i$ uses concrete Web service $c_{ix_i}$, such that

$$F_{\text{obj}}(X) = \sum_{l=1}^{2} \left( \frac{Q_{l}^{\text{max}} - Q_{l}(X)}{Q_{l}^{\text{max}} - Q_{l}^{\text{min}}} * W_l \right) + \sum_{l=3}^{5} \left( \frac{Q_{l}(X) - Q_{l}^{\text{min}}}{Q_{l}^{\text{max}} - Q_{l}^{\text{min}}} * W_l \right)$$

(3.1) is maximal, subject to the constraints $D$ and $C$. Function $F_{\text{obj}}(X)$ returns the overall score of the web service selection plan $X$, in which $Q_{l}(X)$ denotes the value of the $l^{th}$ QoS criterion of the composite web service under the web service selection plan $X$, and $Q_{l}^{\text{max}}$ and $Q_{l}^{\text{min}}$ denote the maximal and minimal value of the $l^{th}$ QoS criterion respectively, $1 \leq l \leq 5$. (Function $F_{\text{obj}}(X)$ is defined following the simple additive weight method for multiple QoS criteria, which is proposed by Jaeger etc. [49] to transform multiple QoS-based objectives into a single objective, as well as to scale the single objective in the range $(0,1]$.)

### 3.3 Related Work

QoS-based web service selection problem was firstly reported by Zeng et al. [118], and soon was of great interest by many researchers. The problem, by its nature, is an optimisation problem. It has been modelled as diverse types of optimisation problem by researchers using different theories. Based on mathematics optimisation theory, Zeng et al. [119] formulated the problem as an integer programming problem. Based on combinational optimisation theory, Tao
et al. formulated the problem as a multi-dimension multi-choice 0-1 knapsack problem (MMKP) \cite{117}. Based on graph theory, Tao et al. transformed the problem into a multi-constraint optimal path (MCOP) problem \cite{117}. Despite the difference in the problem models, the work share two common features from the computational point of view. 1) They all consider the optimisation of multiple QoS criteria for a composite service. The QoS criteria considered contains response time, price, reputation, reliability and availability. The requirement of optimising these QoS criteria simultaneously introduces the challenge of how to handle multiple conflicting tasks. 2) They consider end-to-end QoS constraints, i.e. the response time and/or price of a composite web service should be less than a value given by customers, and/or the reputation, reliability and availability of the composite web service should be greater than an expected value.

According to these models, researchers put forward different QoS-based web service selection methods. Based on the mathematical programming model, Zeng et al. \cite{118} presented an integer programming (IP)-based approach for the problem. The advantage of the IP-based approach is that it is an exact algorithm therefore always optimal. However, the scalability of the IP-based approach is poor. Its computation time will increase exponentially as the number of tasks and the number of candidate web services for each task increase. Therefore, the IP-based approach is not suitable for large scale web service selection problem. Additionally, when solving the problem, this method needs an extra process to
build an IP mathematic model for the problem, which is too complex for runtime web service-based selection. Based on the multi-dimension multi-choice 0-1 knapsack model, Tao et al. [117] advocated the use of the Pisinger’s algorithm, an efficient modified dynamic programming approach for the multi-choice 0-1 knapsack problem, to address the QoS-based web service selection problem. Tao et al. [117] also presented a constrained shortest path (CSP) algorithm for the multi-constraint optimal path (MCOP) problem. Experimental results demonstrated that the Pisinger’s algorithm is a more scalable algorithm than the CSP algorithm, and the Pisinger’s algorithm is suitable for addressing large scale QoS-based web service problems.

Heuristic algorithms [10, 88] and meta-heuristic algorithms [18] have also been presented by researchers for the QoS-based web service selection problem. Rainer et al. [10] presented a heuristic algorithm and the core idea behind the algorithm is the H1_RELAX_IP heuristic that uses a backtracking algorithm on the results computed by a relaxed integer program. Lianyong et al. [88] presented another heuristic algorithm named Local Optimization and Enumeration Method (LOEM), which uses a local heuristic to filter the candidates of each task to a small number of promising ones, and then enumerates all the composition solutions on the reduced but most promising space. Ge et al. [18] presented a genetic algorithm for the web service selection problem. In Ge et al.’s genetic algorithm, the optimising of multiple QoS criteria is aggregated into a single optimisation
3.3 Related Work

objective based on the weighted sum method, and the end-to-end constraints are handled by a penalty-based method. All these heuristic and meta-heuristic algorithms present good scalability that is suitable for handling large scale web service selection problems, as well as good efficiency in finding satisfactory albeit suboptimal solutions.

Different from the work mentioned above, this research addresses the QoS-based web service selection problem with constraints on inter-service dependence and conflict. As demonstrated by Verma et al. [106], considering constraints on inter-service dependence and conflict, caused by business preferences, technical incompatibilities and many other practical factors, is essential for many real-world web service-based applications, in order to generate feasible composite web services. Unfortunately, this problem has been largely ignored by the work mentioned above. Verma et al. [106] transformed the problem of QoS-aware web service composition accommodating constraints on inter-service dependence and conflict into a constraint satisfaction problem. The authors further proposed a constrained-driven web service composition framework [106] to support the representation of this kind of constraints with semantic techniques (i.e. OWL-S [68]), and to facilitate the generation of compatible sets of web services at runtime with some reasoning techniques. However, after acquiring the compatible sets of services, how to select web services for composition accommodating constraints on inter-service dependence and conflict were not mentioned in their
A few work claimed to deal with QoS-based service selection with inter-service conflicts [43]. In this work, the authors adopted an IP-based method to handle the service conflicts constraints. However, this approach is only suitable for small size problems, that is, the number of abstract services, the number of candidate services, as well as the constraint pair number are small, due to the poor scalability of the integer programming-based approaches. Thus, it is not suitable for the environments having large number of services and various kinds of constraints.

In this thesis, we use genetic algorithms to address the QoS-based web service selection problem with constraints on inter-service dependence and conflict. A major challenge in the problem is how to handle constraints on inter-service dependence and conflict to ensure feasibility of a composite web service. As introduced in Chapter 1, genetic algorithms have different constraint handling techniques, such as penalty function-based methods, repairing methods and hybrid hybrids [26]. The above constraint handling techniques for managing constraints on inter-service dependence and conflict were used. This research resulted in three novel genetic algorithms for the web service selection problem with constraints on inter-service dependence and conflict. Each of which adopts a different constraint handling technique. The three genetic algorithms present its own highlight and weaknesses for handling the QoS-based web service selection
problem with constraints on inter-service dependence and conflict.

### 3.4 New Methods for Service Selection

Fundamentally, the problem formulated in Section 3.2 is a constrained optimisation problem. As introduced in Section 3.1, how to achieve the best QoS for a composite web service, namely the optimality issue, and how to ensure a composite web service satisfies constraints on inter-service dependence and conflict, namely the correctness issue, are two major challenges. GAs were used to address these challenges. By the nature of GAs, they can only address unconstraint optimisation problems. Fortunately, they can integrate with some constraint handling methods to handle constraints, such as penalty function methods, repairing methods and some hybrid methods, a detailed introduction of these constraint handling methods is presented in Chapter 2. In this project, different constraint handling methods to handle constraints on inter-service dependence and conflict were used and resulted in three new GAs. They are a penalty-based GA, a minimal conflicts hill climbing (MCHC) repairing GA, and a hybrid GA. The penalty-based GA uses a new penalty function to handle constraints on inter-service dependence and conflict involved in the problem. The repairing GA uses a repairing procedure called minimal conflicts hill climbing repairing to handle the constraints. So the repairing GA is also called Minimal Conflicts Hill Climbing GA. The application of minimal conflicts hill climbing is
used to accelerate the repairing process of transforming an infeasible solution into a feasible one, in order to promote search efficiency. For the hybrid GA, the core concept is a novel knowledge-based crossover operator for handling constraints. The crossover operator encompasses characteristics of cultural evolution in the form of local refinement. In the following parts of this section, the three GAs are introduced in detail.

### 3.4.1 A Penalty-Based Genetic Algorithm

As introduced in the previous section, a penalty mechanism is perhaps the most popular technique to handle constraints. A new penalty function was introduced to penalty-based GA to handle constraints on inter-service dependence and conflict. The basic idea behind the penalty mechanism is that of giving a penalty to an infeasible solution. This violates constraints on inter-service dependence and conflict to decrease its fitness value. The infeasible solution has less chance to survive in the evolution process than a feasible one.

A penalty-based GA follows the same steps of a classic GA. Algorithm 2 describes the pseudocode of the penalty GA. Like a classical GA, the major components of the penalty-based GA include representation, genetic operators, and a fitness function. Particularly, the fitness function of the penalty GA contains a penalty strategy to penalty infeasible individuals that violates constraints on inter-service dependence and conflict (see Step 2 and Step 7 in Algorithm 2).
Below is an elaboration of the components of the penalty-based GA.

**Algorithm 2:** A penalty-based genetic algorithm for the QoS-based web service selection problem

1. initialise population with random candidate solutions
2. evaluate the fitness of each individual based on the fitness function defined in Equation 3.2 which contains a penalty strategy to penalty infeasible individuals
3. while termination condition is not true do
   4. select fit individuals for reproduction
   5. probabilistically apply the crossover operator to generate offspring
   6. probabilistically apply the mutation operator to offspring
   7. evaluate the fitness of each individual based on the fitness function defined in Equation 3.2
4. end

**Representation**

The chromosome (or genome, or individual) is encoded by an integer array of $n$ elements (see the left plot of Fig 3.2), $X_1, X_2, \ldots, X_n$, and the values of $X_i$ range from 1 to $m_i$, where $n$ is the number of abstract web services in the composite web service that will be planned and $m_i$ is the number of concrete Web services for each of the abstract web services $X_i$. The right plot of Figure 3.2 is an illustrative example of the encoding method, in which the selection plan for a composite service with ten abstract web services is represented by a chromosome with ten genes. In the chromosome, $X_1 = 3$ means the first abstract service $X_1$ uses the third concrete web service, in the candidate web service set for abstract web service $X_1$, and $X_2 = 2$ means the second abstract service $X_2$ uses the second
concrete web service, in the candidate web service set for that abstract web service $X_2$, etc.

**Crossover and Mutation Operators**

The crossover operator adopted is the classical one-point crossover. A single crossover point is chosen at random and the parts of the two parents after the crossover position are exchanged to form two offspring. Mutation is performed by randomly selecting an abstract web service, namely a position in the chromosome, and replacing the current concrete web service with another. Figure 3.3 illustrates the basic idea behind the two genetic operators.

It should be observed that both the crossover operator and the mutation operator may produce infeasible individuals. Presume there are two conflicts $(C_{1,3}, C_{5,5})$ and $(C_{1,3}, C_{6,3})$. Before crossover, both parent1 and parent2 don’t violate the constraints. However, after crossover, the individual child1 violates the conflict of $(C_{1,3}, C_{5,5})$. The mutation illustrated in Figure 3.3 also leads to the violation of the constraint of $(C_{1,3}, C_{6,3})$. To handle the infeasible solution, a
penalty mechanism is introduced in the following section.

**Fitness Function**

As discussed above, some individuals generated by the crossover and mutation operators may be infeasible. Therefore, the GA must address this issue. Infeasible individuals may have some schemata that are essential to build an optimal solution. If the infeasible individuals are excluded, the GA may not be able to reach the best solution. Consequently, the strategy adopted by this GA is to allow infeasible individuals in the population, but gives a penalty to their fitness values. The following two general guidelines are used when defining the fitness function. 1) Should be guaranteed that an infeasible individual has a lower fitness value than any feasible individual. 2) An infeasible individual that violates more constraints should be more harshly penalized than an infeasible individual that violates fewer constraints. Equation 3.2 and Equation 3.3 give the definition of
the fitness function and the penalty function respectively.

\[
\text{Fitness}(X) = 0.5 + 0.5 \cdot F_{\text{obj}}(X) + p(X)
\]  

(3.2)

\[
p(X) = \begin{cases} 
0, & \text{if } V(X) = 0; \\
-0.5 - \frac{V(X)}{V_{\text{max}}}, & \text{otherwise.}
\end{cases}
\]  

(3.3)

In Equation 3.2, \( F_{\text{obj}}(X) \) is the objective function defined in Section 3 and \( p(X) \) is the penalty value given to the individual \( X \). In Equation 3.3, \( V(X) \) stands for the total number of constraint violations of \( X \), and \( V_{\text{max}} \) is the maximal number of the constraint violations. Thus, when \( V(X) \) equals zero, it implies \( X \) is a feasible individual; otherwise, \( X \) is an infeasible individual.

According to Equation 3.3, if an individual \( X \) is feasible, its penalty value is 0. Otherwise, the penalty value given to the infeasible individual \( X \) is calculated by the expression \(-0.5 - \frac{V(X)}{V_{\text{max}}},\) which can guarantee that the more constraints that an infeasible individual violates, the higher penalty it receives. According to Equation 3.2 and Equation 3.3, the fitness value of a feasible individual \( X \) can be calculated by the expression \( 0.5 + 0.5 \cdot F_{\text{obj}}(X), \) the value of which is between 0.5 and 1 (the value of the objective function \( F_{\text{obj}}(X) \) is in the range \([0, 1]\)). The fitness value of an infeasible individual \( X \) can be calculated by the expression \( 0.5 \cdot F_{\text{obj}}(X) - \frac{V(X)}{V_{\text{max}}}, \) the value of which is less than 0.5. Thus, it can be guaranteed that an infeasible individual has less fitness value than any feasible
New Methods for Service Selection

3.4.2 A Min-Conflict Hill-Climbing Repairing Based Genetic Approach

Instead of using penalty techniques for handling constraints, the repairing method is an alternative technique. The basic idea of a repairing method is to use domain-specific knowledge to correct those infeasible individuals in the population such that all individuals are always feasible. A new repairing genetic algorithm, namely the Min-Conflict Hill-Climbing (MCHC) repairing GA, to handle constraints on inter-service dependence and conflict is used. Algorithm 3 describes the pseudocode of the MCHC repairing GA. As shown in Algorithm 3, the MCHC repairing GA follows the same steps of a classic GA, except that the MCHC repairing phase (see Step 7) is introduced before the fitness evaluation phase, to repair infeasible individuals in the population. Below explains the MCHC repairing GA in detail.

Genetic Encoding and Decoding

The MCHC repairing GA adopts the same representation method (see Section 3.4.1) as the penalty-based GA introduced in the previous section.
Algorithm 3: A min-conflict hill-climbing repairing genetic algorithm for the QoS-based web service selection problem

1 initialise population with random candidate solutions
2 evaluate the fitness of each individual
3 while termination condition is not true do
4 select fit individuals for reproduction
5 apply the crossover operator to generate offspring
6 apply the mutation operator to offspring
7 repair each infeasible individual using the min-conflict hill-climbing repairing procedure
8 evaluate the fitness of each individual
9 end

Genetic Operators and Elitism

The MCHC repairing GA adopts the same genetic operators (see Section 3.4.1) as the penalty-based GA introduced in the previous section. In addition, the MCHC repairing GA uses an elitist operator that preserves the best solutions found by maintaining a group of them in the next generation.

Min-Conflict Hill-Climbing Repairing Procedure

In order to enforce feasibility, the minimise-conflict hill-climbing (MCHC) repairing procedure is used to quickly repair individuals who violate the constraints on inter-service dependencies and conflicts. The repairing procedure is conducted after the generation of new offspring by the genetic operators; crossover and mutation.

The MCHC repairing procedure uses a heuristic, called minimise-conflict [76]: selecting a variable that is in conflict and assigning it a value that minimises
the number of conflicts. In the MCHC repairing procedure, i.e., an infeasible individual, an individual that violates some constraints on inter-service dependence and conflict, is iteratively repaired using the minimise-conflict heuristic until it becomes feasible. The repairing process is like a hill-climbing process. In each step, it makes a small change to the solution, and each time the solution improves a little, with fewer conflicts. Normally both variable and value selection involve an element of randomness. Variables are chosen at random from all those that have conflicts and values are chosen at random from the set of minimise-conflict values for the selected variable. The solution might not improve after a certain number of repairs. To avoid endless repairing, use a maximal number of repairs to control the termination of the algorithm when the solution can’t be improved further. Algorithm 4 describes the pseudocode of the MCHC repairing procedure, in which \( N_{\text{fix}} \) denotes the current number of repairs and \( N_{\text{max}} \) denotes the maximal allowed number of repairs for one solution. The repairing process stops after finding a feasible solution or reaching the maximal repair number \( N_{\text{max}} \).

**Fitness Function**

Use the same fitness function for the MCHC GA as the one used by the penalty-based GA defined by Equation 3.2.
Algorithm 4: The min-conflict hill-climbing repairing procedure used by the min-conflict hill-climbing repairing genetic algorithm

input : $X$ an infeasible individual that violates constraints on inter-service dependence and conflict

output: $X'$ a repaired individual

1 $N_{fix} \leftarrow 0$
2 repeat
3 \hspace{1em} random select a gene $g$ from $X$ which violates the constraints
4 \hspace{1em} assign the gene $g$ of $X$ with the value that will cause minimal number of constraint violations
5 \hspace{1em} $N_{fix} \leftarrow N_{fix} + 1$
6 until the individual $X$ is feasible or $N_{fix} == N_{max}$
7 $X' \leftarrow X$

3.4.3 A Hybrid Genetic Algorithm

This section elaborates the hybrid genetic algorithm. This hybrid genetic algorithm utilises a knowledge-based crossover operator to handle constraints on inter-service dependence and conflict. The basic idea behind the knowledge-based crossover is that of copying all good genes in one parent individual $p_1$ that have no conflicts with each of the other genes, this is a way of exploiting domain knowledge to its child, and copying the remaining gene values from the other parent $p_2$ to the child, making the child always has less conflicts than $p_1$. The knowledge-based crossover is in essence a local improvement procedure to eliminate conflicts for handling constraints on inter-service dependence and conflict. Another feature of the hybrid genetic algorithm is the use of a local optimiser to improve the solution quality of individuals in the population. Below
introduces the hybrid genetic algorithm in detail.

**Genetic Encoding**

The hybrid GA still uses the same representation method (see Figure 8) as the penalty-based GA introduced in the previous section.

**Fitness Function**

The fitness function is the same as the one used by the penalty-based GA.

**Genetic Operators**

Unlike the crossover operator used in the penalty-based genetic algorithm and the repairing-based genetic algorithm, the crossover operator used in the hybrid genetic algorithm is a knowledge-based crossover operator. The crossover operator takes two parents, $p_1$ and $p_2$, and produces two children $c_1$ and $c_2$.

When producing $c_1$, the crossover operator firstly finds all the concrete web service selections that do not have any constraints with any other concrete web service selections in $p_1$ and copies them to $c_1$. The rest of the concrete web service selections of $c_1$ are copied from $p_2$. Similarly, when producing $c_2$, the crossover operator firstly finds all the concrete web service selections that do not have any constraints with any other concrete web service selections in $p_2$ and copies them to $c_2$. The rest of the concrete web service selections of $c_2$ are copied from $p_1$. Figure 3.4 illustrates how a child ($c_1$) is produced.
An advantage of the above crossover operator is that child $c_1$ always has less conflicts than parent $p_1$, and child $c_2$ always has less conflicts than parent $p_2$. In other words, a local improvement in terms of constraint violation times can be made by applying this crossover operator. This is because for a child individual of length equal to $m$, which was generated by copying all $n$ ($n < m$) genes from parent $p_1$ that do have any constraints (obviously, $p_1$ has $n - m$ genes that violate constraints), and the remaining $n - m$ copying from parent $p_2$, the worst case is that the number of genes that violate constraints in this child is $n - m$ (only when all the $n - m$ genes copied from $p_2$ happen to violate constraints), equal to the number of genes that violate constraints in parent $p_1$. In most cases, the number of genes that violate constraints in the child is less than $n - m$.

The mutation operator is the same with the mutation operator used in the penalty-based genetic algorithm and the repairing-based genetic algorithm (see Figure 3.3).
Local Optimiser

Given an individual solution, which may or may not be feasible, the local optimiser is used to optimise the individuals in the population. The local optimiser is used at the beginning of the genetic algorithm to improve the individuals in the initial population, which are randomly generated, and at the end of each generation to improve the individuals in the population.

The local optimiser improves the fitness value of an individual by increasing its overall QoS value and by reducing the number of constraint violations, if any, simultaneously. This is done by systematically checking all the concrete web service selections one by one to see if there exists an alternative concrete web service that gives the individual a better fitness value. If the fitness value is improved, then the current web service selection is replaced with the alternative concrete web service. According to the definition of the fitness function, when the fitness value of an individual increases either the overall QoS value increases, or the number of constraint violations, if any, decreases, or both. Thus, the local optimiser contributes to both maximising the overall QoS value and minimising the number of constraint violations of an individual.

An abstract web service may have many implementations, such as concrete web services. Although, if we check all the alternatives of a concrete web service selection it would be too time consuming. Each time we check an
alternative we need to re-calculate the new fitness value of the individual. Which is computationally expensive as the fitness function calculation involves a calculation of the response time transformed into the problem of computing the critical path of the workflow. Therefore, in order to reduce the computation time, the following strategies are adopted. If the current concrete web service selection does not violate any constraint, then we check the alternatives one by one in a decreasing order of their weighted QoS value. Once a better alternative concrete web service is found, we replace the current concrete web service selection with the alternative immediately and then move onto the next concrete web service selection. However, if the current concrete web service selection violates any constraints, we check through all the alternatives and pick up the alternative that gives the best fitness value. Doing this, we can strengthen the constraint violation repairing ability of the local optimiser.

In order to improve the computation time of the local optimiser, instead of sorting all the concrete web services for each of the abstract web services, they are sorted only once at the beginning of the genetic algorithm, before the local optimiser is invoked. Algorithm 5 is the algorithm description for the local optimiser.
Algorithm 5: Local optimiser algorithm

1. randomly generate a sequence of abstract web services, \( W_{x_1} W_{x_2} \cdots W_{x_n} \)

2. for \( x = x_1 \) to \( x_n \) do

3. if the concrete web service selection at position \( x \) does not violate any constraints then

4. for each concrete web service \( w \) of \( W_x \) do

5. calculate the fitness value of the new web service selection plan

6. if the concrete web service selection at position \( x \) does not violate any constraints then

7. update the web service selection plan;

8. end

9. end

10. replace the concrete web service selection with the best alternative

end

else

11. for each concrete web service \( w \) of \( W_x \) do

12. calculate the fitness value of the new web service selection plan

13. if the new fitness value is greater than the old fitness value then

14. replace the web service selection with the alternative and exit

15. end

16. end

end

end
Algorithm Description of the Hybrid GA

Having defined the fitness function, genetic operators, initial population generation and local optimiser, the hybrid genetic algorithm for the web service selection problem is now presented as Algorithm 6.

**Algorithm 6**: A hybrid genetic algorithm for the web service selection problem

1. randomly create an initial population of $\text{PopSize}$ individuals, $\text{Population}$
2. **foreach** concrete web service $w$ of $\text{W}_x$ **do**
   3. optimise it using the local optimiser described by Algorithm 5 and then calculate its fitness value using the fitness function
3. **end**
4. **while** termination condition is not true **do**
   5. select fit individuals for reproduction
   6. **apply the knowledge-based crossover operator to generate offspring**
   7. apply the mutation operator to offspring
   8. optimise each individual using the local optimiser described by Algorithm 5 and then calculate its fitness value using the fitness function
5. **end**

3.5 Evaluation

This section evaluates the performance of the three GAs. We conducted the following two kinds of experiments to test their performance:

1. To fully test the effectiveness and scalability of the GAs, we tested the GAs on a number of problems, with different problem sizes and different levels
of constraints on inter-service dependence and conflict.

2. A limitation of the above experiments is that the optimality of the solutions found by the three GAs is not known by only comparing the three GAs of themselves. We therefore also compared the performance of the three GAs with the Integer Programming (IP)-based method whose solutions are always optimal by the nature of the method. However, because the IP-based can only address the problems with simple workflow patterns, the sequential pattern and the parallel pattern, and because the IP-based method and the GAs use different QoS aggregation models when involving the parallel pattern, which might result in different QoS results between the IP-based method and the GAs under the same solution, we only used the test problems with the sequential pattern, which can always guarantee that the QoS results of the IP-based method and the three GAs are the same under the same solution.

The details of the experiments are introduced in the following parts of this section.

3.5.1 Experimental Setting

In order to test the performance of the three GAs and the IP-based method, they were implemented in Microsoft Visual C\textsuperscript{#} 2005. The parameter settings for the three GAs are shown in Table 3.2. All these parameters were obtained through
doing trials on randomly generated test problems. The parameters that led to the best performance in the trials were selected as the settings of the algorithms for the experiments introduced below. As shown in the table, the population size for the penalty GA, the repairing GA and the hybrid GA were 100, 100 and 30 respectively. The crossover rate was 0.90 for each of the three GA. The mutation rate was set to 0.15 in each GA, which is higher than a traditional mutation rate (i.e. 0.08). Although this led to the high fluctuations of the average fitness value during the evolution of our GAs, the high mutation rate resulted in a better performance of our GAs than the traditional mutation rate in our trials. This fact is illustrated by Figure 3.5 and Figure 3.6. Figure 3.5 shows the comparison result of the average fitness progress during the evolution of the penalty GA with a mutation of 0.15, and the penalty GA with a mutation of 0.08. As can be seen from the figure, the penalty GA with a mutation of 0.15 had a higher fluctuation of the average fitness value during the evolution of the GA. However, the penalty GA with a mutation of 0.15 also resulted in a better best fitness value than the penalty GA with a mutation of 0.08, as can be seen from Figure 3.6. The reason that a high mutation rate led to a better performance of our GAs might because the landscape of the test problems changes abruptly, where a high mutation rate has a selective advantage [47]. Besides the above settings, the termination condition of the three genetic algorithms was “no improvement on the best solution in 15 consecutive generations”. All the experiments were conducted in a desktop computer with a
Table 3.2: GA parameters setting

<table>
<thead>
<tr>
<th>Parameters</th>
<th>PGA</th>
<th>RGA</th>
<th>HGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>100</td>
<td>100</td>
<td>30</td>
</tr>
<tr>
<td>Crossover rate</td>
<td>0.90</td>
<td>0.90</td>
<td>0.90</td>
</tr>
<tr>
<td>Mutation rate</td>
<td>0.15</td>
<td>0.15</td>
<td>0.15</td>
</tr>
</tbody>
</table>

2.33 GHz Intel Core 2 Duo CPU and a 1.95 GB RAM.

3.5.2 Simulation Model

The Simulation Model for Evaluating the Performance of the Three GAs

The computation time and quality of the results produced by the GAs depend on the size and the complexity of the web service selection problem. The size of the problem is dependent on two parameters. 1) The number of abstract web services in the workflow. 2) The number of concrete web services for each of the abstract web services. The complexity of the problem largely depends on the number of constraints on inter-service dependence and conflict in the problem. Therefore, three sets of test problems were generated. To construct the test problems, a workflow was used (see Figure 3.7) with the four typical control-flow constructs, including Sequential Pattern, Conditional Pattern, Parallel Pattern and Loop Pattern, as the building block, in order to fully test the performance of the algorithms for the problems with complex control-flow structures. Below describes the three sets of test problems constructed based on the building block.
Figure 3.5: Comparison of the average fitness progresses during the evolution of the penalty GA with a mutation rate of 0.15, and the penalty GA with a mutation of 0.08
Figure 3.6: Comparison of the best fitness progresses during the evolution of the penalty GA with a mutation rate of 0.15, and the penalty GA with a mutation of 0.08
1. The first set of test problems included 10 test problems with different numbers of abstract web services. The number of abstract web services ranged from 10 to 100 with an increment of 10. This was done by concatenating a certain number of building-block composite web service workflow shown in Figure 3.7. When generating a composite web service workflow with 30 abstract web services, we concatenated three copies of the workflow shown in the figure. The number of concrete web services for each of the abstract web services was fixed to 30. The number of constraints was fixed to 60. This set of test problems was used to test how the computation time and result quality of the GAs changes with the number of abstract web services.

2. The second set of test problems also included 10 test problems. The number of concrete web services for each of the abstract web services ranged from 10 to 100 with an increment of 10. The number of abstract web services was fixed to 10 and the total number of constraints was fixed to 60. This set of test problems was designed to test how the computation time and result quality of the GAs changes with the number of concrete web services.

3. The third set of test problems consisted of 10 test problems with different numbers of constraints. The number of constraints in the test problems ranged from 50 to 500 with an increment of 50. The number of abstract
web services was fixed to 10, and the number of concrete web services for each of the abstract web services was fixed to 10. Since the number of abstract web services and the number of concrete web services were all fixed, the number of constraints reflected the constraints density. This set of test problems was used to test how the computation time and solution quality of the GAs changes with the complexity of the problem.

In all the generated test problems, the execution path was randomly selected and the number of iterations of the loops was a randomly generated value between 0 and 6. The QoS values for concrete web services were randomly generated as well. The ranges of the values for response time, cost, reputation, availability and reliability were $[1, 10]$, $[1, 10]$, $[1, 5]$, $[0.9, 1]$, and $[0.9, 1]$, respectively, and the weighting given to the QoS criteria were fixed to 0.4, 0.3, 0.1, 0.1, and 0.1, respectively.
The Simulation Model for the Comparison of the Performance of the Three GAs and the Integer Programming-based method

We constructed ten test problems with different numbers of constraints. The number of constraints in the test problems ranged from 0 to 1800 with an increment of 200. The number of abstract web services was fixed to 20, and the number of concrete web services for each of the abstract web services was fixed to 80. Since the number of abstract web services and the number of concrete web services were all fixed, the number of constraints reflected problem complexity. All the test problems only contain the sequential workflow pattern.

3.5.3 Experimental Results

The Impact of the Number of Abstract Web Services on Solution Quality and Computation Time of the Three GAs

Firstly, the three GAs were used to solve each of the test problems in the first test set. Considering the stochastic nature of the GAs, each of the experiments was repeated 30 times and then calculated the average fitness value and average computation time for each of the experiments for each of the GAs. In addition, statistical significance tests based on T-test [13] were also performed to test whether the differences between the average fitness values of the three GAs for each of the problem are statistically significance.

Table 3.3 shows the average fitness values obtained by the three GAs for the 10 test problems in the first test set, Table 3.4 shows the T-test results, indicating
whether the differences between the average fitness values of the three GAs for the 10 test problems are significant, and Figure 3.8 shows how the average computation times of the three GAs changed when the number of abstract web services changed from 10 to 100. In the figures and tables, PGA, RGA and HGA stand for the penalty-based GA, the repairing-based GA and the hybrid GA, respectively.

![Figure 3.8: The effect of the number of abstract web services on computation time](image)

It can be seen from Table 3.3 that the hybrid GA found the best average fitness value, in bold, for seven of the 10 test problems. In addition, as shown in Table 3.4, the differences between each of the seven best results and the results of the other two GAs for the same problem were statistically significant (The difference between two average values is statistically significant if the p-value generated by a T-test is less than 0.05 [13], in non-bold in Table 3.4). The repairing-based GA found the best result for two test problems. Table 3.4 demonstrates that the
<table>
<thead>
<tr>
<th>Abstract Services</th>
<th>PGA</th>
<th>Avg. Fitness</th>
<th>Dev.</th>
<th>RGA</th>
<th>Avg. Fitness</th>
<th>Dev.</th>
<th>HGA</th>
<th>Avg. Fitness</th>
<th>Dev.</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.85</td>
<td>0.859652</td>
<td></td>
<td>0</td>
<td>0.850317</td>
<td></td>
<td>0</td>
<td>0.85137</td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>0.86</td>
<td>0.858636</td>
<td></td>
<td>0</td>
<td>0.858321</td>
<td></td>
<td>0</td>
<td>0.85821</td>
<td></td>
<td>06</td>
</tr>
<tr>
<td>4</td>
<td>0.87</td>
<td>0.858647</td>
<td></td>
<td>0</td>
<td>0.858274</td>
<td></td>
<td>0</td>
<td>0.858321</td>
<td></td>
<td>08</td>
</tr>
<tr>
<td>6</td>
<td>0.87</td>
<td>0.859660</td>
<td></td>
<td>0</td>
<td>0.858321</td>
<td></td>
<td>0</td>
<td>0.85821</td>
<td></td>
<td>07</td>
</tr>
<tr>
<td>8</td>
<td>0.87</td>
<td>0.859649</td>
<td></td>
<td>0</td>
<td>0.858274</td>
<td></td>
<td>0</td>
<td>0.858321</td>
<td></td>
<td>06</td>
</tr>
<tr>
<td>10</td>
<td>0.85</td>
<td>0.856154</td>
<td></td>
<td>0</td>
<td>0.858321</td>
<td></td>
<td>0</td>
<td>0.85821</td>
<td></td>
<td>09</td>
</tr>
<tr>
<td>20</td>
<td>0.86</td>
<td>0.856949</td>
<td></td>
<td>0</td>
<td>0.858321</td>
<td></td>
<td>0</td>
<td>0.85821</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>30</td>
<td>0.86</td>
<td>0.857859</td>
<td></td>
<td>0</td>
<td>0.858321</td>
<td></td>
<td>0</td>
<td>0.85821</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>40</td>
<td>0.86</td>
<td>0.857859</td>
<td></td>
<td>0</td>
<td>0.858321</td>
<td></td>
<td>0</td>
<td>0.85821</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>50</td>
<td>0.86</td>
<td>0.857859</td>
<td></td>
<td>0</td>
<td>0.858321</td>
<td></td>
<td>0</td>
<td>0.85821</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td>60</td>
<td>0.86</td>
<td>0.857859</td>
<td></td>
<td>0</td>
<td>0.858321</td>
<td></td>
<td>0</td>
<td>0.85821</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td>70</td>
<td>0.86</td>
<td>0.857859</td>
<td></td>
<td>0</td>
<td>0.858321</td>
<td></td>
<td>0</td>
<td>0.85821</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>80</td>
<td>0.86</td>
<td>0.857859</td>
<td></td>
<td>0</td>
<td>0.858321</td>
<td></td>
<td>0</td>
<td>0.85821</td>
<td></td>
<td>06</td>
</tr>
<tr>
<td>90</td>
<td>0.86</td>
<td>0.857859</td>
<td></td>
<td>0</td>
<td>0.858321</td>
<td></td>
<td>0</td>
<td>0.85821</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>0.85</td>
<td>0.857859</td>
<td></td>
<td>0</td>
<td>0.858321</td>
<td></td>
<td>0</td>
<td>0.85821</td>
<td></td>
<td>06</td>
</tr>
</tbody>
</table>

Table 3.3: Comparison results of the PGA, the RGA and the HGA for test problems with different numbers of abstract web services.
Table 3.4: T-test results for the pair of the PGA and the RGA, the pair of the PGA and the HGA, and the pair of the RGA and the HGA, for test problems with different numbers of abstract web services

<table>
<thead>
<tr>
<th>Abstract Services</th>
<th>PGA-RGA P-Value</th>
<th>PGA-HGA P-Value</th>
<th>RGA-HGA P-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.16</td>
<td>1.76E – 21</td>
<td>6.05E – 21</td>
</tr>
<tr>
<td>20</td>
<td>4.38E – 3</td>
<td>6.24E – 05</td>
<td>0.04</td>
</tr>
<tr>
<td>30</td>
<td>0.44</td>
<td>1.77E – 04</td>
<td>4.02E – 05</td>
</tr>
<tr>
<td>40</td>
<td>0.40</td>
<td>1.53E – 03</td>
<td>9.37E – 04</td>
</tr>
<tr>
<td>50</td>
<td>0.03</td>
<td>0.01</td>
<td>2.80E – 07</td>
</tr>
<tr>
<td>60</td>
<td>0.02</td>
<td>0.01</td>
<td>4.04E – 06</td>
</tr>
<tr>
<td>70</td>
<td>0.10</td>
<td>8.01E – 04</td>
<td>0.04</td>
</tr>
<tr>
<td>80</td>
<td>0.45</td>
<td>6.33E – 05</td>
<td>2.16E – 04</td>
</tr>
<tr>
<td>90</td>
<td>0.02</td>
<td>9.80E – 15</td>
<td>4.59E – 12</td>
</tr>
<tr>
<td>100</td>
<td>1.55E – 05</td>
<td>1.71E – 18</td>
<td>3.58E – 15</td>
</tr>
</tbody>
</table>

Statistical results were significant for the two test problems. The penalty-based GA found the best result among the three GAs for only one problem. However, the difference between the best one and the result of the RGA for the problem was not statistically significant (p-value was 0.45, greater than 0.05). Overall, the hybrid GA can find better quality solutions than the other two GAs. The repairing GA can find better quality solutions than the penalty GA.

It can be seen from the figure that the computation time of the penalty GA and the repairing GA increases close to linearly when the number of abstract web services involved in a composite web service increases. The hybrid GA was the slowest and its computation time increased with a super-linear trend, but the longest time it took to solve the test problem was less than 40 seconds, which is
acceptable as the optimal web service selection is done at design time rather than at run time.

**The Impact of the Number of Candidate Web Service for Each Abstract Service on Solution Quality and Computation Time of the Three GAs**

The three GAs were tested to solve each of the test problems in the second test set. Each repeated of the experiments 30 times and then calculated the average fitness value and average computation time for each of the experiments for each of the GAs. T-tests were also performed to test whether the differences between the average fitness values of the three GAs for each of the problem are statistically significant. Table 3.5 shows the average fitness values obtained by the three genetic algorithms for the 10 test problems in the second test set, Table 3.6 shows the T-test results, and Figure 3.9 shows how the average computation times of the three GAs changed when the number of concrete web services changed from 10 to 100.

It can be seen from Table 3.5 and Table 3.6 that the hybrid GA outperformed the other two GAs as it found the best fitness value for nine of the 10 test problems and eight of the nine best results were statistically significant. The repairing GA performed slightly better than the penalty-based GA, as it found the better fitness value for two of the 10 test problems compared with the penalty GA, while for the remaining eight test problems there were no statistically significant differences between the results of the repairing GA and the penalty GA. Regarding
Table 3.5: Comparison results of the PGA, the RGA and the HGA for test problems with different number of concrete web services for each abstract web service

<table>
<thead>
<tr>
<th>Candidate Services #</th>
<th>PGA Avg. Fitness</th>
<th>PGA Dev. (%)</th>
<th>RGA Avg. Fitness</th>
<th>RGA Dev. (%)</th>
<th>HGA Avg. Fitness</th>
<th>HGA Dev. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.84133</td>
<td>0.39</td>
<td>0.84013</td>
<td>0.14</td>
<td>0.84747</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>0.87848</td>
<td>0.02</td>
<td>0.87856</td>
<td>0.01</td>
<td>0.87859</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0.88633</td>
<td>0.08</td>
<td>0.88657</td>
<td>0.03</td>
<td>0.89116</td>
<td>0.20</td>
</tr>
<tr>
<td>40</td>
<td>0.87409</td>
<td>0.24</td>
<td>0.87471</td>
<td>0.21</td>
<td>0.88117</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>0.89136</td>
<td>0.39</td>
<td>0.89266</td>
<td>0.17</td>
<td>0.89361</td>
<td>0</td>
</tr>
<tr>
<td>60</td>
<td>0.90001</td>
<td>0.26</td>
<td>0.90025</td>
<td>0.17</td>
<td>0.90131</td>
<td>0.04</td>
</tr>
<tr>
<td>70</td>
<td>0.90818</td>
<td>0.18</td>
<td>0.90829</td>
<td>0.20</td>
<td>0.90950</td>
<td>0.02</td>
</tr>
<tr>
<td>80</td>
<td>0.90812</td>
<td>0.29</td>
<td>0.90971</td>
<td>0.30</td>
<td>0.91196</td>
<td>0.01</td>
</tr>
<tr>
<td>90</td>
<td>0.91296</td>
<td>0.37</td>
<td>0.91552</td>
<td>0.35</td>
<td>0.91916</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>0.90521</td>
<td>0.29</td>
<td>0.90472</td>
<td>0.35</td>
<td>0.90981</td>
<td>0.08</td>
</tr>
</tbody>
</table>
Table 3.6: T-test results for the pair of the PGA and the RGA, the pair of the PGA and the HGA, and the pair of the RGA and the HGA, for test problems with different number of concrete web services for each abstract web service

<table>
<thead>
<tr>
<th>Candidate Services #</th>
<th>PGA-RGA P-Value</th>
<th>PGA-HGA P-Value</th>
<th>RGA-HGA P-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.38</td>
<td>2.09E − 17</td>
<td>7.76E − 21</td>
</tr>
<tr>
<td>20</td>
<td>0.12</td>
<td>0.03</td>
<td>0.14</td>
</tr>
<tr>
<td>30</td>
<td>0.07</td>
<td>5.49E − 35</td>
<td>1.52E − 37</td>
</tr>
<tr>
<td>40</td>
<td>0.10</td>
<td>1.73E − 16</td>
<td>9.84E − 20</td>
</tr>
<tr>
<td>50</td>
<td>0.12</td>
<td>1.11E − 3</td>
<td>1.55E − 3</td>
</tr>
<tr>
<td>60</td>
<td>0.37</td>
<td>0.04</td>
<td>0.01</td>
</tr>
<tr>
<td>70</td>
<td>0.29</td>
<td>2.24E − 05</td>
<td>9.77E − 05</td>
</tr>
<tr>
<td>80</td>
<td>0.03</td>
<td>2.77E − 07</td>
<td>1.03E − 05</td>
</tr>
<tr>
<td>90</td>
<td>0.04</td>
<td>1.28E − 07</td>
<td>1.95E − 06</td>
</tr>
<tr>
<td>100</td>
<td>0.37</td>
<td>6.80E − 09</td>
<td>1.70E − 12</td>
</tr>
</tbody>
</table>

Figure 3.9: The effect of the number of concrete web services per abstract web service on the computation time.

The computation time, it can be seen from the figure that the computation time of the three GAs had no correlation with the number of concrete web services per abstract web service. The penalty GA and the repairing GA took less than 0.5 seconds for all test problems. The hybrid GA took a longer time than the other
two GAs for all of the test problems in the second test set. However, the longest one was less than 2 seconds, which is still acceptable.

The Impact of Constraint Density on Solution Quality and Computation Time of the Three GAs

The three GAs were used to solve the test problems in the third test set one by one. Each of the experiments was repeated 30 times and we then calculated the average fitness value and average computation time for each of the experiments for each of the GAs. T-tests were also performed to test whether the differences between the average fitness values of the three GAs for each of the problem are statistically significance. Table 3.7 shows the average fitness values obtained by the three GAs for the 10 test problems in the second test set, Table 3.8 shows the T-test results, and Figure 3.10 displays how the average computation times of the three GAs changed when the number of constraints changed from 50 to 500.

![Figure 3.10: The effect of constraint density on the computation time](image-url)
Table 3.7: Comparison results of the PGA, the RGA and the HGA for test problems with different constraint densities

<table>
<thead>
<tr>
<th>Constraint Pair</th>
<th>PGA</th>
<th>RGA</th>
<th>HGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>0.2920</td>
<td>0.2915</td>
<td>0.2920</td>
</tr>
<tr>
<td>50</td>
<td>0.8299</td>
<td>0.8299</td>
<td>0.8299</td>
</tr>
<tr>
<td>100</td>
<td>0.8309</td>
<td>0.8309</td>
<td>0.8309</td>
</tr>
<tr>
<td>150</td>
<td>0.8319</td>
<td>0.8319</td>
<td>0.8319</td>
</tr>
<tr>
<td>200</td>
<td>0.8329</td>
<td>0.8329</td>
<td>0.8329</td>
</tr>
<tr>
<td>250</td>
<td>0.8339</td>
<td>0.8339</td>
<td>0.8339</td>
</tr>
<tr>
<td>300</td>
<td>0.8349</td>
<td>0.8349</td>
<td>0.8349</td>
</tr>
<tr>
<td>350</td>
<td>0.8359</td>
<td>0.8359</td>
<td>0.8359</td>
</tr>
<tr>
<td>400</td>
<td>0.8369</td>
<td>0.8369</td>
<td>0.8369</td>
</tr>
<tr>
<td>450</td>
<td>0.8379</td>
<td>0.8379</td>
<td>0.8379</td>
</tr>
<tr>
<td>500</td>
<td>0.8389</td>
<td>0.8389</td>
<td>0.8389</td>
</tr>
</tbody>
</table>

Table 3.7: Comparison results of the PGA, the RGA and the HGA for test problems with different constraint densities.
Table 3.8: T-test results for the pair of the PGA and the RGA, the pair of the PGA and the HGA, and the pair of the RGA and the HGA, for test problems with different constraint densities.

<table>
<thead>
<tr>
<th>Constraint Pair #</th>
<th>PGA-RGA P-Value</th>
<th>PGA-HGA P-Value</th>
<th>RGA-HGA P-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.36</td>
<td>3.45E−10</td>
<td>1.64E−13</td>
</tr>
<tr>
<td>100</td>
<td>0.49</td>
<td>2.21E−04</td>
<td>3.32E−10</td>
</tr>
<tr>
<td>150</td>
<td>5.18E−03</td>
<td>7.32E−07</td>
<td>2.02E−12</td>
</tr>
<tr>
<td>200</td>
<td>1.31E−05</td>
<td>1.44E−06</td>
<td>7.18E−10</td>
</tr>
<tr>
<td>250</td>
<td>9.53E−08</td>
<td>5.31E−08</td>
<td>2.15E−14</td>
</tr>
<tr>
<td>300</td>
<td>0.01</td>
<td>0.01</td>
<td>5.78E−10</td>
</tr>
<tr>
<td>350</td>
<td>2.76E−13</td>
<td>9.21E−12</td>
<td>7.01E−15</td>
</tr>
<tr>
<td>400</td>
<td>1.17E−07</td>
<td>2.45E−07</td>
<td>6.41E−08</td>
</tr>
<tr>
<td>450</td>
<td>0.06</td>
<td>1.09E−12</td>
<td>0.25</td>
</tr>
<tr>
<td>500</td>
<td>0.10</td>
<td>0.41</td>
<td>0.35</td>
</tr>
</tbody>
</table>

It can be seen from Table 3.7 that for problems with 50 to 250 constraints, the penalty-based GA can always find feasible solutions. However, for problems with a higher constraint density, it can’t always find feasible solutions. This is reflected by the deviation of the fitness values. For the problem with 300 constraint pairs, the deviation of the fitness values is 17.26%. This is because as the constraint density becomes higher, it is more difficult to find a feasible solution, or there might be no feasible solution at all. It can also be seen from the table that the repairing GA and the hybrid GA failed to find a feasible solution when the number of constraint pairs increased to 450. In addition, the overall fitness value of the hybrid GA was better than the repairing GA and the results were reliable in terms of statistical significance for most of the test problems, as can be seen...
from Table 3.7 and Table 3.8. So, the hybrid GA still performed best among the three GAs, in terms of solution quality. The repairing GA performed better than the penalty-based GA.

It can be seen from the figure that basically the computation time of the three GAs increased linearly with the number of constraints. The hybrid GA took a longer time than the other two GAs for all of the test problems in the third test set. However, the longest one was only less than 22 seconds, which is still quick enough.

**Comparison of the performance of the Three GAs and the Integer Programming-based method**

The IP-based method and the three GAs were tested to solve each of the test problems in the simulation model for the comparison of the GAs and the IP-based method. Each GA repeated of the experiments 30 times, and then obtained the best and worst fitness values, and also calculated the average fitness value and average computation time for each of the experiments for each of the GAs. For the IP-based method, we performed each test problem only once, because the execution times and the solutions found are fixed for each test problem. The comparison results of the computation times and the solutions are shown in Figure 3.11 and Table 3.9, respectively.

It can be seen from the figure that the computation time of the three GAs increased slowly as the number of constraints increases. The GAs spent less than
60 seconds for all test problems. It can also be seen that the computation time of the IP-based method increased dramatically from 0.07 seconds to 2134 seconds as the number of constraints increases from 0 to 800. As the number of constraints becomes larger, the IP-based method failed to find a solution.

It can be seen from the table that for the problems with 0 to 800 constraints, the best solutions found by the three GAs were the same with the solutions found by the IP-based method, the optimal solutions. For the problems with more than 800 constraints, we don’t know the fitness value of the optimal solution because the IP-based method could not find a solution in a reasonable time. But we know that the fitness value of the optimal solution was less than 0.9926 (the fitness value of the optimal solution for the problem with 800 constraints), as more constraints were introduced in the same test instance. From the table we can see that, for the problems with 1000 to 1400 constraints, even the worst results of the PGA were very closed to the optimal solutions. The PGA failed to find a feasible solution when the number of constraints was greater than 1600. For the HGA and the RGA, they can always find a feasible solution for all test problems, and the results found by the two GAs were always very closed to the optimal solutions.

Based on the above experimental results for the four sets of test problems, the following conclusions can be drawn:
Table 3.9: Comparison results of the integer programming-based method, the three GAs for problems of different complexities.

<table>
<thead>
<tr>
<th>Constraints</th>
<th>IP</th>
<th>PGA</th>
<th>RGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.9963</td>
<td>0.998</td>
<td>1</td>
</tr>
<tr>
<td>200</td>
<td>0.9889</td>
<td>0.996</td>
<td>1</td>
</tr>
<tr>
<td>400</td>
<td>0.9926</td>
<td>0.9926</td>
<td>0.9814</td>
</tr>
<tr>
<td>600</td>
<td>0.9926</td>
<td>0.9926</td>
<td>0.9777</td>
</tr>
<tr>
<td>800</td>
<td>0.9926</td>
<td>0.9926</td>
<td>0.9740</td>
</tr>
<tr>
<td>1000</td>
<td>0.9814</td>
<td>0.9740</td>
<td>0.973</td>
</tr>
<tr>
<td>1200</td>
<td>-</td>
<td>0.9814</td>
<td>0.9665</td>
</tr>
<tr>
<td>1400</td>
<td>-</td>
<td>0.9703</td>
<td>0.9480</td>
</tr>
<tr>
<td>1600</td>
<td>-</td>
<td>0.9591</td>
<td>&lt;0.5</td>
</tr>
<tr>
<td>1800</td>
<td>-</td>
<td>0.9591</td>
<td>&lt;0.5</td>
</tr>
</tbody>
</table>

| Best Fitness | Worst Fitness | Average Fitness | | |
|-------------|---------------|----------------|---|
| HGA         | PGA           | IP             | # |
| 0.9963      | 0.966          | 0.966         | 0 |
| 0.9889      | 0.9814         | 0.9814        | 0 |
| 0.9963      | 0.9814         | 0.9814        | 0 |
| 0.9963      | 0.966          | 0.966         | 0 |
| 0.9963      | 0.966          | 0.966         | 0 |
| 0.9963      | 0.966          | 0.966         | 0 |
| 0.9963      | 0.966          | 0.966         | 0 |
| 0.9963      | 0.966          | 0.966         | 0 |
| 0.9963      | 0.966          | 0.966         | 0 |

Table 3.9: Comparison results of the integer programming-based method, the three GAs for problems of different complexities.
Scalability  (1) All the three GAs are scalable. The computation time of the penalty GA and MCHC GA increases linearly when the number of abstract web services increases. It does not change significantly when the number of concrete web services increases or when the number of constraints increases. As the slowest algorithm of the three GAs, the hybrid GA takes less than 40 seconds for all the test problems, which is acceptable for design time web service-selection problems. In addition, among the three GAs, the penalty GA is the fastest. This is because the penalty GA does not include the time costly repairing or local optimisation procedures.  (2) The scalability of the IP-based method is poor. Its computation time increases quickly as the problem becomes complex. For many complex problems where the constraint densities are high, it could not find a solution in a reasonable time.
Performance  (1) All the three GAs are effective. They can find the optimal solutions or the suboptimal solutions for most complex problems, where the constraint densities are high. (2) The hybrid GA outperforms the two other GAs in terms of solution quality, as for most of the test cases the hybrid GA can find a better result than the others two. This is mainly because of the application of a local optimiser in the hybrid GA, which exploits QoS-related information, while the two others don’t have this ability. In addition, for most of the test cases, the hybrid GA can find a feasible solution even though the constraint density is high. Only for a few cases, it failed and this may be because there is no feasible solution. The MCHC GA performs better than the penalty-based GA from the perspective of solution quality. Additionally, it handles constraints on inter-service dependence and conflict, benefiting from the use of the MCHC repairing operator to exploit local information to eliminate constraints.

3.6 Verification

Experiments have also been conducted to verify the correctness of the QoS results of the composite services generated by the GAs via simulation. Measuring the actual QoS values of the composite services, and then comparing them with the simulation results. Monitoring the actual execution times for a test composite service under different execution plans, as the execution time of a composite service is very simple to track, was conducted. Then, the actual values with the
simulation values were compared. The actual execution times are expected to be consistent with the simulation execution times.

**Experimental Setup**  The experimental environment contained the following components:

- **A composite web service** — A travel planning composite web service was developed in Microsoft BizTalk and deployed on a BizTalk server (2 Duo 2.33GHz CPU, a 1.95 GB RAM).

- **QoS-based web service selection tool** — This tool (see Figure 3.12) was developed in Microsoft Visual C\(^\text{\#}\) 2005. The new GAs are used by the tool for making the web service selection decision according to a customer’s QoS preferences.

- **A set of concrete web services** — We have developed 10 candidate web services for each task involved in the travel planning composite web service. These candidate web services have different execution time and price. They were developed in Microsoft Visual C\(^\text{\#}\) 2005 and deployed on a web server (2 Duo 2.33GHz CPU, a 1.95 GB RAM).

- **A web-based client application** — The client application was developed in Microsoft ASP.NET, which is responsible for sending requests for invoking the travel planning composite web service.
Results  Two extreme problems were tested. Optimising the execution time for the travel planning composite service and optimising the cost for the travel planning composite service. It should be noted that the real execution time contains both the sum of execution times of each task, and the composite service instance startup time. However, the startup time is not counted by the GAs into the simulation execution. Therefore, there should be a fixed gap between the real execution time and the simulation execution time, which is the startup time.

For the first problem, the simulation execution time of the composite service under the execution plan found by the QoS-based web service selection tool
was 2.7 seconds, and the actual execution time was 4.8 seconds. There was an approximately 2.1 seconds gap between the simulation value and the actual value. For the second problem, the simulation execution time was 18.5 seconds, and the actual execution time of the composite service under the execution plan found by our GAs was 22 seconds. The gap is also around 2 seconds. From the two groups of results, it was concluded that the real execution time of the composite service is consistent with the simulation value of the proposed GAs.

### 3.7 Summary

This chapter investigated the QoS-based web service selection problem with constraints on inter-service dependence and conflict. The major contributions of this research are the three new GAs for the problem. The three GAs use different strategies for handling constraints on inter-service dependence and conflict. The penalty-based GA uses a penalty function for handling the constraints. The min-conflict hill-climbing GA (MCHC GA) uses a fast repairing method for handling the constraints. The hybrid GA uses a novel knowledge-based crossover operator to solve the constraint problem, as well as a local optimiser to enhance solution quality. The three GAs were compared through intensive evaluation on a number of problems. We also compared the performance of the three GAs and the IP-based method. Experimental results demonstrated the good scalability and effectiveness of the three GAs, and also demonstrated that the scalability of the
three GAs are better than the IP-based method. Verification experiments were also conducted to verify the correctness of the solutions found by the GAs. These three GAs presents its own highlight for handling the QoS-based web service selection problems. 1) The penalty-based genetic algorithm is suitable for addressing large-scale QoS-based web service selection problems without overly strong constraints on inter-service dependence and conflict. 2) The MCHC GA is suitable for the QoS-based web service selection problems with very strong constraints on inter-service dependence and conflict, thanks to the fast mini-conflict hill-climbing repairing procedure. 3) The hybrid GA is preferred when customers attach a great value to solution quality, as the hybrid GA outperforms the other two GAs in terms of solution quality.
Chapter 4

QoS-Based Resource Allocation and Scheduling for Multiple Composite Web Services on Hybrid Clouds

An important application area for web service technology is cloud computing. Shared resources, software, and information can be deployed on cloud platforms as virtualised resources. These virtualised resources are provided in the form of web services which can be composed to build complex cloud service-based applications. This new service delivery mechanism is becoming popular as it brings many benefits, e.g. reducing costs and enhancing scalability and reliability.

QoS is very important for all kinds of web service-based applications. This is no exception for cloud service-based applications. A central problem that must be addressed to guarantee the QoS for cloud-based applications, is the QoS-based cloud resource allocation and scheduling problem for multiple composite web services. This is a challenging issue, especially so in a hybrid cloud where there may be some cost-free resources available from private clouds, but some
fee-paying resources from public clouds. Meeting this challenge involves two classical computational problems. One is assigning resources to each of the tasks in the composite web service. The other is scheduling the allocated resources when each resource may be used by more than one task, and may be needed at different points in time. This chapter investigates QoS-based resource allocation and scheduling for multiple composite web services on hybrid clouds.

4.1 Introduction

The previous chapter introduces the QoS-based web service selection problem. In part, the problem can be considered a resource allocation problem, where the candidate web services for each task in a composite service can be viewed as resources, and the problem is to allocate a candidate web service to each of the tasks in a composite service. For multiple web service-based applications, considering only the resource allocation problem for composite services is sufficient for guaranteeing QoS. However, there are also some specialised web service-based applications, such as grid-based web service applications and cloud-based web service applications [42], for which considering only the resource allocation problem is inadequate to guarantee their QoS. A grid-based web service application is built based on the grid computing paradigm and web service technologies. A cloud-based web service application is built based on a cloud computing paradigm and web service technologies. For both kinds of web service-
based applications, resources can be used by multiple tasks of composite web services. This requires us to consider not only the resource allocation problem, but also the resource scheduling problem, that is, assigning resources to competing tasks in different time periods to avoid conflicts. This chapter concentrates on the cloud-based resource allocation and scheduling problem. This is a more complex problem in the sense of resource allocation strategies, than the problem in the context of grid computing. The methods for cloud-based problems can be easily extended to solve grid-based problems.

Cloud computing [6] is a new Internet-based computing paradigm driven by economies of scale, whereby a pool of computation resources deployed as web services, are provided on demand over the Internet, in the same manner as public utilities. Recently, cloud computing has become popular because it brings many benefits for enterprises to build their own web service-based applications: 1) This paradigm provides an economical way to build various services. Instead of owning all the required computational resources, a company can rent required resources from a public cloud data centre and pay only for what they need. In this way, capital expenditure on hardware, software, and services, as well as ownership and maintenance costs, can be reduced or avoided. 2) In addition, the application of public cloud services for building complex services can greatly enhance the services’ scalability, because the large number of computing resources available from public clouds can be employed on demand.
When an enterprise builds a web service-based application based on the cloud computing paradigm, it uses web services provided by cloud providers. With regard to cloud providers, this research distinguishes public clouds and private clouds. When a cloud is made available in a pay-as-you-go manner to the public, we call it a public cloud. Examples of public cloud service providers include Amazon web services, Google AppEngine, and Microsoft Azure [6]. Private clouds refer to internal data centres of a business or other organisations that are not made available to the public. Presently, the categorisation of clouds into private clouds and public clouds is still being debated. Some researchers argue that private clouds don’t belong to cloud computing, because their use does not benefit from lower up-front capital costs and less hands-on management [109]. Nevertheless we think it is necessary to consider private clouds in the allocation and scheduling problem because their use is still preferred in many scenarios. For example, large enterprises have often already had substantial investments in the infrastructure required to provide resources in-house. Furthermore, many organisations would prefer to keep sensitive data under their own control to ensure security.

In general, a composite web service built by an enterprise is composed of two or more component web services. These component web services can be provided by the private cloud owned by the enterprise, a number of public clouds, or both private and public clouds together. Such a computing environment is called a
Component web services provided by the private cloud and the public clouds have the same functionality, but may have different Quality-of-Service (QoS) values, such as response times. In addition, in the private cloud a component web service may have a limited number of instances, each of which may have different QoS values. In public clouds, a component web service may have a large number of instances. The QoS values of the instances of the same component web service in the same public cloud are assumed to be identical. However, the QoS values of instances in different public clouds may vary.

There may be many composite web services in an enterprise. Each of the component web services in a composite web service needs to be allocated an
instance of the component web service. A single instance of a component web service may be allocated to more than one component web service task in the composite web service, as long as it is used at different times. This is the purported component web service allocation problem.

In addition, in order to maximise the utilisation of the component web services in the private cloud, and minimise the cost of using component web services in public clouds, allocated component web service instances can only be used for a fixed period of time. This involves scheduling the allocated component web service instances. This is the purported component web service scheduling problem.

There are two typical QoS-oriented component web service allocation and scheduling problems in the hybrid cloud. A possible solution to the first problem, called the deadline constraint problem, is to find a resource allocation and scheduling plan which can minimise the total cost of the composite web service, satisfying given response time constraints for each composite web service. The key to the second problem, called the cost constraint problem, is to find a resource allocation and scheduling plan which minimises the total response times of each composite web service, satisfying a total cost constraint. These two component web service allocation and scheduling problems can be categorised as variants of traditional resource allocation and scheduling problems.

Fundamentally, the problem we address is a resource allocation and
scheduling problem. An example of this type of problem is of mapping tasks on distributed services, a well known NP-hard problem [114]. It is surmised that the computational complexity of the problem we address is NP-hard. To address it in the context of hybrid cloud services, we present a new random-key genetic algorithm and a cooperative coevolutionary genetic algorithm for the problem.

4.2 Problem Description

Based on the requirements introduced in Section 4.1, the problem can be formulated as follows:
Inputs

1. A set of composite services $W = \{W_1, W_2, \ldots, W_n\}$ modelled by directed acyclic graphs (DAGs), where $n$ is the number of composite services. Each composite service consists of several abstract web services. We define $O_i = \{o_{i,1}, o_{i,2}, \ldots, o_{i,n_i}\}$ as the abstract web services set for composite service $W_i$, where $n_i$ is the number of abstract web services contained in composite service $W_i$.

2. A set of candidate cloud services $S_{i,j}$ for each abstract web service $o_{i,j}$, $S_{i,j} = S^u_{i,j} \cup S^v_{i,j}$, where $S^v_{i,j} = (S^v_{i,j,1}, S^v_{i,j,2}, \ldots, S^v_{i,j,\ell})$ denotes an entire set of $\ell$ private cloud service candidates for abstract web service $o_{i,j}$, $S^u_{i,j} = (S^u_{i,j,1}, S^u_{i,j,2}, \ldots, S^u_{i,j,m})$ denotes an entire set of $m$ public cloud service candidates for abstract web service $o_{i,j}$.

3. A response time and price for each public cloud service $S^u_{i,j,k}$, denoted by $t^u_{i,j,k}$ and $c^u_{i,j,k}$ respectively, and a response time and price for each private cloud service $S^v_{i,j,k}$, denoted by $t^v_{i,j,k}$ and $c^v_{i,j,k}$ respectively.

Deadline Constraint Problem

Output An allocation and scheduling plan $X = \{X_i \mid i = 1, 2, \ldots, n\}$, such that the total cost $\text{Cost}(X)$ is minimal. Let $X_i$ denote an allocation and scheduling plan for composite service $W_i$, such that $X_i =$
\[ ((M_{i,1}, F_{i,1}), (M_{i,2}, F_{i,2}), \ldots, (M_{i,n_i}, F_{i,n_i})) \]

where \( M_{i,j} \) represents the selected cloud service for abstract web service \( o_{i,j} \) and \( F_{i,j} \) represents the finishing time of an instance of abstract web service \( o_{i,j} \). Equation 4.1 gives the definition of the total cost for a composite service.

\[
Cost(X) = \sum_{i=1}^{n} \sum_{j=1}^{n_i} Cost(M_{i,j}) \tag{4.1}
\]

**Constraints** Provision of web services is subject to both precedence and resourcing constraints, and all composite services are subject to deadline constraints.

\[
F_{i,k} \leq F_{i,j} - d_{i,j}, \text{ where } j = 1, \ldots, n_i \text{ and } k \in Pre_{i,j} \tag{4.2}
\]

\[
\sum_{j \in A(t)} r_{j,m} \leq 1, \text{ where } m \in S_{i,j}^v \tag{4.3}
\]

\[
F_{i,n_i} \leq d_i, \text{ where } i = 1, \ldots, n \tag{4.4}
\]

In Equation 4.2, set \( Pre_{i,j} \) denotes all abstract web services that must execute before the abstract web service \( o_{i,j} \). Equation 4.2 imposes the precedence relationships between abstract web services.

In Equation 4.3, set \( A(t) \) denotes the abstract web services being used at time \( t \).

We let \( r_{j,m} = 1 \) if abstract web service \( j \) requires private cloud service \( m \) to be
processed and \( r_{j,m} = 0 \) otherwise. Equation 4.3 states that each private cloud service can only process one abstract web service at a time.

In Equation 4.4, \( d_i \) denotes the deadline promised to the customer for composite service \( W_i \). \( F_{i,n_i} \) is the finishing time of the last abstract service of the whole composite service \( W_i \), that is, the execution time of the composite service \( W_i \). Equation 4.4 imposes the deadline constraint for each composite service.

Cost Constraint Problem

Output  An allocation and scheduling plan \( X = \{X_i \mid i = 1, 2, \ldots, n\} \), such that the total response time \( Time(X) \) is minimal.

\[
Time(X) = \sum_{i=1}^{n} (F_{i,n_i})
\]  (4.5)

Here, Equations 4.5 gives the definition of total response times for all composite services.

Constraints  Provision of web services is subject to both precedence (defined by Equation 4.2) and resourcing (defined by Equation 4.3) constraints, and each composite web service is subject to a cost constraint (defined by Equation 4.6, in which \( Cost_i(X) \) denotes the execution cost of composite service \( W_i \), and \( c_i \) denotes the execution cost promised to the customer for composite service \( W_i \).)
\[ \text{Cost}_i(X) \leq c_i, \text{ where } i = 1, \ldots, n \] (4.6)

4.3 Related Work

Cloud computing is a new computing paradigm. It has been conceived as a potential paradigm to transform a large part of the IT industry, making software even more attractive as a service, and shaping the way IT hardware is designed and purchased [6]. Buyya et al. [16] defined this kind of computational resources provision paradigm, based on cloud computing, as market-oriented cloud computing. The authors also presented state-of-the-art cloud technologies and identified research gaps that must be filled to realise market-oriented cloud computing. One such gap to be filled is to find mechanisms and algorithms for allocation of cloud resources to meet Service Level Agreements (SLAs). There are a few initial pieces of work towards this aim. Sotomayor et al. [95] emphasised designing a virtual infrastructure manager and a resource lease manager to effectively manage and schedule visualised services on a local pool of resources, namely private clouds, as well as on external clouds, namely public clouds. The virtual infrastructure manager and resource lease manager facilitate flexible resource provision models on clouds. This includes immediate resource provision, best-effort resource provision and advance reservation, which are beneficial to the effective management of cloud resources. Deelman et al. [32] studied the cost
of running a workflow on a cloud. This work revealed that different execution plans of the same application may result in significantly different costs. The computational resources involved in their study include computing, storage and communication resources.

The work in this chapter can be categorised into an active research area called workflow-based scheduling. To date, many heuristic and meta-heuristic based scheduling algorithms have been proposed for scheduling workflow applications in grid-based environments. These algorithms can be classified into two types from the perspective of the objective function. One type of scheduling algorithms optimises the execution time of workflow applications. Well-known heuristics algorithms of this type include Min-min, Max-min, Sufferage, and HEFT [65]. Meta algorithms include genetic algorithms [87], and simulated annealing [73]. With the emergence of service-oriented workflows, the other type of scheduling algorithms, with the objective function of optimising QoS properties such as response time and cost, has become a big concern. Scheduling problems with QoS-related objectives are more complex than problems with only a time-related objective, because the former often requires optimising one QoS property, and merely satisfying constraints on other QoS properties. Existing heuristics and meta-heuristics for problems with only a time-related objective will not work well for QoS-related scheduling problems. Sakellariou et al. [91] proposed two new heuristics called LOSS and GAIN to optimise execution time of
workflow applications while satisfying cost constraints. Yu et al. in their serial papers [114, 115] studied the application of genetic algorithms to solve problems with different objectives. These objectives included optimising response times and satisfying cost constraints, optimising costs and satisfying time constraints, and optimising multiple objectives, containing both response times and costs.

A common feature of the above QoS-based scheduling algorithms is that they only consider scheduling for a single workflow. This is unrealistic, assuming that no other workflows will compete for resources with the workflow being scheduled. In this thesis, we considered how to schedule for multiple workflow applications. This is a more practical consideration in the real world, though bringing further complex problems, because of the larger solution space and more complex constraints involved. Existing single workflow scheduling algorithms cannot guarantee to find a feasible solution for the problem, because they ignore competition for resources between workflows. The multi-workflow scheduling problem bears much in common with multi-project scheduling problems that have been intensively studied in the field of manufacturing planning and scheduling. Algorithms for the multi-project scheduling problems include extract methods, such as: - the zero-one programming approach [86] - the hybrid branch and bound / dynamic programming algorithm [36] - heuristic algorithms, such as the minimum slack first heuristic [39] and the maximum total work content heuristic [59] - meta-heuristic algorithms, such as a genetic algorithm [46]. This
prior research motivated the direction of this thesis to study the multi-project problem for workflow applications, namely workflow in the context of web services. Here, we study the problem with a QoS-related objective function and constraints. This is different from the work conducted in the field of manufacturing planning and scheduling, where a time-related objective is the primary concern.

Another feature of our work is that we focus on scheduling resources in a hybrid cloud, where there are both constrained resources (private cloud services) and unconstrained resources (public cloud services). To the best of our knowledge, no previous research on workflow-based scheduling considers this situation. However, it is important for make-to-order companies, to allow the company to effectively use cloud services to save their expenditure.

4.4 New Genetic Algorithms

Two genetic algorithms (GA) were developed to solve the problem formulated in the previous section. The first GA is called the random-key GA which uses a random-key method [8] for its chromosome representation. When addressing sequencing problems such as this, GAs typically have difficulty maintaining individuals’ feasibility from parent to offspring, due to the existence of precedence constraints. A random-key genetic algorithm is one that uses a random-key representation to overcome this difficulty. The basic idea behind the random-
key representation is that of encoding a solution with a group of random values, where these values are used as sort keys in the decoding phase to deduce the solution. For each given group of random numbers, namely an individual in the context of the GA, it can always be interpreted as a feasible solution using a certain decoding procedure. Furthermore, each offspring bred by conducting crossover and mutation operators on parents, is in essence another group of random numbers that can also be decoded as a feasible solution. Therefore, an individual’s feasibility can always be met in both the initial phase and and evolution phase of our GA.

The second GA is called the cooperative coevolutionary GA. The characteristic feature of this GA is the use of a cooperative coevolution model in the algorithm in order to cope with the ever-increasing complexity of the problem. In the following parts of this section, the two GAs are introduced in detail.

### 4.4.1 The Random-Key Genetic Algorithm

**Algorithm 7:** The random-key genetic algorithm

```plaintext
1 initialise population with random candidate solutions
2 evaluate the fitness of each individual
3 while termination condition is not true do
4   select fit individuals for reproduction
5   probabilistically apply the crossover operator to generate offspring
6   probabilistically apply the mutation operator to offspring
7   decode each individual using the decoding algorithm (see Algorithm 8) and then evaluate its fitness
8 end
```
Genetic Encoding

Using the random-key encoding method, we encode each individual (or chromosome) as a list of $N$ real numbers, where $N$ is the number of abstract web services involved in all composite web services. Each gene in the chromosome corresponds to an abstract web service. The value of a gene consists of two parts: 1) an integral part, indicating which service is assigned to the abstract web service, among a set of candidate web services that could be used by the abstract web service, and 2) a fraction generated randomly in the range $[0, 1)$, determining the order of the web service in the schedule.

The sort key of a gene represents its corresponding abstract web service’s priority to occupy a cloud service, when there are multiple abstract web services competing for the same cloud service. Here, the abstract web service with the lowest fractional value has the highest priority to occupy the cloud service among all competing services. Given a sort key for each gene of a chromosome, a unique scheduling plan can be generated using the decoding algorithm introduced in Section 8. Hence, the random-key of a chromosome is, in essence, an indirect representation of a scheduling solution, providing an efficient way of representing potential schedules.

Consider the following example of the random-key method. As shown in Figure 4.2, there are two composite services, one of which has five abstract web
services $A_{1,1}$ to $A_{1,5}$, and the other which has four abstract web services $A_{2,1}$ to $A_{2,4}$. Sets $T_1, \ldots, T_5$ represent the service functions that must be fulfilled by the nine abstract web services. Abstract web services using the same functionality are in conflict if they use the same cloud service at the same time. Potential conflict sets in this example are $(A_{1,1}, A_{2,1})$, $(A_{1,2}, A_{1,4}, A_{2,2})$, and $(A_{1,3}, A_{2,3})$.

Based on the method we use, an individual in this example can be encoded as a real number list with nine, the number of abstract web services, elements. In the sample chromosome, the value of gene $A_{1,1}$ is 2.45. The integer part of this means that abstract web service $A_{1,1}$ uses the cloud service with index 2 in the set of abstract web service $A_{1,1}$’s candidate cloud services. The fractional part means that the priority of abstract web service $A_{1,1}$ to use the cloud service is 0.45. Abstract web service $A_{2,1}$ has the same functionality as abstract web service $A_{1,1}$ but has gene value 2.71, which means it also uses cloud service 2 and its priority is 0.71. According to this scenario, abstract web service $A_{1,1}$ will use the cloud service first when both want to use the cloud service at the same time.

**Decoding**

Decoding is the procedure of generating an allocation and scheduling solution based on each chromosome, so as to evaluate the fitness of the chromosome. Priorities in the chromosome need to be employed to deduce a scheduling plan. Generated schedules can be classified into semi-active, active, non-delay, and
hybrid schedules. These are based on different heuristic strategies, such as as-early-as-possible sequencing of abstract web services, and no resource idle sequencing of abstract web services [11]. In this algorithm, we adopt the algorithm introduced by Bierwirth et al. [11] to generate hybrid schedulers. Bierwirth et al. report that the procedure for generating hybrid schedules can improve the solution quality and reduce the computation time simultaneously.

In addition, we modify the algorithm to sequence the abstract web services which use public cloud services. This variation is imperative because existing algorithms cannot cope with public cloud services, which have no well-defined bounds, unlike limited resource private cloud services. A common assumption
made by existing algorithms is that a service-resource can only be used by one abstract web service at any time. For a public cloud service however, this is not the case, because a ‘single’ service can be used by multiple abstract web services at the same time thanks to the multi-tenant technology.

Algorithm 8 describes the pseudocode of the decoding algorithm we use. A major difference between this algorithm and existing ones is the introduction of an extra step (see Step 3), which processes abstract web services consuming public cloud services. The basic idea behind the algorithm is that we can schedule abstract services which use public clouds in the set of abstract services which are ready to be processed (namely the abstract services in $A$). This is because scheduling public abstract services will not delay other abstract services. After these abstract services have been scheduled, the remaining abstract services in $A$ can be scheduled in the usual way.

**Genetic Operators**

Uniform crossover is adopted in this algorithm. To generate offspring, two parents swap their genes with a fixed probability, 0.5. Figure 4.3 illustrates the uniform crossover.

The purpose of the mutation operator is to improve the diversity of genes and therefore to prevent premature convergence. Here, the mutation is used in a broader sense than usual. We introduce some new individuals into the next
Algorithm 8: The decoding procedure

**Data:** $o_{ij}$ the $j$th abstract service of composite service $i$
**Data:** $\sigma_{ij}$ the earliest starting time of abstract service $o_{ij}$
**Data:** $p_{ij}$ the processing time of abstract service $o_{ij}$
**Data:** $\tau_{ij}$ the earliest completion time of abstract service $o_{ij}$
**Data:** $\theta$ a tuneable parameter in the range $[0, 1]$ used to produce hybrid schedules. (The detailed meaning of $\theta$ is described by Bierwirth et al. [11].)

1. construct the set of all abstract services that are ready to be scheduled, $A := \{o_{ij} \mid 1 \leq i \leq N\}$

2. repeat
3.  if exist abstract services in $A$ using public cloud services then
4.      randomly select an abstract service $o^{*}_{ij}$ from $A$ which uses a public cloud service
5.      append abstract service $o^{*}_{ij}$ to the schedule and calculate its starting and completion time
6.      if a successor $o^{*}_{ij+1}$ of the abstract service $o^{*}_{ij}$ exists then
7.         insert the successor $o^{*}_{ij+1}$ into $A$, i.e., $A = A \cup \{o^{*}_{ij+1}\}$
8.      end
9.  end
10. determine $\tau^{*} = \min\{\tau_{ij} \mid o_{ij} \in A\}$ and the cloud service $m^{*}$ on which $\tau^{*}$ could be realised
11. build a set $B$ from all abstract services in $A$ which are processed on $m^{*}$ and then determine $\sigma^{*} = \min\{\sigma_{ij} \mid o_{ij} \in B\}$
12. build a set $C$ in accordance with parameter $\theta$ such that $C = \{o_{ij} \in B \mid \sigma_{ij} \leq \theta \tau^{*} + (1 - \theta)\sigma^{*}, 0 \leq \theta \leq 1\}$
13. select the abstract service $o^{*}_{ij}$ from $C$ which has the highest priority (determined by the sort keys of the current individual being decoded) among abstract services in $C$ and delete it from $A$, i.e., $A = A - \{o^{*}_{ij}\}$
14. append abstract service $o^{*}_{ij}$ to the schedule and calculate its starting and completion time
15. if a successor $o^{*}_{ij+1}$ of the abstract service $o^{*}_{ij}$ exists then
16.      insert the successor $o^{*}_{ij+1}$ it into $A$, i.e., $A := A \cup \{o^{*}_{ij+1}\}$
17. end
18. until $A \neq \emptyset$
4.4 New Genetic Algorithms

generation, rather than conducting gene-by-gene mutation as is usually done, with a very small probability at each generation. In this algorithm, 83% of offspring are generated from crossover, and 15% of offspring are generated by introducing randomly generated new individuals as is done in the population’s initialisation. The remaining 2% of offspring of the next generation are generated by directly copying the top 2% fittest individuals from the current population. The percentages for these operators were obtained through doing trials on randomly generated test problems, leading to the best performance in the trials.
Fitness Function

For the deadline constraint problem, the fitness function is defined by Equation 4.7, in which $V(X)$ denotes the total constraint violation degree for composite services.

Equation 4.8 gives the definition of $V(X)$, where $V_i(X)$ defined by Equation 4.9 stands for the constraint violation degree for composite web service $W_i$.

$$\text{Fitness}(X) = \begin{cases} \frac{F_{\text{Cost}}^{\text{Max}}}{F_{\text{obj}}(X)}, & \text{if } V(X) \leq 1; \\ \frac{1}{V(X)}, & \text{otherwise.} \end{cases} \quad (4.7)$$

$$V(X) = \prod_{i=1}^{n} (V_i(X)) \quad (4.8)$$

$$V_i(X) = \begin{cases} \frac{F_{i,n_i}}{d_i}, & \text{if } F_{i,n_i} > d_i; \\ 1, & \text{otherwise.} \end{cases} \quad (4.9)$$

In Equation 4.7, $V(X) \leq 1$ means there is no constraint violation. $V(X) > 1$ means some constraints are violated, and the larger the value of $V(X)$, the higher the degree of constraint violation. $F_{\text{Cost}}^{\text{Max}}$ is the worst $F_{\text{obj}}(X)$, namely the maximal total cost, among all feasible individuals in a current generation. $\frac{F_{\text{Cost}}^{\text{Max}}}{F_{\text{obj}}(X)}$ is to scale the fitness value of all feasible solutions into range $[1, \infty)$. Using Equation 4.7, we can guarantee that the fitness of all feasible solutions in a generation are better than the fitness of all infeasible solutions. In addition, the
lower total cost for a feasible solution, the better fitness the solution will have.
The higher degree of constraints that are violated by an infeasible solution, the worse fitness the solution will have.

For the cost constraint problem, the fitness function is defined by Equation 4.10, in which $F_{Time}^{Max}$ denotes the worst $F_{obj}(X)$, namely the maximal total response time, among all feasible individuals in a current generation. $V(X)$ denotes the total constraint violation degree for all composite services. Equation 4.11 gives the definition of $V(X)$, where $V_i(X)$ defined by Equation 4.12 stands for the constraint violation degree for composite web service $W_i$.

$$Fitness(X) = \begin{cases} \frac{F_{Time}^{Max}}{F_{obj}(X)}, & \text{if } V(X) \leq 1; \\ \frac{1}{V(X)}, & \text{otherwise}. \end{cases}$$ \hspace{1cm} (4.10)

$$V(X) = \prod_{i=1}^{n} V_i(X)$$ \hspace{1cm} (4.11)

$$V_i(X) = \begin{cases} \frac{Cost_i(X)}{c_i}, & \text{if } Cost_i(X) > c_i; \\ 1, & \text{otherwise}. \end{cases}$$ \hspace{1cm} (4.12)

### 4.4.2 The Cooperative Coevolutionary Genetic Algorithm

This section presents a new cooperative coevolutionary genetic algorithm (CCGA) for the QoS-based resource allocation and scheduling problem. Different from the random-keys genetic algorithm that uses a single population for evolution,
the CCGA uses multiple populations for co-evolution. Each population solves a subproblem, and all populations cooperate in order to solve the entire problem. The CCGA can exploit problem space parallelism. It has the potential to generate better solutions than traditional single population-based GAs for very complex QoS-based resource allocation and scheduling problems, where both the number and the size of the composite services are large. The remainder of this section describes the CCGA.

The Cooperative Coevolution Model

The core idea behind the CCGA is a cooperative coevolution model, that is, several species, or subpopulations are co-evolved together. In this model, each species constitutes a partial solution to the entire problem, and the combination of an arbitrary individual from all these species generates a complete solution for the entire problem. The subpopulations of the CCGA evolve independently for solving their corresponding subproblem. However, purely independent evolution of these subpopulations is inadequate for solving an entire problem. Therefore they must also interact periodically to acquire feedback on how well they are working together at solving the entire problem, which also guides their subsequent progress. In such a model, two major issues must be addressed: problem decomposition, and interaction between subpopulations. The first issue is to determine how many subpopulations are required and what role each should play.
The latter issue is to determine how these subpopulations should interact for co-adaptation to solve the entire problem. Below is an explanation of the strategies we use to address these two issues:

**Problem Decomposition** Problem composition can be either static, where the entire problem is partitioned by hand and the number of subpopulations is fixed, or dynamic, where the number of subpopulations is adjusted during run time. Since the problem studied here can be naturally decomposed into a fixed number of subproblems by hand that will be demonstrated in the following part of this section, the algorithm for the problem is a static one, containing a fixed number of subpopulations, each of which solves one subproblem separately.

As previously written, the issue is to find a satisfactory resource allocation scheduling solution for all composite web services. We define finding a resource allocation scheduling solution for each composite web service as a subproblem. Therefore, the CCGA has $N$ subpopulations, where $N$ is the number of composite web services involved in the entire problem, and each subpopulation is responsible for solving one subproblem independently.

Taking the problem illustrated in Figure 4.2 as an example, there are two composite web services. Based on problem decomposition method we use, the CCGA will have two subpopulations, each of which is to maintain the allocation and scheduling solutions for one of the two composite web services.
The Interaction Method  For CCGAs, interaction between subpopulations occurs when evaluating the fitness of individuals of each subpopulation. The fitness of a particular individual of a certain population is an estimate of how well it cooperates with other species to produce good solutions. Guided by the fitness obtained in this way, different subpopulations can develop complementary features that together can solve the problem. This fitness interaction consists of the following two steps:

1. **Collaborator selection** — selecting collaborator subcomponents from each of the other subpopulations, and assembling the subcomponents with the current evaluating individual to form a complete solution. There are many ways of selecting collaborators [108]. We have conducted an initial experiment to compare the existing performance methods, indicating that there is no significant difference between these strategies. Therefore, we use the simplest one — choosing the best individuals from the other subpopulations, and assembling these individuals with the current individual to form a complete solution. This method is also called *greedy collaborator selection method* [108].

2. **Credit assignment** — assigning credit to the individual. This is based on the principle that the higher the fitness the complete solution has - constructed by the above collaborator selection method - the more credit
the individual will obtain. Regarding the fitness of the complete solution, we still use the ones defined in Equation 4.7 for the deadline constraint problem, and Equation 4.10 for the cost constraint problem. By doing so, in the following evolving rounds, an individual resulting in better cooperation with its collaborators will be more likely to survive. In other words, this credit assignment method can enforce the evolution of each population towards a better direction for solving the entire problem.

**CCGA Algorithm** Algorithm 9 describes the pseudocode of the CCGA algorithm. The CCGA algorithm works as follows:

- **Step 1** is responsible for initialising all the subpopulations in the CCGA.

- **Step 2** to **Step 7** are responsible for evaluating the fitness of each solution in the initial subpopulations. The evaluation of each subpopulation member consists of two steps: 1) combing the current individual $\text{indiv}[i][j]$ ($\text{indiv}[i][j]$ denotes the $j$th individual in the $i$th subpopulation in the CCGA) with the $j$th individual from each of the remaining subpopulations to form a complete solution (Step 4), and 2) assigning the fitness value of the complete solution (based on Equation 4.7 or on Equation 4.10) as the fitness value of the current evaluating individual (Step 5).

- **Step 8** to **Step 18** are responsible for co-evolving of the $N$ subpopulations
in the CCGA. In a co-evolving round, the $N$ subpopulations of the CCGA evolve one by one from the 1st subpopulation to the $N$th subpopulation. When evolving a subpopulation $SubPop[i]$, we apply the selection, crossover, mutation and fitness evaluation on the subpopulation, like a traditional GA. The only difference is the fitness evaluation, in which we use the greedy collaborator selection method and the credit assignment method introduced above to evaluate the fitness of the individuals in $SubPop[i]$. In the algorithm, the co-evolving runs iteratively until a certain termination criteria is satisfied.

With respect to the model, and still using the example in Figure 4.2, the co-evolving process of the CCGA is as follows:

1. Generate two subpopulations, as there are two composite services in the problem. The subpopulations are initialised randomly.

2. (a) Evolve the first subpopulation using a traditional GA, while the other subpopulation remains inactive by applying the selection, crossover and mutation operators on the first subpopulation to generate a new population of individuals (offspring). Then evaluate the fitness of each individual in the first subpopulation based on the interaction method introduced above. 1) Select the best individual of the second subpopulation and combine it with the current evaluating individual in
Algorithm 9: The cooperative coevolutionary genetic algorithm

1. Construct $N$ sets of initial populations, $SubPop[i], i = 1, 2, \ldots, N$
2. for $i \leftarrow 1$ to $N$ do
   3. foreach individual $indiv[i][j]$ of the subpopulation $SubPop[i]$ do
      4. $c \leftarrow \text{SelectPartnersBySamePosition}(j)$
      5. $indiv[i][j].\text{Fitness} \leftarrow \text{ObjectFunc}(c)$
   6. end
3. end
4. while termination condition is not true do
   5. for $i \leftarrow 1$ to $N$ do
      6. Select fit individuals in $SubPop[i]$ for reproduction
      7. Apply the crossover operator to generate new offspring for $SubPop[i]$
      8. apply the mutation operator to offspring
   9. foreach individual $indiv[i][j]$ of the subpopulation $SubPop[i]$ do
      10. $c \leftarrow \text{SelectPartnersByBestFitness}$
      11. $indiv[i][j].\text{Fitness} \leftarrow \text{ObjectFunc}(c)$
   12. end
   13. end
   14. end
the first population to form a complete solution for the entire problem.

2) Assign the fitness of the constructed complete solution as the fitness value of the current individual.

(b) Evolve the second subpopulation using a traditional GA, while the first one remains inactive, also by applying the selection, crossover and mutation operators on the second subpopulation, and then evaluating the fitness value of each individual in the second subpopulation based on the interaction method introduced before.

3. End a co-evolving round, and justify whether the stopping condition is satisfied. If not, start another co-evolution round. Otherwise, stop and return the complete solution.

**Representation**

With respect to the problem decomposition methods, there are $N$ subpopulations in the CCGA, where $N$ is the number of composite services involved in the problem. The chromosomes of all $N$ subpopulations are encoded using a random-key method. Based on the method, a chromosome of a subpopulation is encoded as a list of real numbers, where the list’s length equals the number of abstract services involved in the composite service corresponding to the subpopulation of the chromosome. As per the representation method in the random key GA, the value of a gene in the CCGA consists of:
1. an integral part, denoting the index of the cloud service in the candidate cloud service set, which is selected to be used by the abstract service corresponding to the gene

2. a fraction generated randomly from range [0,1), which is used as a sort key to decode the scheduling solution. The sort key represents the priority of the abstract service to use a cloud service, when there are multiple tasks competing for the same cloud service. An abstract service with a lower sort key will have a higher priority to use the conflicting cloud service.

Figure 4.5 shows an example of the random-key representation. An advantage of the random-key representation for the CCGA is that assembling any chromosome from the subpopulations forms a unique allocation and scheduling solution for the entire problem, because the combined key values represent a unique priority sequence. This means the random-key representation can always guarantee feasibility when assembling partial solutions to form a complete solution for the entire problem.

**Decoding**

As explained above, decoding is the procedure of generating an allocation and scheduling solution based on the current chromosome, so as to evaluate the fitness of the chromosome. Unlike the random-key GA, here a chromosome represents only a partial allocation and scheduling solution. It must combine
with the representative chromosomes from other subpopulations to generate a complete solution. Here, we select the best solution from each of the other subpopulations as the representative chromosomes, and combine them with the current chromosome in the active subpopulation to form a complete solution for the entire problem. The combined result serves the same purpose as the chromosome of the random-key GA introduced in the previous section. This is used as an input of the decoding procedure introduced in Section 8 to generate a complete solution, namely, an allocation and scheduling solution for all composite
Genetic Components

All subpopulations use a uniform crossover to generate offspring. That is, two parents of a subpopulation swap their genes with crossover probability 0.5. Figure 4.6 illustrates the uniform crossover. First, two chromosomes are randomly chosen from a subpopulation to act as parents. Then, for each gene, a real random number in the interval [0, 1] is generated. If the random number obtained is smaller than 0.5, then the allele of the first parent is used. Otherwise, the allele used is that of the second parent.

The mutation operator is the same as the one introduced in Section 18. 83% of offspring are generated from crossovers, the remaining 15% of offspring are
generated by introducing randomly generated new individuals as is done during population initialisation. The last 2% of offspring of the next generation are generated by directly copying the top 2% of the fittest offspring from the current population. The percentages for these operators were obtained through doing trials on randomly generated test problems.

4.5 Experiments

Several experiments were conducted to evaluate the scalability and effectiveness of the Random-key GA (RGA) and the Cooperative Coevolutionary Genetic Algorithm (CCGA). In order to conduct the evaluation, both the algorithms were implemented in Microsoft Visual C♯ 2005. All the experiments were conducted in a desktop computer with a 2.33 GHz Intel Core 2 Duo CPU and a 1.95 GB RAM. The parameter settings for the algorithms are illustrated in Table 4.1. These parameters were obtained through doing trials on randomly generated test problems. The parameters that led to the best performance in the trials were selected as the settings of the algorithms for the experiments introduced below.

4.5.1 Simulation Model

The scalability and effectiveness of the RGA and the CCGA were tested on a number of problem instances with different sizes and complexities. The size of the problem is determined by three factors: the number of composite services
Table 4.1: Simulation parameters for the experiments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>RGA</th>
<th>CoGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>200</td>
<td>100</td>
</tr>
<tr>
<td>Crossover rate</td>
<td>85%</td>
<td>85%</td>
</tr>
<tr>
<td>Mutation - replace rate of individuals</td>
<td>15%</td>
<td>15%</td>
</tr>
<tr>
<td>Stop - # of generations without improvement</td>
<td>40</td>
<td>20</td>
</tr>
</tbody>
</table>

involved in the problem, the number of abstract services in each composite service, and the number of candidate cloud services for each abstract service.

We constructed three types of problems, each designed to evaluate how one of the three factors affects the computation time and solution quality of the RGA and the CCGA. The complexity correlates with the constraint density of the problem. Therefore, we constructed problem instances with different constraint densities, and observed the solution qualities of the two GAs on these problems. Below is an elaboration of the test problems:

1. **Test problems with different numbers of composite services** — We constructed five test problems with different numbers of composite services, ranging from 5 to 25. Problem one is illustrated in Figure 4.7, containing five composite services, each of which has ten abstract web services. It is also the building block for constructing the other four problems. For instance, we used two building blocks to construct Problem two with ten composite services. The other test problems were constructed using this
one as a building block. For instance, we concatenated two instances of the original test problem to create one containing ten composite web services. For each of the test problems, we set the deadline constraint density to 'Medium' - its meaning is introduced later - to construct five deadline constraint problems. Similarly, we set the cost constraint density to 'Medium' to construct five cost constraint problems.

2. Test problems with different numbers of abstract services in each composite service — We constructed five test problems by fixing the number of composite services in each problem to ten, and varying the number of abstract web services in each composite service for different problems, ranging from 5 to 25. The test problem with five abstract web services is illustrated in Figure 4.8. We used the test problem as a building block to build test problems with more abstract services. For example, we concatenated two instances of the building block to construct a composite web service with ten abstract services. We used this approach to build test problems containing 15, 20 and 25 abstract web services. The deadline constraint density and cost constraint density were also set to 'Medium' for the deadline constraint problems and the cost constraint problems, respectively.

3. Test problems with different numbers of candidate cloud services for
each abstract service — We constructed five test problems with different candidate web services by fixing the number of composite web services and abstract web services and varying the number of candidate web services from 5 to 25 with an increment of 5. All the constraints were set to 'Medium' for both the deadline constraint problems and the cost constraint problems.

4. Test problems with different constraint densities — We used the second set of problems, namely the set of test instances varying in the number of abstract services, as the test base. We set three constraint densities on each test problem, including ‘High constraint’, ‘Medium constraint’, and ‘Low constraint’, in order to construct test problems with different constraint densities. Below is an introduction of how to obtain the 'High', 'Medium' and 'Low' constraint:

- Different constraint hardnesses for the deadline constraint problem — The high deadline constraint $D_{\text{high}}$ is the longest average execution time for a composite service. It is obtained using a cost optimisation algorithm called greedy cost [114] to solve the cost optimisation problem without a deadline constraint. The low deadline constraint $D_{\text{low}}$ is the shortest average execution time for a composite service. It is obtained using an execution time optimisation
algorithm called Heterogeneous-Earliest-Finish Time [114] for the time optimisation problem without a cost constraint. The medium deadline constraint $D_{\text{medium}}$ equals the average value of the $D_{\text{high}}$ and $D_{\text{low}}$.

- **Different constraint hardesses for the cost constraint problem** —

The high cost constraint and low cost constraint were set using the greedy cost [114] and the Heterogeneous-Earliest-Finish Time [114] for the cost optimisation problem without a deadline constraint, and time optimisation problem without a cost constraint, respectively. The medium cost constraint equals the average value of the total cost for all composite services found by the two algorithms.

### 4.5.2 Experiments on the Number of Composite Services

This experiment evaluated how the number of composite services affects the computation time and solution quality of the two GAs. In this experiment, we also compared the convergence speed of the two GAs. Considering the stochastic nature of genetic algorithms, we ran both the GAs 30 times on all test instances.

The objective for the cost optimisation problems with deadline constraints, is to optimise the total execution cost for all composite services. The results for the cost optimisation problems found by the RGA and the CCGA are presented in Table 4.2. From the table it can be seen that the two GAs can always find a feasible
solution for each test problem. In addition, the CCGA can always find a better result for each test problem than the RGA. For example, for the problem with five composite services, the CCGA found a planning and scheduling solution for the composite services resulting in a total execution cost for all composite services of $79, while the RGA could only find a solution costing $103. Thus, $24 can be saved by using the CCGA. From the table it can also be seen that all the test results were statistical significant based on the T-test results, as all the P-values were less than 0.05 (the reason can be found elsewhere [13]).

The objective for the execution time optimisation problem with cost constraints, is to optimise the sum of the execution times for all composite services. For each of the test problems, both GAs could find a feasible solution,
Figure 4.8: Building blocks for the second kind of test problems with different numbers of abstract services in each composite service

and the CCGA could also find a better result than the RGA (see Table 4.3).

The growth trend of the computation time of the GAs for the deadline constraint problem, as the number of composite services increases, is shown in Figure 4.9. From the figure we can see that the computation time of the RGA increases close to linearly from 25.4 seconds to 226.9 seconds. The computation time of the CCGA increases in a super linear trend, from 6.8 seconds to 261.5 seconds. However, this doesn’t mean that the CCGA is slower than the RGA, because the two GAs have different settings, in terms of population and generation size. To compare their speeds, we recorded the evolving processes of the two GAs, and observed which GA can find a better solution in the same time. Figure 4.10
Table 4.2: Comparison results of the RGA and the CCGA for deadline constraint problems with different numbers of composite services

<table>
<thead>
<tr>
<th>Comp. Serv.</th>
<th>RGA</th>
<th></th>
<th>CCGA</th>
<th></th>
<th>T-Test Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Feas.</td>
<td>Avg. Cost ($)</td>
<td>Feas.</td>
<td>Avg. Cost ($)</td>
<td>P-Value</td>
</tr>
<tr>
<td>5</td>
<td>Yes</td>
<td>103</td>
<td>Yes</td>
<td>79</td>
<td>0.01</td>
</tr>
<tr>
<td>10</td>
<td>Yes</td>
<td>171</td>
<td>Yes</td>
<td>129</td>
<td>8.23E – 04</td>
</tr>
<tr>
<td>15</td>
<td>Yes</td>
<td>326</td>
<td>Yes</td>
<td>251</td>
<td>7.20E – 12</td>
</tr>
<tr>
<td>20</td>
<td>Yes</td>
<td>486</td>
<td>Yes</td>
<td>311</td>
<td>3.65E – 15</td>
</tr>
<tr>
<td>25</td>
<td>Yes</td>
<td>557</td>
<td>Yes</td>
<td>400</td>
<td>9.15E – 07</td>
</tr>
</tbody>
</table>

Table 4.3: Comparison results of the RGA and the CCGA for cost constraint problems with different numbers of composite services

<table>
<thead>
<tr>
<th>Comp. Serv.</th>
<th>RGA</th>
<th></th>
<th>CCGA</th>
<th></th>
<th>T-Test Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Feas.</td>
<td>Time (Sec)</td>
<td>Feas.</td>
<td>Time (Sec)</td>
<td>P-Value</td>
</tr>
<tr>
<td>5</td>
<td>Yes</td>
<td>659</td>
<td>Yes</td>
<td>481</td>
<td>1.94E – 13</td>
</tr>
<tr>
<td>10</td>
<td>Yes</td>
<td>1785</td>
<td>Yes</td>
<td>1371</td>
<td>6.24E – 20</td>
</tr>
<tr>
<td>15</td>
<td>Yes</td>
<td>3009</td>
<td>Yes</td>
<td>2122</td>
<td>3.20E – 10</td>
</tr>
<tr>
<td>20</td>
<td>Yes</td>
<td>4452</td>
<td>Yes</td>
<td>2932</td>
<td>5.04E – 06</td>
</tr>
<tr>
<td>25</td>
<td>Yes</td>
<td>5792</td>
<td>Yes</td>
<td>3779</td>
<td>7.37E – 04</td>
</tr>
</tbody>
</table>
depicts the convergence speeds of the two GAs for the five test instances. Plot one in Figure 4.10 shows that for the problem with five composite services, the RGA took around 7 seconds to converge and the total execution cost, namely the optimisation objective, of the best solution found for the RGA was $97. Finding a solution with the same total execution cost, the CCGA used 10 seconds. So in this case the CCGA is slower than the RGA. But for the rest of the four test instances, with a larger problem size than the first one, the CCGA was always faster than the RGA. For example, for the problem with ten composite services (see plot two in Figure 4.10), the RGA took approximately 45 seconds to converge and the total execution cost of the best solution found by the RGA was $171. Compared to the RGA, the CCGA took only 6 seconds to find a solution with the same quality (approximately $\frac{1}{7}$ of the RGA’s convergence time). For larger size problems, the convergence speed gaps between the two GAs were larger.

### 4.5.3 Experiments on the Number of Abstract Services in Each Composite Service

This experiment was an evaluation of how the number of abstract services in each composite service affects the computation time and solution quality of the two GAs. Convergence speeds of the two GAs were also compared.

The comparison results of the solution quality found by the two GAs for the deadline constraint problems is shown in Table 4.4. It can be seen that both the
4.5 Experiments

GAs successfully found a feasible solution for each of the five test instances. The table also shows that the CCGA can always find a better solution for each of the five test problems verses the RGA, and all the test results are statistical significant based on the T-test results. For the cost constraint problems, the results were quite similar to the deadline constraint problems. Both GAs could find a feasible solution for all the problems and the CCGA outperformed the RGA in terms of solution quality.

The trends of the computation time of the two GAs, as the number of abstract web services involved in each composite service increases, can be seen from Figure 4.11. As is shown in the figure, the RGA’s computation time increases linearly from 29.8 seconds to 152.3 seconds, as the number of abstract web

Figure 4.9: Number of composite services versus the computation time of the RGA and the CCGA
services involved in the each composite service grows from 5 to 25. The CCGA’s computation time also increases linearly from 14.8 seconds to 72.1 seconds.

The convergence speeds of the two GAs for the five test instances were also recorded. Figure 4.12 depicts the convergence speeds of the two GAs for solving the problem with five abstract web services in each composite service. The figure shows that the RGA took 29.8 seconds to find a solution and the total execution cost of the solution was $105. Finding a solution with the same quality, the CCGA took only 3 seconds, only almost 1/10 of the convergence time of the RGA. For the remaining four problems, the results were quite similar, that is, the CCGA always converged faster than the RGA.
4.5 Experiments

Table 4.4: Comparison results of the RGA and the CCGA for deadline constraint problems with different numbers of abstract web services

<table>
<thead>
<tr>
<th>Abst. Serv. #</th>
<th>RGA</th>
<th>CCGA</th>
<th>T-Test Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Feas.?</td>
<td>Avg. Cost ($)</td>
<td>Feas.?</td>
</tr>
<tr>
<td>5</td>
<td>Yes</td>
<td>105</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>Yes</td>
<td>220</td>
<td>Yes</td>
</tr>
<tr>
<td>15</td>
<td>Yes</td>
<td>336</td>
<td>Yes</td>
</tr>
<tr>
<td>20</td>
<td>Yes</td>
<td>458</td>
<td>Yes</td>
</tr>
<tr>
<td>25</td>
<td>Yes</td>
<td>604</td>
<td>Yes</td>
</tr>
</tbody>
</table>

4.5.4 Experiments on the Number of Candidate Cloud Services for Each Abstract Service

This experiment was an evaluation of how the number of candidate cloud services for each abstract service affects the computation time and solution quality of the two GAs. The convergence speeds of the two GAs were also evaluated in this experiment.

Table 4.5 presents the results of the best solution found by the two GAs for the deadline constraint problems, and the T-test results. From the table we can see that the CCGA could always find a better result than the RGA for each of the test problems, and the results were statistical significant based on the T-test results. In addition, both of the two algorithms could find a feasible solution for each problem, satisfying all deadline constraints for the composite services. For the cost constraint problems, similar results were obtained.

Figure 4.13 shows that as the number of candidate cloud services for each
abstract service increases, the computation time of the two GAs was constant. For RGA, its computation time was around 21 seconds for all problems. For CCGA, its computation time was 11 seconds.

Regarding the convergence speeds of the two GAs for the five test problems, the CCGA also converged faster than the RGA. Figure 4.14 presents a result for the problem with 25 candidates for each abstract service.

### 4.5.5 Experiments on the Constraint Density

This experiment was to evaluate the effectiveness of the two GAs on problems with different constraint densities. For 'Low' deadline constraint problems, both GAs could always find a feasible solution. The execution costs of the solutions found by the two GAs were compared, as is shown in Figure 4.15. We can see
Figure 4.12: Comparison of the convergence speeds of the RGA and the CCGA for the problem with five abstract services

that the execution cost of all composite services under the planning and scheduling solution found by the CCGA was always less than the execution cost found by the RGA.

For the ‘Medium’ deadline constraint problem, similar results were obtained. That is, both algorithms could find a feasible solution that can satisfy all deadline constraints for the composite services, and the solutions found by the CCGA were better.

The two GAs could not find a feasible solution for ‘High’ deadline constraint problems. There might be no feasible solution at all because of the highly constrained sets in the problems. Even so, the constraints violation degree (defined by Equation 4.8) of the solutions found by the algorithms were not very
Table 4.5: Comparison results of the RGA and the CCGA for deadline constraint problems with different numbers of candidate cloud services for each abstract service

<table>
<thead>
<tr>
<th>Comp.Serv. #</th>
<th>RGA</th>
<th>CCGA</th>
<th>T-Test Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Feas.?</td>
<td>Avg. Cost ($)</td>
<td>Feas.?</td>
</tr>
<tr>
<td>5</td>
<td>Yes</td>
<td>144</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>Yes</td>
<td>142</td>
<td>Yes</td>
</tr>
<tr>
<td>15</td>
<td>Yes</td>
<td>140</td>
<td>Yes</td>
</tr>
<tr>
<td>20</td>
<td>Yes</td>
<td>141</td>
<td>Yes</td>
</tr>
<tr>
<td>25</td>
<td>Yes</td>
<td>142</td>
<td>Yes</td>
</tr>
</tbody>
</table>

high, as can be seen from Figure 4.17. Figure 4.17 also shows that the CCGA can find solutions with fewer constraint violations than the RGA. For example, for the problem with 15 abstract services in each composite, the CCGA found a solution with a constraint violation equal to 2.0, while the RGA found a solution with a constraint violation equal to 6.7.

Based on the above experimental results, the following conclusions can be drawn:

- The two GAs are both scalable. The computation time of the RGA increased linearly as the number of composite services or the number of abstract web services in a composite service increased. Its computation time was not affected by the number of candidate cloud services for each task. The CCGA is faster than the RGA, which was demonstrated by the comparison results of the convergence speeds of the two GAs on different test instances.
4.5 Experiments

Figure 4.13: Number of candidate cloud services versus the computation time of the RGA and the CCGA

For larger sized problems, the convergence speed gap between the two algorithms was larger, meaning that the CCGA is more suitable for larger size problems.

- The two algorithms are effective. In addition, the CCGA can find better results than the RGA. For ‘low’ and ‘medium’ constraint problems, both algorithms can always find feasible solutions that ensure all the composite services can satisfy their deadline or cost constraints. For all the test problems, the CCGA could find better results than the RGA. Moreover, as the problem size increased and the problems became more complex, the solution quality gap between the two GAs seemed to be larger. For ’high’ constraint problems, even though the two algorithms could not find a
feasible solution for the constructed problems, noting there might be no feasible solution at all, the constraints violation degree of the solutions found by the two algorithms was not very high. In addition, the CCGA still seemed to perform better than the RGA, in terms of constraint violation degree.

4.6 Verification

We have also conducted an experiment to verify the correctness of the Random-key Genetic Algorithm (RGA) and the Cooperative Coevolutionary Genetic Algorithm (CCGA). This was achieved by testing the two GAs on a carefully designed problem whose optimal result can be deduced, and therefore we can
check whether the GAs can find the optimal result.

The problem was constructed as follows: The test instance shown in Figure 4.18 was used as the basis of the problem. As can be seen from the figure, it contains six composite web services, each of which consists of a chain of tasks. The objective of the problem was to minimise the total response times for these composite web services without any constraint on any of the composite
web services. For each task involved in the test problem, it contained a set of candidate cloud web services that could be invoked by the task, which were either public cloud services or private cloud web services. The QoS values, including response time and price, of these candidate web services were generated based on the assumption that for a task, the response times of its public cloud web services are shorter than the response times of its private cloud web services. The QoS dataset used in this verification is shown in Table 4.6. In the table, the first element in pairs gives the execution time in seconds, and the second element in pairs gives the price in dollars. The cloud web services with a price of zero are private cloud web services, and all the others are public cloud web services.

The optimal solution for the above problem (the solution with the minimal total response times) must be the one that uses the fastest public cloud (in bold in Table 4.6) among the candidate web services for each task. This is because on
one hand the use of the fastest public cloud for each task will lead to the shortest
execution time for the task, and on the other hand, the multi-tenancy feature of a
public cloud will cause no extra waiting time for each task.

Any alternative solution that uses any other cloud web service for a task will
worsen the result. This can be assumed because a slower public cloud web service

Table 4.6: QoS values of candidate cloud web services for each task involved in
the problem

<table>
<thead>
<tr>
<th>Candidate Service ID</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(10,10)</td>
<td>(20,10)</td>
<td>(5,8)</td>
<td>(8,6)</td>
<td>(5,6)</td>
<td>(6,3)</td>
</tr>
<tr>
<td>2</td>
<td>(15,5)</td>
<td>(25,8)</td>
<td>(15,6)</td>
<td>(10,5)</td>
<td>(10,4)</td>
<td>(12,5)</td>
</tr>
<tr>
<td>3</td>
<td>(30,0)</td>
<td>(30,6)</td>
<td>(20,0)</td>
<td>(15,0)</td>
<td>(16,0)</td>
<td>(18,0)</td>
</tr>
<tr>
<td>4</td>
<td>(38,0)</td>
<td>(40,5)</td>
<td>(25,0)</td>
<td>(24,0)</td>
<td>(18,0)</td>
<td>(20,0)</td>
</tr>
<tr>
<td>5</td>
<td>(40,0)</td>
<td>(40,0)</td>
<td>(35,0)</td>
<td>(30,0)</td>
<td>(20,0)</td>
<td>(28,0)</td>
</tr>
</tbody>
</table>
will cause the longer execution time for a task. The private cloud web service will result in the longer execution time for a task, and a possible delay of consuming the service for the task because of possible competition among tasks for using the same private cloud service. Therefore, we can deduce that the optimal result, the minimal total response times for all composite services, is the sum of the execution times of all the tasks involved in all composite web services. Each task uses the fastest public cloud web service among the candidate services for the task, resulting in a total execution time of all composite services of 306 seconds.

To verify the correctness of the RGA and the CCGA, we ran both of the algorithms 30 times on the above problem. The best, average and worst results of the RGA and the CCGA on the problem are presented in Table 4.7. As can be seen, for the RGA, the best result found by it was 306 seconds, which is the optimal result. The worst result found by it was 315 seconds, which was close to the optimal result. The RGA found the optimal solution in six of 30 runs. Testing the CCGA, it successfully found the optimal result in 26 of 30 runs, while for only one time it found a result of 308 seconds, which was very close to the optimal result. Based on these results, we can conclude that both the GAs can find the best solution.


Table 4.7: Results of the RGA and the CCGA on the test problem

<table>
<thead>
<tr>
<th></th>
<th>Best</th>
<th>Average</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGA</td>
<td>306</td>
<td>310.2</td>
<td>315</td>
</tr>
<tr>
<td>CCGA</td>
<td>306</td>
<td>306.2</td>
<td>308</td>
</tr>
</tbody>
</table>

4.7 Summary

This chapter studied the QoS-based resource allocation and scheduling problem for multiple composite web services on hybrid clouds. The problem was formulated as a resource allocation and scheduling problem, and was solved using a random-key based genetic algorithm and a CCGA. Simulation results demonstrated the good scalability and effectiveness of the two algorithms. A verification experiment was also performed to validate the correctness of the two GAs.

The positive scalability and effectiveness demonstrated in the experiments give the GAs the potential to be applied in QoS-based resource allocation and scheduling problems for multiple composite web services on hybrid clouds, where the number of composite web services and the size of composite services are large. This is particularly true for the CCGA. The CCGA benefits from its capability of exploiting problem space parallelism by introducing a cooperative cooperation model. The CCGA uses less time to find a better solution, as compared to traditional genetic algorithms. This makes the CCGA a better choice
for addressing large-scale problems rather than traditional genetic algorithms.
Chapter 5

Performance-Driven Composite Web Service Partitioning for Decentralised Execution

When a composite web service is executed centrally, a single web service engine is responsible for coordinating the execution of the components. This may create a bottleneck and degrade the overall performance, such as throughput and response time of the composite service when there are a large number of service requests. This problem can be handled by decentralising execution of the composite web service. However, it raises the issue of how to partition a composite service into groups of component services so that each group can be orchestrated by its own execution engine and the overall performance of the composite is maximised. This chapter investigates this composite web service partitioning problem.
5.1 Introduction

Quality of Service (QoS) via the configuration of external component web services has been the major concern of most existing work [117, 112, 107]. This includes selecting external component web services for composite web services and scheduling external component web services as resources for composite web services. How the component services are combined, for instance, by a BPEL program, can also affect the composite’s QoS. This has been largely ignored by most researchers. A BPEL program executes on a single BPEL server which has to cope with all the computation and communicate with all the external component services. Therefore, the BPEL server tends to become a bottleneck that deteriorates the QoS of a composite web service executed on it.

To overcome the single-server bottleneck, Mangala et al. [67] advocated the use of a decentralised execution mode for BPEL programs. This decentralised execution mode partitions a BPEL program into several subprograms, each executing on a separate BPEL server. An illustrative example of such decentralised execution mode seen in Figure 5.2 for a ‘Loan Approval’ (LA) BPEL program (see the left plot in Figure 5.1) is presented. To improve the performance of the LA composite service, its corresponding BPEL program can be partitioned into five subprograms. Then the subprograms can be deployed on separate servers, D0 to D4, and pairs of servers can communicate via
5.1 Introduction

asynchronous messaging. The topology of the decentralised execution mode for the partitioning plan is shown at the bottom of Figure 5.2. Such decentralised execution of a composite web service is effective for improving response time and throughput. It increases parallelism when executing ‘glue’ code and reduces message overheads [67]. Figure 5.1 shows there are eight incoming edges and outgoing edges of the BPEL server in the topology of the centralised mode for the LA BPEL program. Each represents a communication between the BPEL server with an external entity. However, the bottom of Figure 5.2 shows a decentralised topology whereby the sum of incoming edges and outgoing edges of each BPEL server is no more than four. The communication costs of each BPEL server reduce to approximately half of the BPEL server in a centralised execution mode. In addition, it is apparent that in decentralised execution mode all BPEL servers will have lighter computation loads as they only execute a small block of the entire BPEL program.

Figure 5.1: Centralised execution mode for a ‘Loan Approval’ BPEL program
It is desirable to execute a composite web service in a decentralised manner. However, this raises the issue of how to partition a composite service implemented in a BPEL program into sub BPEL programs so that each sub BPEL program executes on an execution engine and the overall performance of the composite is maximised. This is the so-called BPEL program partitioning problem. The requirements of the problem have been identified by Mangala et al. [67] as listed below:

- A BPEL program consists of a set of statements (or activities) which can be classified into fixed activities and portable activities. A fixed activity must be executed at a particular server. Fixed activities in a BPEL program
include the receive, reply and invoke activities. The receive and reply activities are always located at the first and last server respectively. The invoke activities must collocate with the corresponding web service. A portable activity can be executed at any server. Apart from the three fixed activities, all other activities are portable activities, including send, assign, and if.

- There may exist data dependencies between two activities. In order to keep the semantics of the original BPEL program, the decentralised BPEL program generated by the partitioning must preserve these data dependencies. In the ‘Loan Approval’ BPEL program, a sample data dependence is presented in the statement pair “receive(client, req)” and “b.amt = req.amt”, because the output of the statement “receive(client, req)” is the input of the statement “b.amt = req.amt”. Violation of a data dependence will change the original execution order between two statements, so can not be allowed.

- There may exist control dependencies between two activities. In order to keep the semantics of the original BPEL program, the decentralised BPEL program generated by the partitioning must preserve these control dependencies. Mangala et al. [67] reveals that, in order to preserve the control dependencies, activities with different control dependencies cannot
be partitioned into the same server unless their control activities are in the same server. Again we use the ‘Loan Approval’ BPEL program as an example to demonstrate this requirement. The statement “\(g.risk = 0\)” depends on “\(if(req.amt < 10000)\)” since the value of “\(if(req.amt < 10000)\)” determines whether “\(g.risk = 0\)” is executed. Similarly, the statement “\(res.rate = b2.rate\)” depends on “\(if(b1.rate < b2.rate)\)”.

If “\(g.risk = 0\)” and “\(res.rate = b2.rate\)” are in the same server, their control activities “\(if(req.amt < 10000)\)” and “\(if(b1.rate < b2.rate)\)” must also be on the same server in order to preserve the control dependencies.

- Different decentralised execution topologies will lead to different performances of a composite service. It has been demonstrated by Chafle et al. [67] that a wisely designed decentralised execution topology will minimise communication overheads and balance the computation load for execution servers. Therefore, there is a need to find a partitioning plan from the large number of potential partitioning plans for a given BPEL program so that the overall performance of the composite is maximal.

Fundamentally, this is a graph partitioning problem with an objective to optimise the performance of a composite BPEL-based web service. It is conjectured that the computational complexity of the problem is NP-hard [92] and the computational complexity of two existing heuristic algorithms has been proven
Figure 5.3: Program dependency graph for the ‘Loan Approval’ BPEL program to be exponential. In this chapter, a genetic algorithm (GA) will be used to address the problem. GAs have good scalability and can handle various constraints with ease using a penalty-based technique. To the best of our knowledge, this is the first attempt to use GA to address the BPEL program partitioning problem. The next section presents a formal definition of the BPEL program partitioning with a detailed GA for addressing the problem.

5.2 Problem Formulation

According to the requirements described in the previous section, a formal description of the BPEL program partitioning problem is presented below.

**Inputs**

1. A BPEL program represented by a program dependency graph (PDG) [40]
$G = (F, P, D, C)$ (see Figure 5.3).

- Fixed activities $F = \{f_1, f_2, ..., f_{N_f}\}$, where $N_f$ is the number of fixed activities in the BPEL program. Each fixed activity is denoted by a rectangular node in Figure 5.3.

- Portable activities $P = \{p_1, p_2, ..., p_{N_p}\}$, where $N_p$ is the number of portable activities in the BPEL program. Each portable activity is denoted by a round cornered node in Figure 5.3.

- A set of data dependencies $D = \{\langle v_i, v_j \rangle | v_i, v_j \in F \cup P \}$. Pair $\langle v_i, v_j \rangle$ represents activity $v_j$ uses the data generated by activity $v_i$. Each data dependency is denoted by a dotted edge in Figure 5.3. $D$ is a directed acyclic graph (DAG).

- A set of control dependencies $C = \{\langle v_i, v_j, c_k \rangle | v_i, v_j \in F \cup P \}$, and $c_k \in \{\text{True, False, Parallel}\}$. Triple $\langle v_i, v_j, c_k \rangle$ represents a control dependency edge from $v_i$ to $v_j$ with control condition is $c_k$. Vertex $v_i$ is a control condition statement and vertex $v_j$ is another statement control dependent on $v_i$. Elements $\text{True}$ and $\text{False}$ are used to represent control dependency edge from a conditional statement to another statement. Element $\text{Parallel}$ is used to represent control dependency edge from the “entry” point to a general statement. Each control dependency is represented by a solid edge in Figure 5.3. $C$ is
essentially tree-structured and named control flow graph (CFG) [40].

2. A computation cost for each receive, reply, invoke, send, ..., and assign activity on server \( i \), represented by \( \text{Cost}_{\text{Receive}}(i) \), \( \text{Cost}_{\text{Reply}}(i) \), \( \text{Cost}_{\text{Invoke}}(i) \), \( \text{Cost}_{\text{Send}}(i) \), ..., and \( \text{Cost}_{\text{Assign}}(i) \) respectively.

Constraints

- Precedence constraint set \( PC = D \cup C \). Precedence constraints define a partial order between two statements. \( PC \) equals \( D \cup C \) because each data dependency edge or control dependency edge in the PDG introduces an essential ordering between two statements. \( PC \) is used to guarantee that there are no dependency cycles in a partitioning plan.

- Control dependency constraint set \( CDC = \{ \langle v_i, v_j, r \rangle \mid v_i \text{ and } v_j \text{ are assigned to the same partitioning. There must be a route } r \text{ in the CFG so that all the vertices on route } r \text{ are assigned to the same partition as } v_i \text{ and } v_j \} \). \( CDC \) is used to prevent new predicates being introduced in a partitioning plan.

Output A partitioning plan \( X = \{ \langle p_i, f_j \rangle \mid p_i \in P, f_j \in F \} \), where \( |X| = N_p \), and pair \( \langle p_i, f_j \rangle \) means portable activity \( p_i \) is assigned to the same partition as fixed activity \( f_j \), so that \( F_{\text{obj}} \) is maximal and the constraints in \( PC \) and \( CDC \) are satisfied.
\[ T(s_i) = \frac{\text{Capacity}(i)}{\text{Cost}(i)} \quad (5.1) \]

\[ F_{\text{obj}}(X) = \min(T(s_i)) \quad (5.2) \]

\[ \text{Cost}(i) = \text{Num}_{\text{Receive}}(i) \times \text{Cost}_{\text{Receive}}(i) + \]
\[ \text{Num}_{\text{Reply}}(i) \times \text{Cost}_{\text{Reply}}(i) + \]
\[ \text{Num}_{\text{Invoke}}(i) \times \text{Cost}_{\text{Invoke}}(i) + \]
\[ \text{Num}_{\text{Send}}(i) \times \text{Cost}_{\text{Send}}(i) + \]
\[ \ldots \]
\[ \text{Num}_{\text{Assign}}(i) \times \text{Cost}_{\text{Assign}}(i) \quad (5.3) \]

Equation 5.1 is the definition of throughput on a single server \( s_i \), in which \( \text{Capacity}(i) \) is the capacity of server \( s_i \) and \( \text{Cost}(i) \) is the sum of the computational costs of the activities assigned to server \( s_i \).

Function \( F_{\text{obj}}(X) \) defined in Equation 5.2 returns the throughput of the given BPEL program under partitioning plan \( X \). Function \( T(s_i) \) returns the throughput of server \( s_i \). The overall throughput is the minimal throughput of all the servers.

Equation 5.3 defines the computation cost \( \text{Cost}(i) \) on server \( s_i \), in which \( \text{Num}_{\text{Receive}}(i), \text{Num}_{\text{Reply}}(i), \text{Num}_{\text{Invoke}}, \text{Num}_{\text{Send}}, \text{and} \text{Num}_{\text{Assign}}(i) \) represent the number of receive, reply, invoke, send, and assign activities on server \( s_i \), respectively.
5.3 Related Work

The BPEL program partitioning problem was first introduced by Mangala et al. [67] as a variant of the general problem of automatic program parallelisation. Automatic program parallelisation has been studied intensively in compiler optimisation for multiprocessor systems [93, 60, 17] and hardware-software co-design of embedded systems [70]. The problem is often formulated as separate partitioning and scheduling problems. Both the BPEL program partitioning problem and traditional partitioning and scheduling problems involve dividing programs into tasks and allocating the tasks to processors with a performance trade off between parallelism and communication overheads. However, the BPEL partitioning and scheduling problem has its own distinguishing features: 1) Tasks in a BPEL program are categorised into fixed tasks and portable tasks. Fixed tasks must be pre-allocated to specific processors and portable tasks can be assigned to any processors with spare capacity. This introduces extra constraints for the BPEL program partitioning problem. 2) The objective of the BPEL program partitioning problem discussed by Nanda et al. is to optimise throughput [67], whereas the program parallelisation problem concentrates on optimising execution time.

Program slicing techniques are used for automatic program parallelisation. Among several such techniques [101], performing optimisation based on a program dependency graph (PDG) is the most commonly-adopted strategy. For
many optimisation problems, such as the partitioning and scheduling problem for parallel programs, a PDG is effective because: 1) it can handle data dependencies and control dependencies in a simple way; and 2) an optimisation algorithm can be encoded as a PDG traversal algorithm, while one-time traverse of a PDG can perform optimisation, promoting search efficiency. Various techniques such as the list scheduling heuristic and the critical path heuristic [57] are also applied to accelerate search speed.

Due to the unique features of the BPEL program partitioning problem just mentioned, existing PDG-based approaches and heuristics are difficult to apply to the BPEL program partitioning problem. Mangala et al. [67] reviewed previous algorithms and introduced two new algorithms for the BPEL program partitioning problem. They are the Merge-by-Define-Use heuristic (MDU) and the Pooling-and-Greedy-Merge heuristic (PGM) algorithms. These two algorithms actually use the same idea, to wit, iteratively merging tasks based on a PDG. The only difference between them is the heuristics applied when searching. The Merge-by-Define-Use heuristic is inspired by the fact that merging nodes along data dependency edges will result in a more communication-efficient partitioning. The MDU algorithm can prune the space of possible solutions. The complexity of the MDU algorithm is $O(e^p)$, where $e$ is the maximum number of data dependency edges that enter or exit a portable node, and $p$ is the maximum number of portable nodes that are considered in an iteration. For large BPEL programs, both $e$ and $p$
could be large, thus leading to long computation times. The Pooling-and-Greedy-Merge algorithm combines two heuristics. One is a greedy-merge heuristic which is a refinement of the MDU heuristic and reduces the search space. The second heuristic is a pooling heuristic that merges flow dependence edges together to reduce the number of edges. It also merges tasks along flow dependency edges. Variable $e$ can be made small using the PGM heuristic. However, the heuristic can only be applied to specific node structures in a BPEL program. For some BPEL programs, the value of $e$ might remain large. In addition, the PGM might exclude a large part of the search space where good solutions could lie.

In this research, a new penalty-based GA is developed for the BPEL program partitioning problem. Unlike the MDU and PGM algorithms, the PDG is transformed into two graphs: a dependency graph representing precedence constraints and a control flow graph to represent control dependency constraints. A variation of a topological sorting algorithm working on the dependency graph and a merging algorithm on the control flow graph is developed to calculate penalty values for handling precedence constraints and control dependency constraints. GAs do not fine tune local optima. Therefore, a novel local search algorithm is incorporated with the GA to improve solution quality. This GA has good scalability and performance. It is suitable for large-scale and complex BPEL program partitioning problems.
5.4 A GA for the BPEL Partitioning Problem

In this study, a penalty-based GA is developed to address the BPEL program partitioning problem defined in the previous section. Penalty mechanisms are adopted for handling precedence constraints and control dependency constraints. In addition, a fast local optimiser is incorporated into the GA to improve solution quality. The selection mechanism of the GA is a rank-based selection with elitism [97], whereby the best individual is kept alive across generations.

Algorithm 10 describes the GA. In the algorithm, Step 1 is responsible for generating the initial population using a new initialisation procedure instead of the traditional random generation of initial population. This introduces good seeds in populations to increase the likelihood of finding optimal solutions. Step 2 is responsible for evaluating the fitness of all individuals in the initial population. Step 3 to Step 9 are the evolving processes in which basic components of a GA, such as selection, crossover, mutation and fitness evaluation, as well as a local optimiser to fine tune solutions are included.

The initial population generation, genetic encoding, genetic operators, the penalty mechanisms, and the local optimiser will now be explained. The Loan Approval (LA) BPEL program partitioning problem shown in Figure 5.3 is used to illustrate these concepts.

A feasible partitioning solution for the LA example must meet the following
Algorithm 10: The genetic algorithm for the partitioning problem

1. create an initial population of $PopSize$ individuals, $Population$, based on the initialisation procedure in Section 5.4.2
2. evaluate the fitness of each individual in $Population$
3. while termination condition is not true do
   4. select fit individuals from $Population$ for reproduction
   5. probabilistically apply the crossover operator to generate $PopSize$ offspring
   6. probabilistically apply the mutation operator to offspring
   7. evaluate the fitness of each individual in $Population$
   8. apply the local optimiser in Section 5.4.7 to each feasible individual to explore possible improvements in terms of solution quality
4. end

conditions:

1. Each of the fixed nodes must be assigned to a separate partition. A feasible partitioning is illustrated in Figure 5.4 where there are five partitions as determined by the number of fixed nodes in this workflow.

2. Precedence dependency constraints must be met. Precedence dependency constraints can be represented by a directed acyclic graph (DAG) called a precedence dependency graph. It contains both data dependency edges and control dependency edges from the Program Dependency Graph. Essentially, the precedence dependency graph is a subgraph of the PDG, containing all nodes and edges in the PDG except for the “Entry” node and edges linked with the “Entry” node. For instance, the precedence dependency graph for the LA example is shown in Figure 5.5. Partitioning
will result in a new precedence dependency graph called the *partitions dependency graph* and will represent precedence dependencies between partitions. A partitioning plan is feasible only if there is no dependency cycle in the partitions dependency graph. For example, in Figure 5.5, the middle plot shows a partitioning plan for the LA example and the bottom plot shows the partitions dependency graph generated based on the plan. As there is a dependency cycle between partitioning D3 and D2, this partitioning plan is an infeasible one.

3. Control dependency constraints must be satisfied. The partitioning plan must guarantee that activities with different control dependencies not be partitioned into the same partition unless their control activities are in the
same partition. An infeasible partitioning plan for the LA example is illustrated in Figure 5.6. In the given solution, P6 and P7, both placed in partition D3 are under different conditional branches of the predicate statement P5. However, P5 is not placed in partition D3, meaning that P6 and P7 will execute in a sequential order in partition D3. Therefore, the control dependency constraint is violated.
Figure 5.6: An infeasible partitioning for the ‘Loan Approval’ example violating control dependency constraints

5.4.1 Genetic Encoding

As shown in Figure 5.4, the chromosome is encoded by an integer array of length equal to the number of portable nodes in the BPEL program. Each gene $P_i$ in the chromosome represents a portable node and has a value ranging from 0 to $m$, where $m$ is the number of fixed nodes in the BPEL program. For the ‘Loan Approval’ example there are 7 portable nodes in the BPEL program. Therefore, the chromosome consists of 7 genes, $P_1$ to $P_7$. Each gene’s value ranges from 0 to 4 and represents a fixed node that the portable node is assigned to. For example, $P_1 = 0$ means portable node $P_1$ is assigned to fixed node $F_0$ and $P_3 = 1$ means portable node $P_3$ is assigned to fixed node $F_1$. 
5.4.2 Initial Population Generation

Traditionally, the initial population is generated randomly. However, considering the complexity of the problem caused by various constraints, a procedure that always generates feasible initial solutions satisfying precedence dependency constraints is introduced. The aim of applying this algorithm is to introduce good seeds in populations to increase the likelihood of finding optimal solutions. The procedure is based on topological sorting as this can generate ordered nodes meeting precedence constraints. Partitioning is then conducted on the chains of ordered nodes. Algorithm 11 is the algorithm description for the initialisation procedure. In the algorithm, Step 2 is responsible for topologically sorting nodes in the partially-ordered precedence dependency graph $precGraph$ to generate a totally-ordered nodes list $orderedList$. An example is illustrated in Figure 5.7.

Figure 5.7: Initialization step for the ‘Loan Approval’ example
For the ‘Loan Approval’ example, one possible output of the topological sorting of nodes is; F0, P1, P2, F1, P3, P4, F3, F2, P5, P7, P6, F4. Step 3 is responsible for partitioning the ordered list \textit{orderedlist} while guaranteeing that in each partition there is exactly one fixed node. In each running of Step 2 and Step 3, a feasible initial solution satisfying precedence dependency constraints can be generated. An example of a solution is illustrated at the bottom of Figure 5.7. In the algorithm, Step 2 and Step 3 iteratively run for \textit{PopSize} times to generate \textit{PopSize} initial solutions.

**Algorithm 11:** The initialization procedure

- **input:** \texttt{precGraph} the precedence dependence graph and \texttt{PopSize} population size
- **output:** \texttt{Population} the initial population

\begin{verbatim}
1 for i=1 to PopSize do
2   orderedlist ← TopologicalSorting(precGraph)
3   Population[i] ← PartitionList(orderedlist)
4 end
\end{verbatim}
5.4.3 Genetic Operators

Crossover and mutation are the most important search space exploration operations in genetic algorithms. The algorithm uses one crossover operator and one mutation operator as shown in Figure 5.8. Crossover is based on a ring connected by the first and last nodes in a chromosome. To generate a child chromosome, a randomly-selected slice of the ring is chosen from one parent and the remaining genes are taken from the other parent. A second child is then generated by reversing the selection sequence between the first and second parents. The mutation operator randomly selects a portable node (i.e., a position in the genome) and replaces its current fixed node with another fixed node.

5.4.4 Fitness Function

The fitness function is defined as the sum of a function of the throughput of a partitioning plan and a penalty function determined by the degree of infeasibility raised by violation of precedence and control dependency constraints. It is important to search through infeasible regions, particularly for highly constrained problems. An optimal solution can often be reached most efficiently via an infeasible region. Good feasible solutions can be a product of breeding between feasible and infeasible solutions. Therefore, the strategy adopted by the algorithm is to allow infeasible individuals in the population but apply a penalty to their fitness values.
The following two guidelines are used when defining the fitness function. Firstly, an infeasible individual has a lower fitness value than that of any feasible individual. Secondly, an infeasible individual that violates more constraints should be penalised more than an infeasible individual that violates fewer constraints. Equation 5.4 gives the definition of the fitness function whereby $F_{obj}(X)$ is the objective function defined in Section 5.2, and $V(X)$ stands for the total number of constraint violations of potential BPEL partitioning $X$. Therefore, when $V(X)$ equals zero, it implies partitioning $X$ is a feasible individual. Otherwise, it is infeasible.

$$Fitness(X) = \begin{cases} 
0.5 + 0.5 \times F_{obj}(X)/T_{max}, & \text{if } V(X) = 0; \\
0.5 \times F_{obj}(X)/T_{max} - P_D(X) - P_C(X), & \text{otherwise.} 
\end{cases}$$ (5.4)

$$T_{max} = \text{Max(Capacity}(i))/\text{Min(Cost)}$$ (5.5)

As per Equation 5.4, if a BPEL program partitioning $X$ is feasible, its fitness value is given by expression $0.5 + 0.5 \times F_{obj}(X)/T_{max}$. If a partitioning $X$ is infeasible, its fitness value is calculated as $0.5 \times F_{obj}(X)/T_{max} - P_D(X) - P_C(X)$. Components $P_D(X)$ and $P_C(X)$ denote the precedence dependency constraints penalty and control dependency constraints penalty, respectively. $T_{max}$ is a constant denoting an upper bound of the overall throughput result which is used
to scale $F_{obj}(X)/T_{max}$ in range $(0, 1]$. $T_{max}$ is defined by Equation 5.5 as the throughput of the server whose capacity is maximal (namely $Max(Capacity(i))$) among all the servers. It hosts only one activity that consumes the lowest computation cost $Min(Cost)$ of all activities. This definition of $T_{max}$ guarantees that all possible overall throughput results are less than $T_{max}$. Therefore, $F_{obj}(X)/T_{max}$ is always in range $(0, 1]$.

The sum of $P_D(X)$ and $P_C(X)$ is scaled to range 0 to 0.5. How to calculate $P_D(X)$ and $P_C(X)$ is explained in Sections 5.4.5 and 5.4.6. The value of expression $0.5 + 0.5 * F_{obj}(X)/T_{max}$ is between 0.5 and 1 and the value of expression $0.5 * F_{obj}(X)/T_{max} - P_D(X) - P_C(X)$ is less than 0.5. Therefore, an infeasible partitioning has a lower fitness value than any feasible one.

5.4.5 Precedence Constraints Handling

Partitioning might introduce dependency cycles. Therefore, a variation of the topological sorting algorithm was introduced to check whether or not there were cycles in the partitioning plan. The inputs of the algorithm included a candidate partitioning plan. The output of the algorithm has a penalty value for the potential solution which ranged between 0 and 1. The algorithm used a so-called priority list which is a topological sorting of the fixed nodes in the precedence dependency graph. Fixed node $F_i$ is always bound to partition $D_i$ so partitions in the priority list were used instead. For example, the priority list resulting from the precedence
dependency graph shown in Figure 5.5 is \( \langle D_0, D_1, D_2, D_3, D_4 \rangle \).

A feasible partitioning will give the penalty value 0. An infeasible partitioning will yield a penalty exceeding 0 and the more precedence dependency constraints the infeasible plan violates, the higher a penalty value it will receive.

The algorithm is described by Algorithm 12. The detailed working mechanisms of the algorithm are as follows:

1. In Step 1 we merge nodes in the precedence dependency graph for the BPEL program according to the candidate partitioning. An edge in the graph will be deleted if two nodes connected by an edge are in the same partition. An edge will be added between the two partitions where the two nodes are located. This is called partitions precedence dependency graph \( G \).

2. From Step 6 to Step 21 we conduct topological sorting on graph \( G \). The usual topological sorting procedure is an iterative process. In each step a node with zero in-degree and its edges will be deleted from graph \( G \). The procedure will stop when there is no zero in-degree node or all the nodes are deleted. A slight change of the procedure is made for our purpose here; the procedure will only stop when all the nodes in graph \( G \) are deleted like Step 6. If there is no zero in-degree node but there are still nodes in graph \( G \) like Step 13 then there are cycles in graph \( G \). Therefore, we break the cycles by reversing some edges to make graph \( G \) a feasible one. This edge
Algorithm 12: A penalty value calculating algorithm for precedence constraint violation

**input**: \( \text{precGraph} \) the precedence dependence graph

**input**: \( X \) the current partitioning solution

**output**: \( \text{penaltyd} \) the penalty value given to \( X \) for its precedence dependence constraint violation

1. construct the partition precedence dependence graph \( G \) based on the current solution \( X \)
2. randomly generate a priority list \( \text{list} \) for nodes in \( G \)
3. \( \text{penaltyd} \leftarrow 0 \)
4. find the zero in-degree node \( \text{Node}_0 \) from \( G \)
5. if \( \text{Node}_0 \) exists then push \( \text{Node}_0 \) into stack \( \text{ZeroInDegreeStack} \)
6. while exist nodes in \( G \) do
7.  if \( \text{ZeroInDegreeStack} \) is not empty then
8.     \( \text{Node}_0 \leftarrow \text{Pop} (\text{ZeroInDegreeStack}) \)
9.     delete \( \text{Node}_0 \) from \( G \)
10.    delete \( \text{Node}_0 \) from the priority list \( \text{list} \)
11.  end
12. else if exist nodes in \( G \) then
13.     find the node \( \text{Node}_h \) with the highest priority in the priority list \( \text{list} \)
14.     reverse all incoming edges of \( \text{Node}_h \) in \( G \) to convert \( \text{Node}_h \) into a zero in-degree node
15.     sum the number of total reversed edges and accumulate the value to the penalty value \( \text{penaltyd} \)
16.     \( \text{Node}_0 \leftarrow \text{Node}_h \)
17.     push \( \text{Node}_0 \) into stack \( \text{ZeroInDegreeStack} \)
18. end
19. end
20
21
Figure 5.9: Topological sorting on a new precedence dependency graph to calculate the penalty for precedence constraints violation

reversing process involves two major operations:

(a) Selecting a node that has not been deleted in $G$ as the next output node depends on where it is situated in the priority list. Step 14 shows
that the earlier a node occurs in the priority list, the higher priority it receives.

(b) Reserving all the incoming edges of the selected node (see Step 15).

After the edges reversing process, topological sorting can continue as the selected node becomes a zero in-degree node. In this way, all the nodes can be visited. The number of total reversed edges $Num_{reverse}$ in the whole process represents the distance of changing the input partitioning to a feasible one or the number of constraints violation edges. The precedence dependency penalty $P_D(X)$ equals $Num_{reverse}/4 \times Num_{total}$, where $Num_{total}$ is the total number of edges in the input precedence dependency graph. The range of $P_D(X)$ is $[0, 0.25)$. The control dependency penalty $P_C(X)$ is also scaled to $[0, 0.25)$ so that the sum of $P_D(X)$ and $P_C(X)$ is scaled to range 0 to 0.5 as explained in Section 5.4.4.

With respect to the modified topological sorting procedure, and still using the ‘Loan Approval’ process as an example, the topological sorting process (bottom part of Figure 5.9) for a sample partitioning solution (top part of Figure 5.9) is written below.

1. Output D0, as D0 is a zero in-degree node.

2. Output D1.
3. Find a cycle between D2 and D3 and conduct an edge reversal operation. As the priority list is \( \langle D_0, D_1, D_2, D_3, D_4 \rangle \), D2 is selected as the next output node because it has the highest priority among the partitions not handled. It has two incoming reversed edges.

4. Output D2.

5. Output D3.


The number of reversed edges is 2 and the total number of edges in the precedence dependency graph is 11, so the precedence dependency constraints penalty for the sample partitioning is \( \frac{2}{4} \times 11 = 0.0455 \).

### 5.4.6 Control Dependency Constraints Handling

We used a control flow graph (CFG) based merging algorithm to calculate control dependency penalties. The algorithm merges nodes in each region (namely each partition of the input partitioning plan), based on the CFG from bottom to top to explore execution paths in that region. Nodes in a region forming an execution path are merged together and all nodes in that region are expected to be merged as one node. This means that all the conditional branches in that region are under the same control node and therefore, the region is feasible. However, if there are multiple nodes in a region when merging completes, some conditional branches
will execute in a sequential order between each other, rather than being controlled by a predicate. In this case the region is infeasible. The more unmerged nodes that exist in a region, the more control dependency constraints are violated in that region. The penalty values are determined by the total number of unmerged nodes in all the regions. Algorithm 13 describes the pseudocode of the merging algorithm.

**Algorithm 13:** A penalty value calculating algorithm for control dependence constraint violation

| input: PDG the program dependence graph |
| input: X the current partitioning solution |
| output: $P_C(X)$ the penalty value given to $X$ for its control dependence constraint violation |
| Data: Num parts the number of nodes left in CFG after merging finishes |
| Data: $N_f$ the number of fixed nodes in $PDG$ |
| Data: $N_p$ the number of portable nodes in $PDG$ |

1. generate a control flow graph $CFG$ according to the $PDG$
2. $currentLevel ← lowestLevel$
3. while $currentLevel$ is the top level do
4.  while exists that two nodes are placed in the same region by the candidate partitioning plan and they have the same parent node and control condition merge do
5.  | merge the two nodes together |
6.  end
7.  while exists a node cannot be merged with any of the nodes in the same level but if its parent node is placed in the same region with it, merge the node with its parent do
8.  | merge the node with its parent |
9.  end
10. set $currentLevel$ as its upper level
11. end
12. $P_C(X) = (Num_{parts} - N_f)/4 \times N_p$
Figure 5.10: Control flow graph for the ‘Loan Approval’ example

An illustrative example of the algorithm for the LA example is shown in Figure 5.10. The upper part of Figure 5.10 illustrates a partitioning plan and the lower part gives the merging process of the algorithm. Firstly, the nodes in the lowest level of the CFG, P2, F1, P3, P6, and P7 are merged according to the two merging rules. In this case only node P2 is merged with its parent node P1, using the second rule. Then, the nodes located in the next higher level, F0, P1, P4, F2, F3, P5, F4, are merged. In this step F0 is merged with P1, P4 is merged with F2, and P5 is merged with F4. The number of nodes left in the CFG is 8 and the
penalty given to the sample partition plan is \((8 - 5)/4 \times 7 = 0.107\).

### 5.4.7 Local Optimiser

Pure genetic algorithms perform poorly when fine tuning local optima. Therefore, local improvement heuristics are usually applied to new offspring to fine tune them [14]. We introduced a local search optimiser tailored to our problem. The local optimiser was applied to each feasible individual after an evolving round of the GA to explore the neighboring solutions of the individual for possible improvement. An intensive search of the whole neighbourhood space could be a time-consuming process. Therefore, we introduced two mechanisms in the local optimiser to prune the search space. The local optimiser only searched the most promising neighbourhood space for a better solution. The two mechanisms are as follows:

1. **Bottleneck partition-based local search** — The local optimiser only focused on the optimisation of the bottleneck partition. Other partitions were ignored. A bottleneck partition is one that has the lowest throughput of all the partitions. Its throughput is exactly the throughput of the entire partitioning solution. An improvement of throughput in the bottleneck partition is most likely to improve the throughput of the overall solution, compared with optimising the throughput of other partitions. Based on this idea, we generate neighbourhood solutions by moving one portable
node assigned to the bottleneck partition to another randomly selected partition. Moving a portable node might help to improve the throughput of the partition because the computation cost of the partition is decreased. If a neighbouring solution has a better fitness than the original one, it is accepted to replace the original. This means an improvement is made and the local optimiser stops. Otherwise, we try different neighbouring solutions using the neighbourhood generation rule until an improvement is made or no alternatives are left.

2. Exploring the neighbourhood space using the merge-by-define-use heuristic — The merge-by-define-use heuristic introduced in Section 5.3 is employed by the local optimiser to further reduce the neighbourhood search space. The idea behind the merge-by-define-use heuristic is that merging nodes along data dependency edges might reduce communication between partitions. A more communication-efficient partition can then be generated. When generating a neighbouring solution, a portable node located in the bottleneck partition is moved to another partition that has a merge-by-define-use relationship with the bottleneck partition, instead of a randomly selected partition. A move of the portable node in this way may decrease the computation cost on the bottleneck partition and its communication cost.
Algorithm 14 describes the pseudocode of the local optimiser. The algorithm repetitively applies the above two mechanisms to find all possible neighbourhood solutions of the input solution $X$ until it finds a neighbourhood solution $X'$ that has better fitness than the input solution $X$. This means an improvement is achieved. Then, the algorithm uses $X'$ to replace input $X$.

**Algorithm 14:** Local optimizer algorithm

**input:** $X$ the current partitioning solution  
**output:** $X'$ the improved partitioning solution

1. find the bottleneck partition $F_{bottleneck}(X)$ under the current partitioning solution $X$
2. construct set $P_{bottleneck}$, containing all the portable nodes assigned to bottleneck partition $F_{bottleneck}(X)$
3. **foreach** portable node $p_{cur}$ of $P_{bottleneck}$ **do**
4. construct set $P_{Define-Use}(p_{cur})$, containing all portable nodes which have a define-use relationship with portable node $p_{cur}$
5. choose $p_{target}$ to be a random node selected from set $P_{Define-Use}(p_{cur})$
6. create a variant of partition $X$, called $X'$, in which portable node $p_{cur}$ is moved to the partition where portable node $p_{target}$ is located
7. **if** $Fitness(X') > Fitness(X)$ **then**
8. replace input $X$ with $X'$
9. break
10. **end**
11. **end**

### 5.4.8 Complexity Analysis

The genetic algorithm consists of five components: initial population generation, selection, crossover, mutation, calculation of the fitness of an individual, and local optimisation. We used a bookkeeping method to find the computational complexity for each of these components.
The population initialisation phase introduced in Section 5.4.2 performs a topological sorting procedure $N_{\text{pop}}$ times on the precedence dependency graph (the graph denoted by $D$ in Section 5.2) of the BPEL program to generate $N_{\text{pop}}$ individuals. Each topological sorting procedure has complexity $O(|E| + |V|)$, where $|E|$ and $|V|$ are the number of edges and vertices in the precedence dependency graph, respectively. The population initialisation phase has complexity $O(N_{\text{pop}} \times (|E| + |V|))$. In the genetic algorithm, $N_{\text{pop}}$ is a constant set to 100. Therefore, the population initialisation phase has complexity $O(|E| + |V|)$.

The rank-based selection procedure sorts individuals in ascending order by their fitness in each generation of the evolution. A sorting algorithm has complexity $O(N_{\text{item}} \lg(N_{\text{item}}))$ for $N_{\text{item}}$ items and there are $N_{\text{pop}}$ individuals in the population. So, performing the rank-based selection procedure has complexity $O(N_{\text{pop}} \lg(N_{\text{pop}}))$. Since the genetic algorithm evolves $Max_{\text{gen}}$ generations, the selection procedure is performed $Max_{\text{gen}}$ times. As a result, the rank-based selection has complexity $O(Max_{\text{gen}} \times (N_{\text{pop}} \lg(N_{\text{pop}})))$. In the GA, both $N_{\text{pop}}$ and $Max_{\text{gen}}$ are constants, set to 100 and 200, respectively. Therefore, the overall rank-based selection has complexity $O(1)$.

When conducting a crossover operator, the genetic algorithm visits all
genes of two parent individuals and copies the genes from the two parent individuals to two child individuals. An individual in the genetic algorithm is represented in an array of length $|P|$, where $|P|$ is the number of portable nodes, defined in Section 5.2, in a BPEL program. Visiting all genes in the array has complexity $O(|P|)$, so the crossover operator has complexity $O(|P|)$. The crossover operator is used $p_{crossover} \times N_{pop}/2$ times. However, because $p_{crossover}$ and $N_{pop}$ are constants in the genetic algorithm, the computational complexity of the overall crossover operation is still $O(|P|)$.

- The mutation operator randomly visits one gene in an individual and replaces its current value with another one, therefore having constant complexity $O(1)$. It is invoked $p_{mutation} \times N_{pop}$ times. As a result, the computational complexity for the overall mutation operation is still $O(1)$ as both $p_{mutation}$ and $N_{pop}$ are constants in the genetic algorithm.

- Evaluating the fitness of an individual comprises three procedures (see Section 5.4.4):

  1. Decoding the individual to a partitioning solution and calculating its throughput. The decoding phase contains the procedure for constructing the partitions precedence dependency graph for the partitioning plan. This is used in the stage of precedence constraints handling looked at in Section 5.4.5. It is based on the original
Performance-Driven Composite Web Service Partitioning for Decentralised Execution

precedence dependency graph. Each edge of the original precedence dependency graph is visited and operated on, either preserving or deleting the edge, with complexity $O(|V|)$. Calculating the throughput involves evaluating all partitions to get the minimal one as the throughput of the entire solution. Calculating the throughput has complexity $O(|F|)$, where $|F|$ is the number of fixed nodes (defined in Section 5.2) in a BPEL program. The first procedure has complexity $O(|V| + |F|)$ and because $|F| < |V|$, the complexity becomes $O(|V|)$.

2. Calculating the data dependence penalty. This is a topological sorting of the partitions dependency graph with complexity $O(|E| + |V|)$.

3. Calculating the control dependence penalty. This is a traversal of the control dependency graph, a tree structured graph defined by $C$ in Section 5.2, with complexity $O(|V|)$ (i.e., the complexity of traversing a tree with $|V|$ vertices).

Summing up the complexity of these three procedures, evaluating the fitness of an individual has complexity $O(|E| + |V|)$. The total number of fitness evaluations is $N_{pop} \times M_{gen}$. However, since both $N_{pop}$ and $Max_{gen}$ are constants, the computational complexity for the overall fitness evaluation operation is still $O(|E| + |V|)$.

- The local search optimiser needs to explore neighbouring solutions for
possible improvements. In the worst case, each such exploration needs to
generate \(|P|\) neighbouring solutions. This occurs when all portable nodes
are assigned to the bottleneck partition. The procedure tries to improve the
throughput of the bottleneck partition, by moving one portable node in the
bottleneck partition to another partition. Each portable node is used only
once, so there are up to \(|P|\) neighbouring solutions. Evaluating the fitness
of one neighbouring solution has complexity \(O(|E| + |V|)\). Therefore, the
local search procedure in the worst case has complexity \(O(|P| \times (|E| + |V|))\).
The local search optimiser is invoked \(N_{\text{pop}} \times Max_{\text{gen}}\) times. \(N_{\text{pop}}\) and
\(Max_{\text{gen}}\) are constants, so the computational complexity for the overall local
optimisation is still \(O(|P| \times (|E| + |V|))\).

The dominant complexity in each step of the genetic algorithm is that of the local
search optimiser, giving an overall complexity of \(O(|P| \times (|E| + |V|))\).

5.5 Experiments

This section compares the performance of the penalty-based GA with the
heuristic-based MDU and PGM algorithms [67]. We tested the results and
scalability of the three algorithms by applying them to a number of BPEL
programs with different problem sizes and different levels of precedence and
control dependency constraints.
5.5.1 Test Problems

BPEL programs in practice can be grouped into two categories. The first is BPEL programs without control condition statements. These have precedence dependency constraints, but do not have any control dependency constraints. The second category is BPEL programs with control condition statements. These have both precedence and control dependency constraints. Therefore, we constructed two categories of test problems with various complexities, as explained below.

Test Problems Without Control Dependency Constraints

The polymorph search workflow shown in Figure 5.11 is applied to the area of theoretical chemistry for constructing test problems of different complexities [38]. The figure shows it has no control condition dependencies. The polymorph search
Figure 5.12: Polymorph workflow test instance one

Workflow has a number of parallel branches, each of which may also contain parallel sub-branches. The number of parallel branches can be up to 38 and the number of parallel sub-branches can be up to 200 in order to increase the polymorph search’s precision.

In this experiment we used the polymorph search workflow to create instances of BPEL programs of different sizes and different complexities of precedence constraints. Since the problem size is determined by the number of portable and fixed nodes, we can easily increase the problem size by increasing the number of portable and fixed nodes in the workflow. The complexities of precedence constraints can be augmented by increasing the number of precedence constraint pairs in the workflow. Eight tests were generated from the examples. Figures 5.12
to 5.15 show four smaller ones. The instances were named according to the number of fixed nodes and portable nodes in the PDG of the BPEL program. For example, Problem 9-9 in Plot 1 of Figure 5.12, means there are 9 fixed nodes and 9 portable nodes in the PDG of its corresponding BPEL program. This can be seen in Plot 2 of Figure 5.12. Problems 12-13, 16-18 and 22-26 are shown in Figure 5.13, 5.14 and 5.15, respectively. Larger problems were constructed by adding more building blocks to the workflow. This is shown by the dotted box in Figures 5.15. Problem 42-52 has 4 parallel building blocks and Problems 52-65 and 62-78 have 5 and 6 building blocks, respectively.
Test Problems With Control Dependency Constraints

Test problems with control dependency constraints are classified into two types. Firstly, those with single-level control conditions. Secondly, those with two-level control conditions such as a BPEL if statement nested in another. Generally, problems with two-level control conditions have a higher density of control dependency constraints than those with single-level conditions.

The travel plan workflow shown in Plot 1 in Figure 5.16 was used as the basis for generating the tests. The building blocks for the single-level control condition problems are shown in Plot 2, circled by the dashed line. Plot 2 is the PDG for
the Travel Plan workflow and was used as the simplest one-level control condition problem with only one building block. It was named Problem 1. Figure 5.17 is a complex problem with two building blocks, named Problem 2. In the same way, Problems 3 to 8 were constructed using 3 to 8 building blocks.

The building blocks for the two-level control condition problems as seen in Figure 5.18, were generated based on the building blocks for single-level problems, by adding a new if statement nested in an existing if branch. Eight instances were constructed using 1 to 8 building blocks. Figure 5.19 is an example of the two building blocks problem.
5.5 Experiments

According to Equations 5.1, 5.2 and 5.3, $Capacity(i)$, $Cost_{Receive}(i)$, $Cost_{Reply}(i)$, $Cost_{Assign}(i)$, $Cost_{Invoke}(i)$, and $Cost_{Send}(i)$ must be known to calculate the throughput of a decentralised BPEL program under a partitioning plan. $Capacity(i)$ is the capacity of a server where a partition, namely a BPEL subprogram, resides. In the experiments, the capacity of each server, $Capacity(i)$,
was set to 100%. $Cost_{Receive}(i)$, $Cost_{Reply}(i)$, $Cost_{Assign}(i)$, $Cost_{Invoke}(i)$, and $Cost_{Send}(i)$ were set to 0.6%, 0.45%, 0.6%, 2.5% and 0.5% respectively as per Nanda et al.’s benchmark [67].

The population size of the genetic algorithm was 100 and the maximum number of generations was 200. The crossover probability and mutation probability were 0.8 and 0.2, respectively.
5.5 Experiments

5.5.3 Test Environment

All three algorithms were implemented in Microsoft Visual C♯ 2005 and executed on a desktop computer with a 2 Duo 2.33GHz CPU and a 1.95GB RAM.

5.5.4 Results for Problems Without Control Dependency Constraints

We used the three algorithms to solve each of the test problems and recorded their computation times and the throughput of the solutions they found. Considering the stochastic nature of GAs, we performed 30 runs of the algorithm for each test problem. For the MDU and PGM algorithms, we performed each test problem only once, as they both are deterministic algorithms. The results are shown in Figure 5.20 and Table 5.1.

Figure 5.20: Comparisons of the computation time of the genetic algorithm versus the MDU and PGM heuristic algorithms for partitioning polymorph workflows of different sizes
Table 5.1: Test results for the genetic algorithm, against the MDU and PGM heuristic algorithms, for polymorph search workflow partitioning, showing the algorithms’ execution times and the throughputs of the solutions found

<table>
<thead>
<tr>
<th>Portable nodes—fixed nodes (#)</th>
<th>GA</th>
<th>MDU</th>
<th>PGM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec)</td>
<td>Best</td>
<td>Worst</td>
</tr>
<tr>
<td>9-9</td>
<td>14.909</td>
<td>23.810</td>
<td>23.810</td>
</tr>
<tr>
<td>12-13</td>
<td>7.620</td>
<td>21.739</td>
<td>21.739</td>
</tr>
<tr>
<td>19-22</td>
<td>20.867</td>
<td>20.833</td>
<td>20.833</td>
</tr>
<tr>
<td>22-26</td>
<td>20.917</td>
<td>20.833</td>
<td>20.833</td>
</tr>
<tr>
<td>52-65</td>
<td>25.095</td>
<td>10.582</td>
<td>10.582</td>
</tr>
</tbody>
</table>

We can see that the computation time of the MDU algorithm increased dramatically as the problem size increased. The smallest problem, Problem 9-9, had only one branch. Therefore, its computation time was only 0.092 seconds. However, Problem 16-18 had 3 branches so the MDU algorithm took 127.746 seconds to find a solution. For problems larger than Problem 16-18, it could not find a solution in 15 minutes.

The PGM algorithm spent the least time among the three algorithms when the problem size was smaller than 42-52. However, for larger problems its computation time exceeded that of the GA and was more than 15 minutes when the problem size reached 62-78. Both the MDU and PGM algorithms exhibited exponential growth.

By contrast, the genetic algorithm always found a solution within 30 seconds for all the test problems. The computation time of the GA increased slowly as the problem size increased. Another interesting feature of the experimental results
was that the least time used by the GA was not for the smallest problem, 9-9 (14.909 seconds), but for the second smallest one, 12-13 (7.620 seconds). This is because the computation time of the GA depends not only on the number of nodes, but also on the corresponding partitioning topology of the solutions produced by the GA during evolving steps. The local optimiser dominates the computation cost of the GA. The computation cost of the local optimiser depends on the topology of the current solution to be optimised by the local optimiser. It specifically looks at the number of portable nodes located in the bottleneck region of the current solution.

The MDU algorithm found the best quality solutions. However, it could not find a solution in a reasonable time when the problem size was 19-22 or larger. The results found by the MDU algorithm in Table 5.1, showed that throughput decreased as more branches were introduced into the BPEL program. As the number of branches increased, the nodes at the conjunction point of the branches became a bottleneck and deteriorated overall throughput. The genetic algorithm’s results were generally equal to or close to those of the MDU algorithm. For Problems 9-9 and 12-13, the genetic algorithm always found the same solutions as the MDU algorithm. For Problem 16-18, the best result found by the GA had throughput of 21.505 requests/second, the same as the solution found by the MDU algorithm. The worst solution found by the GA had throughput of 20.833 requests/second. The gap between the worst and best solutions was only around
Figure 5.21: Comparisons of the computation time of the genetic algorithm versus the MDU and PGM heuristic algorithms for problems of different sizes with single-level control dependency constraints

0.67 requests/second. Therefore, the worst one was very close to the best one. For the remaining five problems with larger sizes, the solutions found by the genetic algorithm were always the best compared to the MDU and PGM algorithms. The MDU algorithm could not always find a solution in an acceptable time and the PGM algorithm’s solutions were the worst of all. The biggest gap between the solutions of the PGM and MDU algorithm’s was 6.11 requests/second for Problem 12-13. The results of the PGM algorithm were all 15.625 requests/second from Problem 12-13 to Problem 22-26. Therefore, the PGM algorithm tended to converge at a local optimal solution.
5.5 Experiments

Table 5.2: Test results for the genetic algorithm, and the MDU and PGM heuristic algorithms, for partitioning workflows with single-level control conditions, showing the algorithms’ execution times and the throughputs of the solutions found

| Control cond. (#) | GA | | | | MDU | | | | PGM | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Time (sec) | Throughput (requests/sec) | Time (sec) | Throughput (requests/sec) | Time (sec) | Throughput (requests/sec) | | | | | | |
| Best | Worst | Average | | | | | | | | | | |
| 1 | 8.825 | 18.868 | 18.868 | 18.8679 | 0.001 | 16.949 | 0.001 | 16.949 | 0.001 | 16.949 | 0.001 | 16.949 |
| 2 | 34.180 | 15.385 | 15.385 | 15.3846 | 0.029 | 15.625 | 0.016 | 15.625 | 0.001 | 16.949 | 0.001 | 16.949 |
| 3 | 43.266 | 15.385 | 15.152 | 15.3458 | 0.584 | 15.625 | 0.139 | 15.625 | 0.016 | 15.625 | 0.001 | 16.949 |
| 4 | 21.147 | 15.152 | 15.152 | 15.1515 | 12.776 | 15.152 | 0.996 | 15.152 | 0.016 | 15.625 | 0.016 | 15.625 |
| 5 | 26.776 | 15.152 | 15.152 | 15.1515 | 12.776 | 15.152 | 0.996 | 15.152 | 0.016 | 15.625 | 0.016 | 15.625 |
| 6 | 31.807 | 15.152 | 15.152 | 15.1515 | 12.776 | 15.152 | 0.996 | 15.152 | 0.016 | 15.625 | 0.016 | 15.625 |
| 7 | 35.501 | 15.152 | 14.286 | 15.1227 | 12.776 | 15.152 | 0.996 | 15.152 | 0.016 | 15.625 | 0.016 | 15.625 |
| 8 | 44.782 | 15.152 | 14.286 | 15.0361 | 12.776 | 15.152 | 0.996 | 15.152 | 0.016 | 15.625 | 0.016 | 15.625 |

5.5.5 Results for Problems With Control Dependency Constraints

The results of the three algorithms for test problems with single-level control dependency constraints were given in Figure 5.21 and Table 5.2. The computation times of the MDU and PGM algorithms were less than 1 second when the problem size was small such as a BPEL program with fewer than 4 control condition branches built from the TP workflow. However, computation times increased steeply and a solution could not be found in 15 minutes when more than 6 control condition branches were in the workflow. Although the genetic algorithm was slower than the other two for small problems, its computation times increased slowly as the number of control condition branches in the workflow increased. It could find a feasible solution within 45 seconds for all the problems tested.

Results for two-level control workflows are illustrated in Figure 5.22 and
Figure 5.22: Comparisons of the computation time of the genetic algorithm versus the MDU and PGM algorithms for problems of different sizes with two-level control dependency constraints.

Table 5.3. The overall trend of the computation times as the problem size increased was similar to the result for single-level control workflows. The MDU algorithm could not find a solution in 15 minutes when the number of branches was increased to 6, and the PGM algorithm could not find a solution in 15 minutes when the number was increased to 7. The GA could always find a feasible solution for all the test problems within 183 seconds.

Experimental results on the throughput of the partitioning plans found by the three algorithms are also illustrated in Tables 5.2 and 5.3. It can be seen that the throughput of the partitioning plans obtained by the GA are comparable with those of the other two algorithms even though the GA needed much less computation time than the two others for larger test problems. For six of the eight single-level
Table 5.3: Test results for the genetic algorithm, and the MDU and PGM algorithms, for partitioning workflows with two-level control conditions, showing the algorithms’ execution times and the throughputs of the solutions found

<table>
<thead>
<tr>
<th>Control cond. (#)</th>
<th>GA</th>
<th>MDU</th>
<th>PGM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time (sec)</td>
<td>Throughput (requests/sec)</td>
<td>Time (sec)</td>
</tr>
<tr>
<td>1</td>
<td>36.718</td>
<td>14.493</td>
<td>14.493</td>
</tr>
<tr>
<td>3</td>
<td>32.026</td>
<td>13.158</td>
<td>13.158</td>
</tr>
<tr>
<td>6</td>
<td>78.228</td>
<td>13.158</td>
<td>12.048</td>
</tr>
<tr>
<td>7</td>
<td>149.961</td>
<td>13.158</td>
<td>12.048</td>
</tr>
<tr>
<td>8</td>
<td>182.873</td>
<td>13.158</td>
<td>12.048</td>
</tr>
</tbody>
</table>

control condition problems, the GA found solutions equal to or better than the two other algorithms. Only the solutions for Problems 2 and 3 found by the GA were worse than those by the other two algorithms. Even then the gap was only 0.47 requests/second. For seven of the eight two-level control condition problems, the GA found solutions better or equal to the other two algorithms.

From the above experiments, the following conclusions were drawn.

**Scalability**  (1) The GA exhibited good scalability. In practice, its computation time increased slowly as the problem size increased. (2) The MDU algorithm’s scalability was poor. Its computation time increased quickly as the problem size increased. (3) The PGM algorithm was better than the MDU algorithm because it cut down the feasible search space. However, its computation time also increased quickly with respect to the problem size.
Performance  (1) For small problems, the MDU algorithm could find a solution with the best throughput in most cases but at the cost of high computational expenses. For large problems, it could not find a solution in a reasonable time. (2) The PGM algorithm easily converged to a locally optimal solution, however, it was sensitive to the structure of BPEL programs. For some problems it could find a solution as good as the solution found by the MDU algorithm. In all test problems, the largest gap between the PGM algorithm and the best algorithm was 6.11 requests/second. (3) The GA could find a solution for all the problems attempted. Sometimes it produced the best result because it identified a globally optimal solution while the other two algorithms could only converge on a locally optimal solution.

5.6 Verification

Experiments were also conducted to verify the correctness of the throughput generated by our partitioning algorithm. This was done by measuring the real throughput of different partitioning plans and then comparing the real throughput with the simulation results above. The single-level one-building block Travel Plan workflow seen in Figure 5.16 was used for this purpose. In the experiments, the throughput of the partitioning plan found by the genetic algorithm was 18.868 requests/second shown in Table 5.2. The MDU algorithm and the PGM algorithm both found the same partitioning plan with throughput of 16.949 requests/second.
5.6 Verification

To prove the correctness of these results, the real throughput of the partitioning plan found by the algorithm must be greater than the real throughput for the plan found by the MDU and PGM algorithms.

**Experimental Setup** To set up an environment for decentralised execution of composite web services, 5 machines (2 Duo 2.33GHz CPU and a 1.95 GB RAM) were used. The machines were connected by a 100 Mb/s LAN as composite web service execution servers. Each hosted a Microsoft BizTalk Server (2006 Version) [53] which was responsible for executing one subprogram of the test workflow. Microsoft BizTalk Servers cannot execute BPEL programs directly, but instead execute their own style workflows transformed from BPEL programs. The transformation was straightforward because for each BPEL statement used in the test problems there was a corresponding activity in the BizTalk notation. The BizTalk servers in the experiments invoked web services using SOAP over HTTP. Communication between BizTalk servers was based on asynchronous SOAP messages.

Web services invoked by the test workflow were also implemented and each deployed on a separate PC. These PCs were connected by a 100 Mb/s LAN.

A stress tests tool, Microsoft LoadGen 2007 [77], was used to generate a steady stream of requests with different rates. A performance monitoring tool (Perfmon) [110] executed on each Microsoft BizTalk Server recorded
Results  For each test case, LoadGen 2007 generated steady request rates from 5 requests/second to 22 requests/second. Each request rate was run for 10 minutes to test stable throughput of the workflow. The sizes of messages generated by LoadGen 2007 were all 18Kb. The experimental results are summarised in Figure 5.23. As shown, the maximal throughput of the workflow under the partitioning solution found by the MDU and PGM algorithms was 17 requests/second, while the maximal throughput of the workflow under the solution found by the genetic algorithm was 19 requests/second. Therefore, the solution found by the algorithm performed better in practice, as predicted.
5.7 Summary

Judicious partitioning of composite web services across parallel servers can increase web service throughput. Previously the problem of partitioning composite web services expressed as BPEL programs has been tackled using heuristic algorithms. In this chapter, a new penalty-based GA was described for handling the BPEL program partitioning problem. To the best of our knowledge, this is the first attempt to apply a GA to this problem. Experimental results showed that the performance of the GA compared favorably with previous algorithms as the size of the partitioning problem increased. Furthermore, the throughput results it produced were as good as, and sometimes better than, those produced by the slower algorithms.
Chapter 6

Conclusions and Future Work

This chapter summarises the findings and the contributions that originate from the research we did. It also presents possible future research directions.

6.1 Summary

In this thesis, the application of genetic algorithms (GAs) to the problems of QoS-based web service composition was studied. The overall goal was to develop effective and scalable GAs to address the QoS-based web service composition problems, which are characterised as complex, large-scale, highly constrained and multi-objective problems. This goal was successfully achieved by a proposed set of new GAs in the thesis. The results from conducted simulations and verifications allow quantitative statements about the good scalability and effectiveness of the GAs.

More specifically, three important technical problems involved in QoS-based web service composition were addressed by using GAs. The three problems were
QoS-based web service selection with inter-service dependencies and conflicts, QoS-based resource allocation and scheduling for multiple composite web services on hybrid clouds, and performance-driven composite service partitioning for decentralised execution. We conjectured that they are NP-hard problems. We proposed new GAs for each of the problems. These GAs were tested through experiments on a number of test instances. Experimental results demonstrated good scalability and effectiveness for the new GAs. This gave the new GAs the potential to be applied to QoS-based composite service management tools to deliver feasible, reliable and high-quality composite services.

Chapter 3 investigated the QoS-based web service selection problem. The two challenges involved in delivering a high quality composite web service, which used individual web services from external service providers, were identified as optimality and correctness. The optimality challenge is raised by customers’ demands on using the best services in terms of QoS. The correctness challenge is raised by many constraints on inter-service dependence and conflict inhabiting a web service-based environment. This is caused by technical incompatibilities, business preferences and other practical factors. Therefore, delivering a feasible composite service accommodating the constraints was a difficult task. The investigation of existing QoS-based web service selection algorithms demonstrated that these algorithms had difficulty addressing the problem introduced in the research. Most of the algorithms ignored the
correctness issue. A few that solved the problem suffered from poor scalability. We therefore explored how to use GAs to overcome the two challenges, with particular emphasis on the exploration of constraint handling techniques to handle constraints on inter-service dependence and conflict. Constraint handling techniques are critical for an efficient algorithm. The three most commonly used constraint handling techniques by GAs are: the penalty function method, the repairing method, and the hybrid method. These techniques were employed to handle constraints on inter-service dependence and conflict, resulting in three new GAs. We then compared the performance of the three GAs and the existing integer programming-based method. In addition, the correctness of the results generated by the GAs was also verified on real world web service-based applications.

Chapter 4 studied the QoS-based resource allocation and scheduling for multiple composite web services on hybrid clouds. Two challenging problems were identified. One was assigning resources to each of the tasks in the composite web service. The other was scheduling the allocated resources when each resource could be used by more than one task and may be needed at different points of time. We developed a random-key based genetic algorithm and a cooperative coevolutionary genetic algorithm (CCGA) for the problem. The major feature of the random-key GA was its random-key representation which was used to overcome the feasibility issue involved in the GA representation. The major feature of the CCGA is a cooperative coevolution model to deal with
the increasing complexity involved in the problem. Experiments presented in Chapter 4 demonstrated the good performance of the two GAs.

Chapter 5 investigated the composite web service partitioning problem for decentralised execution. It was formulated as a graph partitioning problem. We presented a new GA to address the problem. The core ideas behind the GA were the penalty function to handle data dependence constraints and control data dependence constraints, and a local optimiser for improving the solution quality of the GA. The chapter presented a comprehensive evaluation model in order to fully test the performance of the proposed GA and existing heuristic algorithms, and also a verification experiment to validate the correctness of the experimental results.

6.2 Major Contributions

This thesis made several noteworthy contributions to QoS-aware web service composition research. It tackled three important and challenging problems in this research field, the QoS-based web service selection problem, the QoS-based resource allocation and scheduling for multiple composite web services on hybrid clouds problem, and the performance-driven composite web service partitioning for decentralised execution problem. This thesis also made several contributions to genetic algorithm research, by introducing new genetic operators and new novel local optimizers, as well as by providing comprehensive evaluation of the
6.2 Major Contributions

new genetic algorithms, which adds substantially to our understanding of using genetic algorithms to address this kind of problem. The detailed contributions to QoS-aware web service composition and genetic algorithms are enumerated and discussed as follows:

1. The QoS-based web service selection problem (Chapter 3)

The QoS-based web service selection problem has been intensively studied by previous research resulting in several QoS-based web service selection algorithms. However, these algorithms had difficulty addressing many real-world QoS-based web service selection problems due to suffering from either or both of two issues: 1) The poor scalability of an algorithm makes it unsuitable for addressing large-scale QoS-based web service selection problems. 2) Ignoring constraints on inter-service dependence and conflict, a practical kind of constraint in a web service-based environment, might lead to the generation of incorrect composite web services. This work enriched the study of the QoS-based web service selection problem by developing new genetic algorithms that successfully tackled these two issues.

Besides the contribution to the QoS-based web service selection problem mentioned above, this thesis also contributed to genetic algorithm research by providing a new knowledge-based genetic operator. This is used by
the hybrid genetic algorithm mentioned in the following section. The new knowledge-based genetic operator will handle constraints on inter-service dependence and conflict. The generic type of constraints on inter-service dependence and conflict are binary constraints, which are constraints that involve exactly two variables. Binary constraints exist in many classic constraint optimisation problems (i.e. the n-queens problem [105]). The knowledge-based genetic operator that we developed can also be applied to handle these constraints. In other words, this work provided a new way for genetic algorithms to address constraint optimisation problems with binary constraints.

Specifically, the key findings of this problem are as follows:

- **A new problem model.** This study enhanced our understanding of the QoS-based web service selection problem, by providing a new problem model with consideration of constraints on inter-service dependence and conflict. The new problem model can also be used as the basis for constructing the simulation model for testing algorithms’ for the QoS-based web service selection problem.

- **Three genetic algorithms for the QoS-based web service selection problem with constraints on inter-service dependence and conflict.**

  We developed three genetic algorithms for the formulated problem.
Effectively handling constraints on inter-service dependence and conflict is a critical issue for successfully addressing this problem. We explored the commonly used constraint handling techniques by genetic algorithms for handling constraints on inter-service dependence and conflict. This resulted in three genetic algorithms: 1) a Penalty-based Genetic Algorithm (PGA) that uses a penalty function method to handle the constraints; 2) a Mini-Conflict Hill-Climbing repairing Genetic Algorithm (MCHC GA) that uses a fast repairing procedure to quickly repair infeasible individuals; and 3) a Hybrid Genetic Algorithm (HGA) that uses a knowledge-based operator to handle the constraints, with a local optimiser to further improve solution quality.

- An intensive evaluation which confirmed good scalability and effectiveness of the three genetic algorithms, as well as demonstrated the improved scalability of the three genetic algorithms compared with the existing integer programming-based method.

(a) The PGA showed an attractive consumption of computation time under real-world conditions. In addition, the solutions found by the PGA were quite good for practical use albeit sometimes suboptimal for the test problems, except those with strong inter-
Conclusions and Future Work

service dependence and conflicts constraints. These findings suggest that the penalty-based genetic algorithm was suitable for addressing large scale QoS-based web service selection problems without overly strong constraints on inter-service dependence and conflict.

(b) The MCHC GA also presented good scalability for large scale QoS-based web service selection problems, even though its computation times were comparatively longer than the penalty-based genetic algorithm. Moreover, the MCHC GA outperformed the PGA in terms of solution quality. In particular, the MCHC GA showed very good capability when handling problems with very strong constraints on inter-service dependence and conflict thanks to the fast min-conflict hill-climbing repairing procedure.

(c) The hybrid genetic algorithm outperformed the two other GAs in terms of solution quality due to the local optimiser in this algorithm. Its computation times were longer than the two other GAs in general, but they were still acceptable for many practical scenarios. As the slowest algorithm among the three GAs, its computation times were much shorter than the integer
6.2 Major Contributions

programming-based method, when the problem were complex, demonstrating that the GAs are more suitable for complex problems than the integer programming-based method.

- Verification experiments that demonstrated the correctness and efficiency of the algorithms we developed. The three GAs were used by a QoS-based web service selection tool in the study. Applying this tool to perform web service selection for real world composite web services demonstrated the correctness and efficiency of the GAs.

2. The QoS-based resource allocation and scheduling for multiple composite web services on hybrid clouds (Chapter 4)

QoS-based resource allocation and scheduling for multiple composite web services on hybrid clouds was a brand new problem. The contributions we made lie in the following two respects: 1) The work addressed the issue of allocation and scheduling resources for multiple composite web services simultaneously. This was a practical scenario in the real world, rather than allocating and scheduling resources for only a single composite web service as was done by past work. 2) In contrast to previous work on the issue of resource allocation and scheduling for composite web services built on a grid-based environment, where there were only single-user and small-scale grid-based resources, this work addressed the issue of resource allocation
Conclusions and Future Work

and scheduling for composite web services on a hybrid-based environment. This was a more complex scenario where there were single-user, small-scale resources known as private cloud services and multi-user, large-scale resources called public cloud services.

According to research to date, the work we developed was the first to use a cooperative coevolutionary genetic algorithm to address a workflow-based resource allocation and scheduling problem. This resulted in a new cooperative coevolution model tailored to the workflow-based resource allocation and scheduling problem. The novelties of the model lie in the following respects: 1) a new problem decomposition method that enabled the solving of a complex problem in parallel using multiple populations, and 2) a new credit assign method that forced the cooperation among multiple populations.

The detailed findings of this problem are as follows:

- **A new problem model.** We provided a precise understanding of the QoS-based resource allocation and scheduling for multiple composite web services on hybrid clouds by developing a new problem model based on the knowledge of resource allocation and scheduling theory.

- **Two novel genetic algorithms for the problem.** This study found two novel genetic algorithms for this problem: a Random-
key Genetic Algorithm (RGA) and a Cooperative Coevolutionary Genetic Algorithm (CCGA). Both contained a modified decoding procedure that considered elasticity of public cloud web services and accommodated multiple concurrent usage. This could lead to a more effective use of resources on hybrid clouds than traditional methods. In particular, the work used a cooperative coevolution model. This model was used for solving problems with a large solution space. It addressed the problem of resource allocation and scheduling for multiple composite web services. The advantage of exploiting search space parallelism of the CCGA makes the algorithm suitable for addressing large scale problems where the numbers of composite web services and the sizes of composite web services are large. No other algorithms for composite web services resource allocation and scheduling problems have demonstrated this ability.

- *Experiments that confirmed the good scalability and effectiveness of the two GAs.* Thorough practical experiments were performed. The experiments confirmed that both the GAs had good scalability to solve large scale problems and both were effective in finding satisfactory allocation and scheduling solutions in a reasonable time. Moreover, the comparison results of the RGA and the CCGA demonstrated that
the CCGA showed a clear advantage for solving large scale problems, in terms of solution quality and convergence speed. This was due to the cooperative coevolution model used in the CCGA.

3. The performance-driven composite web service partitioning problem (Chapter 5)

Partitioning a composite web service program (i.e. a Business Process Execution Language (BPEL) program) for decentralised execution has the potential to improve the performance of a composite web service. Present heuristic algorithms for BPEL program partitioning suffer from high computational cost for large scale problems. This makes them unsuitable for many real-world BPEL program partitionings. We presented an original algorithm based on the technique of genetic algorithms that successfully overcame this limitation.

The study on this problem also contributed to genetic algorithm research. According to recent literature, there are only a few others that use genetic algorithms to address automatic program parallelisation. Automatic program parallelisation is a generic type of the composite web service partitioning problem. This study added to the knowledge of using genetic algorithms to address automatic program parallelisation. We developed two novel strategies in the GA to handle data dependence constraints and control
dependence constraints. According to research to date, no other GA reports these features. The GA we developed has the potential to address many automatic program parallelisation problems in the domains. These include compiler optimisation for multiprocessor systems and hardware-software co-design of embedded systems.

Specifically, the main findings of this problem are listed as followed:

- **A novel genetic algorithm for BPEL program partitioning.** We presented an original GA to address the BEPL program partitioning problem. This GA contained two new procedures to handle data dependence and control dependence in a BPEL program. In addition, it contained a fast local optimiser to further improve solution quality. Research to date reveals no investigations whereby genetic algorithms are used to address the BPEL program partitioning problem.

- **Experiments revealed the improved performance achieved by the new genetic algorithm.** Experimental results showed that the performance of the GA we developed compared favourably with previous algorithms as the size of the partitioning problem increased. Furthermore, the throughput results it produced were as good as, and sometimes better than, those produced by the slower algorithms.

- **Tests on real composite web services applications proved the**
Conclusions and Future Work

correctness and efficiency of the GA we developed. A centralised composite web service was partitioned in different decentralised ways and deployed on parallelising composite service execution servers, based on the partitioning plans found by this algorithm and previous heuristic algorithms. Comparing the throughput of these decentralised composite web services revealed that the composite web services generated by the GA achieved the best throughput. It was better by 10% compared to the decentralised composite web services generated by other algorithms.

6.3 Future Work

QoS constraints were not considered for the QoS-based web service selection problem in Chapter 3. Therefore, the next step of this research will consider the QoS constraints and inter-service dependence, conflicts constraints, and other constraints, in order to apply the web service selection to more practical scenarios. An investigation into the most effective algorithm settings, such as the population sizes and probabilities of genetic operators, will also be beneficial.

Only a single QoS-based objective was considered for the QoS-based resource allocation problem in Chapter 4. For example, either minimising response time or cost. In the future, we will extend the developed GAs to support optimising multiple QoS-based objectives. This will include optimising response and cost
simultaneously, or optimising response time, cost and other QoS attributes, such as throughput and availability. The cooperative coevolutionary genetic algorithm’s performance might further be improved by addressing the well-known diversity issue involved in every cooperative coevolution model. Another research direction is to investigate Pareto-front approach [64] for problems with multiple QoS-based optimisation objectives. In addition, replacing GAs with particle swarm optimization (PSO) [56] for QoS-based resource allocation problems is also an interest of our future research.

Finally, for the program partitioning program from Chapter 5, we will enhance the algorithm to consider merging fixed nodes such as when multiple fixed nodes are placed in the same partition. Another research direction is to design a knowledge-based operator, in order to further improve the search efficiency of the new GA.
Bibliography


conference on Genetic and evolutionary computation, pages 1069–1075, New York, NY, USA, 2005. ACM. 10, 66


[54] Yaochu Jin. *Knowledge incorporation in evolutionary computation*. Springer Verlag, 2005. 49


