Agent-Based QoS Management for Web Service Compositions

A thesis submitted for the degree of

DOCTOR OF PHILOSOPHY

Xuan Thang Nguyen B.E. (Hons.), Dip.Prac,

Swinburne University of Technology

Jun, 2008

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; and, any editorial work, paid or unpaid, carried out by a third party is acknowledged.

Xuan Thang Nguyen
Faculty of Information and Communication Technologies
Swinburne University of Technology
Jun, 2008

Acknowledgments

I would firstly like to thank my supervisor, Professor Ryszard Kowalczyk, for his support and guidance throughout my candidature. To me, he has been an excellent mentor and friend. His constant encouragement and positive attitude were the main driving forces that kept me motivated. My thanks also to Professor Jun Han for agreeing to act as an associate supervisor. His comments and suggestions were always helpful.

My warm thanks to my research colleagues in the Intelligent and Multi-Agent System group for making the work environment more friendly and enjoyable. Thanks to Jian Feng Zhang and Tino Schlegel for countless hours spent discussing intelligent agents and constraint satisfaction techniques. I am also grateful to Mohan Baruwal Chhetri, Juliana Wong and Kathie Farn for proof reading this thesis and their valuable and constructive feedbacks. A special thank to my girl friend, Khoi Anh Phan, for her love, her patience and support during the difficult time that I had been through.

Finally, I am indebted to my parents and my sister for many years of hard work they have made to support me. Without them, this thesis would never have started.

Credits

Portions of the material in this thesis have previously appeared in the following publications:

1. X. T. Nguyen, R. Kowalczyk, On Solving Distributed Fuzzy Constraint Satisfaction Problems with Agents. IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-07) , Silicon Valley, USA . (to appear)

2. J. F. Zhang, X. T. Nguyen, R. Kowalczyk, A Graph-based Multi-agent Planning Algorithm with QoS guarantees. IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-07) , Silicon Valley, USA (to appear).

3. X. T. Nguyen, R. Kowalczyk, M. B. Chhetri, and A. Grant, (2005), WS2JADE: A Tool for Run-time Deployment and Control of Web Services as JADE agent services. In M. Calisti, M. Klusch and R. Unland (Eds.) Software Agent-based Applications, Platforms and Development Kits. Whitestein Technologies AG, pp. 223–252.

4. J. F. Zhang, X. T. Nguyen, R. Kowalczyk, Graph-based Multi-agent Replanning Algorithm. The 6th International Joint Conference on Autonomous Agents and Multi Agent Systems (AAMAS 2007), May 2007, Honolulu, Hawaii (USA).

5. X. T. Nguyen, Adaptive QoS Management for Web services using Asynchronous Distributed Searches, pp. 37–42, In Proceedings of the IBM PhD Symposium at the 4th International Conference on Service Oriented Computing (ICSOC-06), USA, December 4-7, 2006

6. X. T. Nguyen, R. Kowalczyk, and J. Han, Using Dynamic Asynchronous Aggregate search for Quality guarantees of multiple Web services Compositions, pp. 129–140, In Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC-06), USA, 2006

7. X. T. Nguyen, R. Kowalczyk, K. A. Phan, Dynamic Asynchronous Aggregate Search for solving QoS compositions of Web services, pp. 142–145, SERCOMP06 workshop in conjunction with IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-06) .

8. X. T. Nguyen, R. Kowalczyk, An agent based QoS conflict mediation framework for Web services compositions, pp.522–528, IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-06).

9. X. T. Nguyen, R. Kowalczyk, K. A. Phan, Modelling and Solving QoS Composition Problem Using DisCSP. pp.179–195, In Proceedings of the 3rd International Conference on Grid Service Engineering and Management (GSEM06), Erfurt, Germany, 2006.

10. X. T. Nguyen, R. Kowalczyk, M. T. Phan. Modelling and Solving QoS Composition Problem Using Fuzzy DisCSP , pp. 55–62 , In Proceedings of the IEEE International Conference on Web Services (ICWS06), 2006.

11. B.Wu, J.Y. Zhang, M. Chhetri, J. Lin, S.K. Goh, X.T. Nguyen, I. Mueller, E. Gomes, L. Zheng, J. Han, R. Kowalczyk (2006). Adaptive Service Agreement and Process Management. pp. 927–930, In Proceedings of the IEEE International Conference on Web Services (ICWS06), 2006.

12. J. Yan, Y. Yang, R. Kowalczyk and X. T. Nguyen, A service workflow management framework based on peer-to-peer and agent technologies, pp. 373–380, In Proceedings of the Fifth International Conference on Quality Software (QSIC 2005), Melbourne (Australia), September 2005, IEEE Computer Society Press, 2005.

13. X. T. Nguyen, R. Kowalczyk, Enabling Agent based management of Web Services with WS2JADE, pp.407–412, In Proceedings of the 5th International Conference on Quality Software (QSIC 2005), Melbourne (Australia), September 2005, IEEE. Computer Society Press, 2005

14. X. T. Nguyen, R. Kowalczyk, WS2JADE: Integrating Web Service with Jade Agents, pp.15–22, In Proceedings of the Workshop on Service-Oriented Computing and Agent-Based Engineering (SOCABE2005) in conjunction with 4th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS2005), Utrecht, the Netherlands, July 25-29, 2005.

15. X. T. Nguyen, Demonstration of WS2JADE, pp.135-136, In Proceedings of the 4th International Joint Conference on Autonomous Agents and Systems (AAMAS2005), Utrecht, the Netherlands, July 25–29, 2005

Note

Unless otherwise stated, all fractional results have been rounded to the displayed number of decimal figures.

Contents

List of Figures

List of Tables

Abstract

Service oriented computing (SOC) is fast becoming the core platform for software applications and integrations. In SOC, applications are modeled as collections of loosely-coupled, interacting services that communicate using standardised interfaces and protocols. These services can be combined together to form new value-added services, or compositions which potentially span across many administrative boundaries.

Managing the quality of Web service compositions is important. However, it is difficult in practice. The difficulty lies in the complexity of composition dependencies such as sharing computing resources or component services. The distributed nature of Web services presents another challenge where a service composition may involve service providers from different organizations and administrative domains. A viable collaboration between these service providers is necessary for any successful management.

This thesis studies the Quality of Service (QoS) management for multiple service compositions. Despite many approaches having been proposed for the management recently, those approaches are limited to an individual composition and management functionalities executed by a single organization. This thesis aims at providing solutions for those limitations. It has the following contributions:

- A new framework supporting a distributed approach to solve the QoS management problem for multiple inter-related compositions is proposed. This problem has not been well investigated in the current Web service research. In the thesis, fresh ideas from Distributed Constraint Satisfaction (DisCSP) research are employed and applied practically to the Web service environments.

- Extensions and improvements to existing DisCSP algorithms are suggested. This enables existing DisCSP algorithms to be used practically in the framework. In the adaptations and improvements, existing DisCSP algorithms are reviewed. Their possible limitations when being used in the Web service environments are analysed and addressed adequately.

- New DisCSP algorithms to solve the QoS management problem are proposed. These newly proposed algorithms can model different preference levels of service providers and work in dynamic environments where compositions and constraints on QoS variables can be added or removed. Optimization of preferences of service providers can also be achieved by using some of the algorithms.

- A novel verification mechanism, for checking the conformance of service providers to a DisCSP algorithm, is developed, and thus allowing the algorithm to be used practically in an untrusted environment such as that of the Web services.

Chapter 1

Introduction

Enterprise software integration is an important factor for businesses to expand their collaboration cycle as well as to enrich their capabilities. Web services [W3C] have emerged during the last decade as a leading interoperable technology to offer a solution for the integration problem. Web services have been widely believed to simplify enterprise software integration whilst creating a level of flexibility and scalability which is difficult to achieve with other integration methods [Sta02].

A Web service, defined by the W3C Web Services Architecture Working Group [W3C], is "a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its descriptions using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards" [W3C]. The W3C Web service architecture is supplemented by a number of specifications including Simple Object Access Protocol(SOAP) [GHM+07], Web Services Description Language (WSDL) [W3C05b] and Universal Description, Discovery and Integration(UDDI) [OAS07]. They are defined as follows:

- SOAP is a protocol for exchanging XML [1] based messages. Theoretically, SOAP can use different underlying transport protocols like TCP, HTTP, or SMTP to execute

[1] XML is the Extensible Markup Language [W3C07a].

services on a remote server.

- WSDL is an XML based language that specifies how a Web service can be described. The description of a Web service contains information such as how to connect to and interact with the service. It also has the details of specific transport protocols supported by the service.

- UDDI is a registry that allows Web services to be published and discovered by service consumers. UDDI is defined as a special Web service.

SOAP, WSDL, and UDDI are major factors that have contributed to the popularity of Web services so far. SOAP can use HTTP as a transport protocol, and hence relieve the difficulty of communication between different organizations. WSDL allows those organizations to encapsulate and expose their software through Web service interfaces using a standard language that everyone understands and agrees on. Finally, UDDI enables organizations to find each other's Web services. SOAP, WSDL and UDDI together provide necessary support for software integration at the inter-organizational level.

Within the topic of software integration, service composition has been the subject of a considerable amount of both practical and theoretical interest. Web service composition refers to the process of creating a new Web service from a set of existing Web services. The process involves assembling and executing the existing Web services in a particular order with the goal of fulfilling some complex task. In this way, a composition can enable valuable functionalities that may be difficult or impossible to achieve with any individual service alone. With compositions, Web services can be considered as the building blocks for more complex enterprise applications. A popular example of Web service composition is the Travel Reservation application [W3C07b]. In this example, a travel agency provides complete vacation packages which include transport tickets, hotel reservation, car rental, and excursion booking for tourists. The travel agency offers the packages by combining existing services from different providers such as airlines, bus companies, and hotel chains. These services are likely to be written in different programming languages and run on different hardware platforms. Nevertheless, they can be composed together through their

Web service interfaces and appear as a single service to end-users.

Since Web services are playing an increasingly important role in software integration, the need for Web service management has never been greater. A realistic Web service should meet both functional and non-functional (or quality of service) requirements of its clients. Functional requirements describe operations and behaviours of a service whereas QoS requirements specify QoS attributes. For the travel reservation example above, the functional requirements for the vacation packages includes presenting an itinerary to a user upon his/her request, and QoS requirements include the time limit for completing the reservation of the trip online. QoS requirements are important for any service in general. If there are multiple services providing the same functionality available to a user, the user's preference to a service will depend upon the service's QoS properties. Management of Web services can be classified according to functional and QoS management. The functional management ensures that a Web service executes correctly whereas the QoS management guarantees that the quality of the Web service meets its users' expectation.

This thesis focuses on the QoS management of Web service compositions. The promising prospects of Web services in software integration has its share of challenges in QoS management of service compositions:

- Management of a composite Web service is difficult because an effective management system is expected to operate across organizational boundaries. This is due to the participation of different organizations in the same composition for which each organization provides a component service. The management system must work with different administrative domains.

- Trust and privacy can be major issues if a centralized management system (e.g. QoS broker) is used. In such a case, every participating organization has to delegate the management task to the management system. It may require an organization to open up part of its internal system for an external access that is typically undesirable. Also a centralized management system may have the problem of scalability if too many services are concurrently under the management.

- Web service composition can be a complex process that can be done not only at design time but also at runtime with different "just-in-time" or "on-the-fly" composition (as opposed to compile or design time integration) techniques. A management system should also be automatic and flexible enough to handle those complex compositions.

- The Web service environment is highly dynamic and experiences high uncertainty with regular changes from businesses and organizations. An effective management system should take these dynamic properties into its consideration.

- Multiple inter-related compositions typically re-use the same component services or share the same underlying resources. These compositions are dependent on each other's performance as well as resource consumption. Consequently, managing these compositions individually would be less effective than managing them as a whole, taking into account the inter-relations.

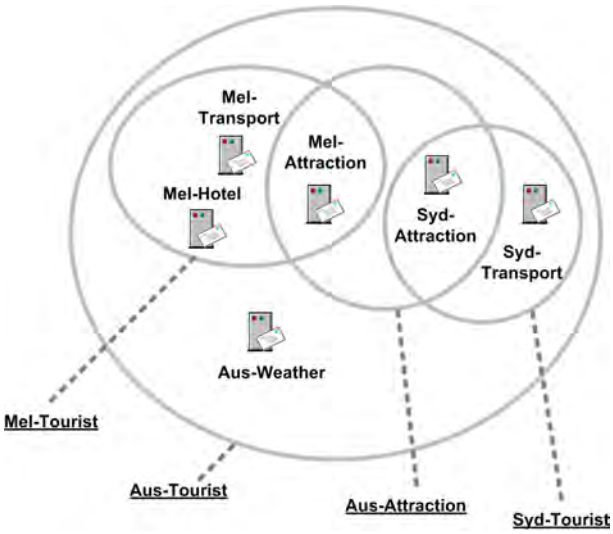


Figure 1.1: An example of multiple inter-related service compositions. Each composition is surrounded by a circle. The compositions are inter-related by having some common component services. Managing one composition may affect other compositions.

Among the challenges listed above, the last one is particularly significant. Service orientation enables virtualization of resources. Virtualization means the ability to have multiple

heterogeneous resources appear as one homogeneous entity to the user [OAS06b]. The virtualization process hides many complex resource dependencies from the user. A number of Web services appearing independently at the service layer may have dependencies which are invisible to the user. The dependencies can be important to the management of those Web services. Figure 1.1 shows an example of multiple inter-related service compositions as an extension of the previous example on travel reservation. In the figure, four composite services: Mel(bourne)-Tourist, Aus(tralia)-Tourist, Syd(ney)-Tourist, and Aus-Attraction are made up from six individual services: Mel-Transport, Mel-Hotel, Mel-Attraction, Syd-Transport, Syd-Attraction, and Aus-Weather. A management activity carried out within a composition (e.g. replacing Mel-Tourist in Mel-Attraction) may affect other compositions (e.g. Aus-Attraction and Aus-Tourist), therefore it is necessary for the managers of those compositions to collaborate.

Despite there being a large number of approaches (e.g. [YL04; JCZ03; PS03; PEWS03; LNZ04]) proposed for QoS management, those approaches share two major limitations. Firstly, they are limited to the management functionalities carried out by an individual organization. Secondly, they tend to rectify QoS violations of a single composition. Those approaches together with their limitations will be detailed in the next chapter.

This thesis aims to address the above limitations by employing techniques from the field of Distributed Constraint Satisfaction. In parallel to the advancement of Web services, distributed constraint satisfaction has attracted a great amount of interest from the MultiAgent Systems (MAS) and Artificial Intelligence (AI) communities in the past few years [Yok01]. Similar to the Web service technologies, distributed constraint satisfaction has been introduced to solve problems emerging in distributed environments. However, in contrast to Web services that address the problem of infrastructure interoperability, distributed constraint satisfaction focuses on solving conflicts arising in distributed settings. More formally, a distributed constraint satisfaction problem (DisCSP)[2] is a problem with a finite number of variables, each of which has a finite and discrete set of possible values and a set of constraints over the variables. These variables and constraints are distributed

[2]In this thesis, Distributed Constraint Satisfaction is referred to as DisCSP instead of the more common DCSP. This is to distinguish it from Dynamic CSP, which is referred to as DynCSP.

among a set of autonomous and communicating agents. A solution in DisCSP is an instantiation of all variables such that all the constraints are satisfied. This thesis will show that major tasks in QoS management can be modeled as instances of DisCSP. Existing distributed constraint satisfaction algorithms will be adapted, and new algorithms will be developed for the QoS management of Web services in the thesis.

1.1 Research Questions

This thesis studies the QoS management of service compositions. In particular, it investigates the feasibility of a distributed management approach in which constraint satisfaction techniques are employed. The following main research questions are pursued in this work:

1. Can a fully distributed management mechanism[3] be used for QoS management of composite services instead of a centralized one ? Centralized mechanisms for Web service management have been widely studied in the context of QoS composition (e.g. [ADK+05; Men04; GNCW03]). Those techniques use a centralized QoS broker which has the problems in trust, privacy, and scalability, as discussed previously.

2. How can the management problem of multi-interrelated compositions be solved ? Whilst the relationships between the compositions are important and useful for any management system, service oriented architectures hide them away from the service level. This creates difficulties for the management system in analysing and using the relationships to manage the compositions. In addition, to the best of our knowledge, no existing work on QoS management has considered such relationships to date.

3. Can a distributed QoS management problem be expressed as a constraint satisfaction/optimization problem ? The requirements of service clients and the resource dependencies between compositions can be considered as constraints in the constraint satisfaction/optimization frameworks. In this context, some distributed constraint satisfaction approaches have focused on the privacy aspect of distributed environments [YDIK92] in the last few years. We consider adapting existing distributed

[3] A management mechanism which is executed concurrently and distributively by peers.

constraint satisfaction and constraint optimization algorithms or developing new ones for QoS management. These algorithms should be able to address problems such as computational complexity and dynamism of service compositions.

4. What are the common limitations of existing distributed constraint satisfaction/optimization algorithms when applying them to the Web service management problem; and how can those limitations be addressed ? Distributed constraint satisfaction may make some assumptions which are impractical in the Web service environment. New solutions should be proposed to avoid those assumptions or to address them adequately.

1.2 Research Contributions

The main objective of this thesis is to: *develop a new framework for coordinated provision and adaptive control of end-to-end QoS of Web service compositions during execution.* The main focus is on new management algorithms for *multi-interrelated service compositions* and a *distributed* management mechanism in which different organizations involved in the compositions can take part. Execution monitoring to detect QoS violations is important however it is not in the scope of this thesis. Some work on QoS monitoring can be found in [BTPT06; MS05].

The contributions of this thesis are as follows:

- Provision of a new *distributed framework* for QoS management of multiple related service compositions. Management of multiple compositions has not been well investigated in the current Web service research. This thesis uses fresh ideas from the field of distributed constraint satisfaction and practically applies them to the Web service environment. In this new framework, different service providers can collaborate and manage the QoS of the compositions together, using distributed algorithms. The framework is provided with:
  - A detailed architectural pattern which describes important components in the

framework. This architectural pattern serves as a reference for any implementation of the framework.

  - A pair of algorithms to find implicit QoS constraints induced from functional compositions of Web services. These constraints are important for the QoS management. They express the relationships between the end-to-end QoS of a composition and the QoS of the component services in the composition.

- *Adaptation of existing distributed constraint satisfaction algorithms* so that they can be used for the QoS management. Specifically, the algorithms (Asynchronous Backtracking Search and Asynchronous Aggregate Search) are investigated for their limitations when being used in the Web service environments. These limitations such as the assumptions of one variable per agent and First In-First Out (FIFO) communication channels, are analysed and addressed in detail. As a result, the algorithms can be adapted and used practically in the Web service environments.

- *Development of new distributed constraint satisfaction algorithms* including Dynamic Asynchronous Backtracking Search (DynABT) and Fuzzy constraint solving using Asynchronous Backtracking (FABT) and their use for the QoS management. DynABT allows Web services to be dynamically added into or removed from the management scope whereas FABT aims at maximizing the quality of a management solution according to different service providers' peferences.

- *Development of a novel verification mechanism* for checking the conformance of service providers to a DisCSP algorithm (ABT or AAS), and thus allowing the algorithm to be used practically in an untrusted environment such as that of the Web services.

1.3 Thesis Structure

The thesis is organized as follows:

- Chapters 2 and 3 provide an overview of the areas related to this thesis. QoS management tasks are explained and elaborated in detail, together with the background to distributed constraint satisfaction.

- Chapter 4 proposes a distributed management framework and describes various components in the framework. This is an important chapter that links different chapters in this thesis together.

- Chapter 5 focuses on constraints at the Web service level. It presents a method to capture the constraints which are important for the final management tasks.

- Chapters 6, 7, 8 and 9 present extensions of existing DisCSP algorithms to handle dynamic changes in the Web service environment. They also provide mechanisms to optimize the joint function of service providers' preferences. The practicability of using DisCSP algorithms is investigated in these chapters.

- Chapter 10 focuses on the implementation of WS2JADE (Web service to Jade) toolkit. This software toolkit allows for integrating Web services and software Agents. It is a useful supporting component for any agent based implementations of the proposed framework.

- Finally, this thesis concludes with Chapter 11 in which the main contributions of the thesis are summarised and possible directions for future research are discussed.

Chapter 2

QoS Management Background and Related Work

2.1 Introduction

Service Oriented Computing (SOC) paradigm is changing the Internet. In that paradigm, services are utilized as the basic constructs to support the development of complex distributed software and applications[1] [Sta02]. Consequently, SOC changes the way that software is designed, integrated, distributed and consumed [W3C]. The first chapter of this thesis introduces the service composition as an important mechanism for creating flexible business processes and applications from simpler services. This chapter extends the previous one by discussing different methods for composition. It then examines a list of popular QoS parameters for Web services. Finally, a survey of existing approaches for QoS management of service compositions is presented together with their limitations.

2.2 Service Oriented Computing and Service Compositions

Services are often regarded as a natural progression from component based software [Sta02]. The visionary promise of SOC is a world of services where application components are

[1] In this thesis, we use the word *service* to refer to the general abstraction of service in SOC model and *Web service* as a specific realisation of SOC in practice.

exposed with little effort into a network of services. These services can be assembled to create flexible enterprise software and applications that span over the boundaries of different organisations. Service orientation also permits a loose coupling between the interface and implementation of a service and hence simplifies interconnections to legacy IT assets. This loose coupling offers flexibility in integrating software systems since it reduces the risk that a change in one implementation module in one organization will force a change in another module in a different organization. In addition, even if these organizations use incompatible implementation technologies, their services can still be joined together on demand to create composite services, or disassembled easily if required.

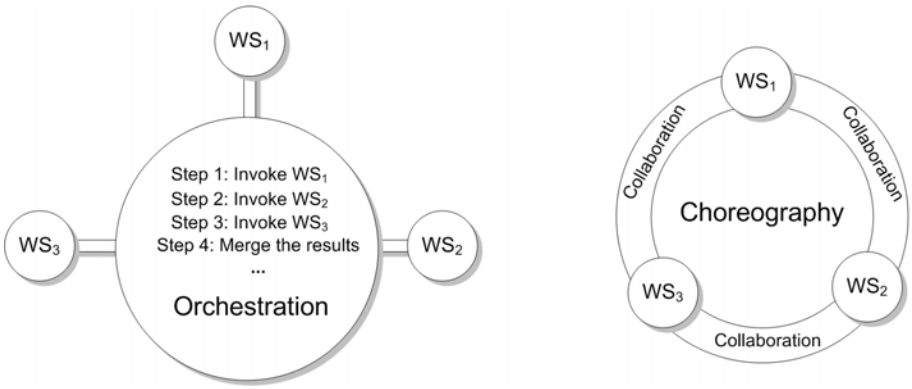


Figure 2.1: Orchestration vs Choreography

A Web service composition can be considered as a *choreography* or an *orchestration* from different viewpoints. A *choreography* describes a composition from the global viewpoint of all participants (i.e. Web service providers who participate in the composition) whereas an *orchestration* has the local viewpoint of a single provider. For this reason, orchestration is often associated with private processes that are managed and composed by a single organization. It always represents control from one party's perspective. Choreography allows more collaboration where each party involved can describe its part in the interaction. The distinction between the two is illustrated in Figure 2.1. Despite this difference, choreography and orchestration together provide a standard approach for con-

necting Web services. While Web service orchestration has enjoyed popularity with an increasing number of support tools, languages (e.g. WS-BPEL, XLANG, WSFL), and implementations [Aal03], Web service choreography standards have emerged later with the replacement of WSCI/WSCL [W3C05a] by WS-CDL [W3C06].

2.3 Quality of Web services

While service functionalities tell a customer which service can be used to perform certain tasks, non-functional aspects such as response time and availability indicates how much better one service can be rated over others. These non-functional qualifiers in general are referred to as Quality of Service (QoS). Services (or applications) at different Internet layers are concerned with different types of QoS. Traditionally QoS at the network layer includes parameters like packet error rate, delay, and jitter [BCS94; SH04]. Most research have focused on guaranteeing and improving these parameters [SZ99; RLLS97]. At the service layer, a number of different QoS parameters including response time, reliability, availability, and capacity have been popular in literature. The following list presents frequently used QoS parameters in the context of QoS management for Web services, together with their popular definitions [Men04; Men02; W3C03; LNZ04].

- **Throughput**: The number of Web service requests served in a given time interval.
- **Response time**: The time required to complete a Web service request.
- **Latency**: The round-trip delay (RTD) between sending a request and receiving the response.
- **Execution time**: The time taken by a Web service to process its sequence of activities.
- **Reliability**: The ability of a Web service to execute a service and to complete the execution during a contracted period. The overall reliability measure of a Web service is related to the number of failures for a specified duration (e.g. per day, week, month, or year).

- **Capacity**: The numerical limit of simultaneous requests which can be provided with guaranteed performance.
- **Robustness**: The degree to which a Web service can function correctly even in the presence of invalid, incomplete or conflicting inputs.
- **Accuracy**: The error rate generated by the Web service.
- **Security**: The availability of existing authentication, authorization, confidentiality, traceability/auditability, data encryption, and non-repudiation for a Web service.

The above list is by no means complete. Also for the same QoS parameter, its definition may slightly change depending on the application scenario. Many QoS parameters have discrete domains in real life applications. All QoS parameters listed above except robustness and security can be considered as having discrete domains. For example, the response time, execution time and latency are often measured in units of milliseconds or nanoseconds. Even with *robustness* and *security*, we can still approximate the QoS values and map them into a discrete set as long as the precision level is acceptable. This is important in order to use DisCSP techniques since they often require variables with discrete domains.

2.4 Management of Quality of Web services

Web services have been viewed as an abstraction of existing computing resources [Sta02]. On the surface, the management of Web services is thus the management of the computing resources. However, there are new requirements for the latter as compared to the former. Classical management systems at the resource level often do not operate across organizational boundaries. On the contrary, with interoperability, Web services are designed to interconnect applications from different administrative domains. Consequently, most effort in the Web service research community has been on distributive related aspects of the management such as distributed service discovery, dynamic service binding, and service composition. In particular, for QoS management of compositions, the following tasks have been the focus of many research [Men04; Men02; W3C03; LNZ04; ADK+05; GNCW03**?**]:

- **Task 1 - QoS composition:** The QoS composition is an aggregation of QoS values supported by component services. It ensures that a set of pre-defined requirements can be satisfied. The task can be associated with an optimization objective such as minimizing the cost of the composition. It is often carried out at the beginning, when the composition is first formed, and consists of three sub-tasks:
  - **Task 1.1 - QoS Discovery:** If the composition is a choreography, the component service providers discover the QoS support capabilities of each other during this sub-task before collaborating and forming the final composition. Otherwise, if the composition is an orchestration carried out by a single organization (i.e. the composer) then the task refers to the activity of searching for component services. The search is performed by the composer and returns a list of component services that provide the same functionalities but have different QoS levels.
  - **Task 1.2 - Ontology Translation:** Component services can belong to different organizations and therefore may use different ontologies. In this case, a translation of QoS ontologies used in these services is necessary for the QoS aggregation for the final composition.
  - **Task 1.3 - Service Selection:** This is the most important sub-task. The task is to solve a combinatorial problem where services must be selected from a list of candidates from Task 1.1 so that the requirements on the QoS of the composition can be satisfied.
- **Task 2 - QoS Monitoring:** This monitoring task is to detect events which may lead to violations of the clients' QoS requirements. Such events include any changes in the surrounding environment or abnormal behaviour of component services during their execution. Depending on the design of a monitoring system, the list of interesting events may vary. A proactive monitoring system looks for early symptoms of a problem; whereas a reactive monitoring system only detects service failures or violations which have already become apparent.

- **Task 3 - QoS Adaptation:** This task is for QoS attainment. The management system responds to changes detected by the monitoring component so that all constraints (especially the clients' QoS requirements) can still be satisfied. The task is often carried out by adjusting the resource used by the services in order to maintain the required QoS. Sometimes this task is associated with an optimization objective such as minimizing the adaptation time.

The framework proposed in the next few chapters of this thesis provides methods and means to complete Task 1.2 and Task 3. These are important tasks as they represent the decision-making capability of a QoS management system. It is assumed that QoS discovery, ontology translation and QoS monitoring are handled by external components that can be integrated into the proposed framework. Such external components can be found in [D'A06], [TRPA06] and [BTPT06].

2.5 A survey of QoS management for Web services

During the last few years, extensive research and work have been carried out for QoS management of Web services [LNZ04; ADK+05; RAC+02; POSV04]. These work can be classified into three different categories: specifications, frameworks, and algorithms to support the QoS management. The scopes of these work vary. While some research developed new tools or focused on selected aspects of QoS management (e.g. [LNZ04]), others attempted to cover the whole spectrum of QoS management (e.g. [wsl04; PSL03]). This section briefly overviews important work in each category.

2.5.1 Specifications for service management

Since Web services have been proposed to solve the organization-wide and cross-enterprise integration challenges, Web service based management has been emerging as a promising approach for managing Web services themselves. By exposing the management capabilities as Web services, various management technologies can be integrated into a cohesive whole. Aligning with this direction, many important Web services-based standards have been developed. Among them are WS-Agreement [Ope03], Web Service Notification

(WSN)[OAS06a], Web service Distributed Management (WSDM) [Hew04c], and Web service Management (WS-Management) [Dis04]. These standards provide instrumentations to assist in the QoS management tasks and have been successfully employed in commercial applications.

SLA and WS-Agreement

A key concept used in many management approaches is Service Level Agreement (SLA). A service level agreement is a formal contract concerning service provision and contains the terms of responsibility for a service provider and a service consumer. These terms should be unambiguously and precisely defined. In the context of QoS management, a service level agreement provides the means for defining, contracting and monitoring the QoS of a Web service [Ope03]. For instance, a service provider and a service consumer can have an agreement on different performance criteria. A management system monitors the real performance and compares it against the criteria to determine whether the contract is violated or not.

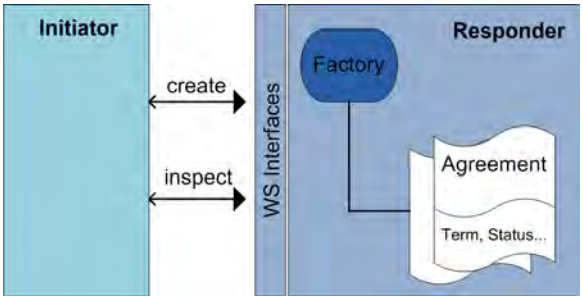


Figure 2.2: Interactions between an initiator (e.g. a client) and a responder (e.g. a Web service provider) in WS-Agreement.

The WS-Agreement specification [Ope03] proposed by Global Grid Forum (GGF) specifies a Web service protocol for establishing an agreement between a service provider and a service client. WS-Agreement uses XML to describe contracts and contract templates. The contract creation, monitoring, and expiration are managed through a Web service interface as shown in Figure 2.2. Upon receiving a contract creation request, the agreement factory will create a new contract template and replies with an endpoint reference to the

initiator. Based on this endpoint reference, the initiator uses the Web service protocol to update or inspect the contract.

Web service Notification

Web service Notification [OAS06a] enables event-driven interactions between Web services. An event-driven interaction pattern is especially useful to aid the QoS monitoring task. Within an event-driven model, an event is generated by an event generator and consumed by event listeners. Listeners are notified by generators about new events. A monitoring system used in QoS management is an event listener. This listener can use a "pull" or a "push" mechanism to continuously collect events that are important to the management tasks (e.g. events of service failure or contract violations). With the "pull" mechanism, the monitoring system must contact the event generators to query and collect events periodically. With the "push" mechanism, the monitoring system requests that the event generators notify it whenever an interesting event occurs. The "push" mechanism offers an advantage over the "pull" mechanism since the monitoring system does not consume resources unnecessarily by periodically checking for new events. Web service Notification can be used for the implementation of both "push" and "pull" mechanisms using Web service protocols.

The Web service Notification specification [OAS06a] has three main parts: WS-BaseNotification, WS-BrokeredNotification, and WS-Topics. Together these provide a standardized way for event producers to disseminate information to a set of Web services. These Web services extend a "Publish/Subscribe" interface so that the event producers can publish their events to these Web services directly, or through an event broker, without any prior knowledge of the Web services. Details of the "Publish/Subscribe" interface can be found in [OAS06a].

WS Distributed Management and WS-Management

Hewlett-Packard (HP) initiated the Web Services Management Framework (WSMF) which is a logical architecture for the management of resources [Hew04c]. Extending this work further, HP together with other companies have released the Web Services Distributed Management (WSDM) specification. WSDM has two parts: Management Using

Web Services (MUWS) [Hew04b] and Management of Web Services (MOWS) [Hew04a]. MUWS defines how the management of any resource can be accessed via Web service protocols, whereas MOWS specifies how Web services can be managed using Web services. The specification has been submitted and accepted as a standard by the Organization for the Advancement of Structured Information Standards (OASIS). Microsoft, in collaboration with various IT companies has released its own SOAP-based protocol for managing systems (including Web services), called Web Services for Management (WS-Management) [Dis04].

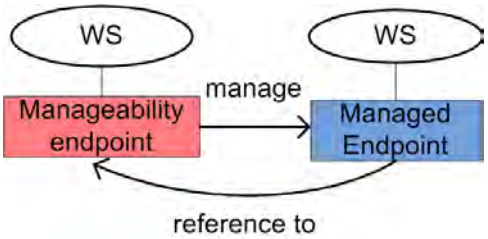


Figure 2.3: In WSDM and WS-Management, a managed Web service has its management capabilities exposed as a Web service. This Web service is called the manageability Web service. The managed Web service keeps a reference to its manageability Web service.

Both WSDM and WS-Management focus on manageable resources. They have in common the way that a manageable resource is identified and communicated. In both specifications, the management capabilities for a manageable resource are exposed as a Web service. In other words, these capabilities are accessible through a Web service endpoint. Such an endpoint is called a *manageability* endpoint. An implementation of this manageability endpoint provides the capability to retrieve and manipulate the manageable resource.

By considering Web services as manageable resources, Web services can also be managed in WSDM and WS-Management. These Web services are referred to as *managed* Web services. Managed Web services have operations that allow their manageability endpoints to be discovered. Figure 2.3 illustrates the relationship between a managed Web service and its manageability endpoint.

2.5.2 Frameworks for QoS management of Web services

There have been a large number of frameworks proposed for Web service management [JCZ03; PS03; PEWS03; RD05]. A major framework proposed by IBM for specifying and monitoring the SLA of Web services is Web service Level Agreement (WSLA) [wsl04]. A prototype of the WSLA framework is implemented in the IBM Emerging Technologies Tool Kit, which is available on the IBM alphaWorks Web site [IBM04]. The framework consists of an XML based language to describe SLA and a runtime architecture comprising several WSLA services. Some important services are SLA Establishment, Deployment, Measurement, Condition Evaluation, and Management services. The SLA Establishment service enables SLA negotiation and establishment. Using this service, a customer can retrieve the QoS levels supported by a Web service and make his/her QoS request. If the provider of the Web service approves the request, a SLA document comprising the relationships and obligations of the service provider and consumer is created. The SLA deployment service then checks and distributes the SLA document, either in full or in part, to the involved parties. The Measurement service stores the configuration information of

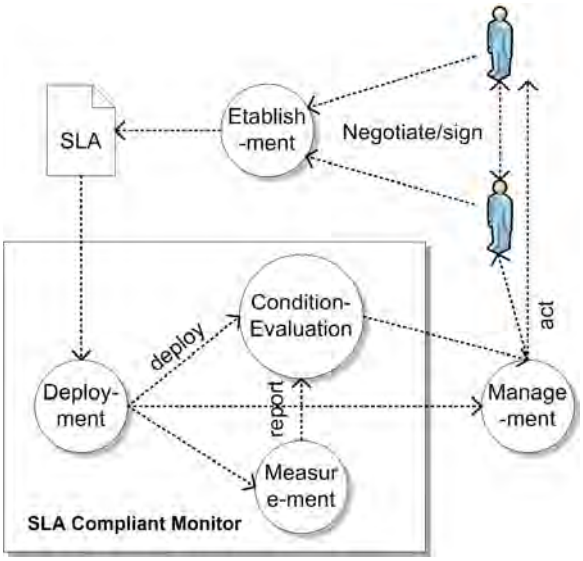


Figure 2.4: In the WSLA framework, different services are used for automated SLA monitoring and management [wsl04].

the current system as well as runtime information on the metrics that are defined in the SLA. It monitors and measures the real values of QoS parameters at runtime. Those real values are compared against the thresholds specified in the SLA by the Condition Evaluation service. This comparison can be done periodically or upon the availability of new QoS values. Any violation is reported to the Management service. The service then takes the available actions as specified in the SLA to correct the problem.

WebQ is a recent framework proposed by C. Patel et al. [PSL03] for QoS management in service environments. The WebQ architecture depicted in Figure 2.5 consists of different components that enable the QoS management of Web Service. Web Service Mediator is responsible for service discovery. It retrieves a list of Web Services that can perform a given task but offer different QoS guarantees. Monitor is responsible for monitoring and measuring actual values of WebQ QoS parameters. The Intelligent Task Execution Engine manages the execution of Web services. The Expert System provides a set of rules that analyzes the current status of the workflow. The Knowledge Base is a repository of facts about QoS parameters. The Rule Repository stores QoS requirements of users, and elicits the steps to be taken in order to satisfy a specified QoS requirement. In the WebQ framework, five different phases are proposed for QoS adaptive management: Workflow modelling, QoS requirements setting, initialization, execution and monitoring, and dynamic and adaptive Web service selection. The first three phases are for the QoS composition task and the last two phases are for the QoS Adaptation task. During the QoS adaptation, under-performing services, based on some predefined rules, are replaced by new services.

The METEOR-S (Managing End-To-End OpeRations for Semantic Web Services) [POSV04] is a project about workflow management for semantic Web services carried out at the Large Scale Distributed Information System (LSDIDS), University of Georgia. One major focus of the project is end-to-end QoS of Web service compositions. METEOR-S implementation provides a GUI for abstract workflow design, a Web service discovery engine, and a service binder. Based on a set of requirements, a user can use the GUI to create a composite service. METEOR-S then discovers, selects appropriate services and

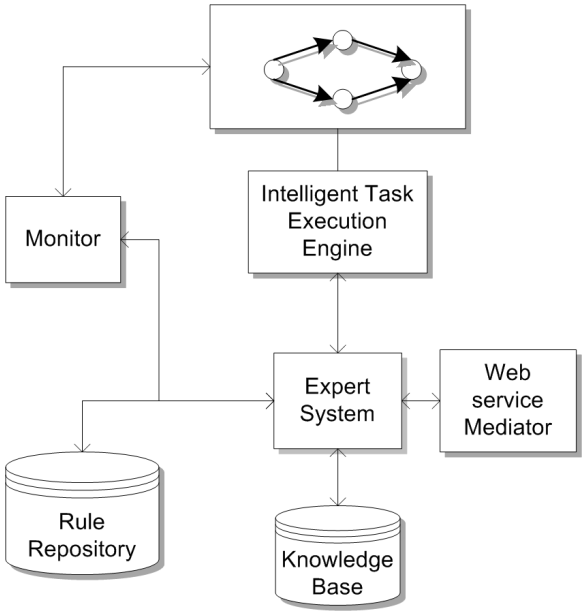


Figure 2.5: Different components in the WebQ architecture [PSL03].

binds them into a final composition. At this stage, METEOR-S only focuses on QoS composition. It is built on top of Managing End-To-End OpeRations (METEOR) and relies on METEOR for other management aspects. METEOR is currently a management system designed primarily for traditional workflows. It remains unclear how METEOR and other capabilities of METEOR-S, besides QoS composition, can be adapted to service oriented environments.

2.5.3 Management Algorithms

A considerable effort has been put into developing new algorithms for QoS management in service environments. Those algorithms are often introduced for QoS composition and adaptation, and are not restricted to any particular framework. Gu et al. proposed an algorithm for QoS-assured service composition in Service Overlay Networks (SON) [GNCW03]. The objective of this algorithm is to find the optimal composition plan, with the assumption that each component in the composition can be fitted with a number of Web services that have different QoS levels. The QoS attributes investigated in the work

are execution duration, reputation, and cost. The algorithm uses the integer programming technique.

In [WVKT06], Wang et al. proposed an algorithm for matching a QoS profile (e.g. a client's request for QoS requirements) with QoS profiles offered by a number of service providers. The algorithm is useful in many QoS adaptation mechanisms (e.g.[JCZ03; PS03; PEWS03]) that replace an underperforming service with a new service. Service matching can be employed in such a QoS adaptation mechanism to select the new service. For each service provider, the algorithm computes a matching level which measures how closely the requested QoS profile and the provider's QoS profile are. This matching level is an aggregation of the similarities between values in the two profiles. The provider that has the highest matching level is selected by the algorithm.

In [YL04], Yu and Lin discussed several algorithms designed for a QoS broker that is responsible for selecting and coordinating service components of a composition. The objective of this algorithm is to maximize the user's utility function value while satisfying the end-to-end delay constraint. The algorithms follow one of two approaches: the combinatorial approach and the graph approach. In the combinatorial approach, QoS composition is modelled as the Multiple Choice Knapsack Problem (MCKP) whereas in the graph approach, it is translated to the constrained shortest path problem. Yu and Lin then use existing solutions for the problem in each approach and apply them to the QoS composition problem.

In [GG05], Guan and Ghose proposed to use constraint hierarchy in dealing with QoS requirements. Particularly, they consider preferences on QoS requirements as soft constraints. A QoS via constraint Hierarchies (QoSCH) model is defined in this study. The steps for selecting an optimal Web service from a set of available Web services are suggested. The study also proposes a Branch and Bound algorithm for service composition. The algorithm can ensure that the actual QoS of a composition do not violate, or deviate minimally from the QoS requirements for the composition.

2.6 Limitations of existing approaches for Quality of Web services management

Whilst many approaches and frameworks have been proposed for solving the QoS management problem as summarized above, there are certain limitations to those approaches. Here we state four major shortcomings of those existing approaches when dealing with the upcoming generation of Web/Grid service applications that are truly distributed and inter-organization oriented:

- Firstly, most of those approaches (e.g. [JCZ03; PEWS03; RD05; ADK+05; LNZ04**?**]) are limited to the management tasks carried out by a centralised QoS broker. This may introduce a single point of failure, thus affecting the management reliability and scalability. In addition, only services in the broker's administrative domain can be effectively managed.

- Secondly, trust and privacy can be an issue since service providers may not be willing to share their private information with others or delegate the managerial tasks of their services to a third party (e.g. the broker). This is especially true in a real dynamic service environment such as those of Web/Grid services, where service providers may not know or trust each other.

- Thirdly, the existing approaches focus on managing QoS of individual compositions separately. This is not sufficient in an environment where complex relationships and dependencies between different service compositions exist. In particular, those approaches lack a mechanism which can analyse and handle inter-composition dependencies in order to manage multiple compositions concurrently.

- Lastly, each service provider may be involved in more than one composition with different and changing QoS requirements from clients. Also, the Web service environment is dynamic by nature and new requirements that lead to new compositions may arrive at any time. Unfortunately, none of the algorithms explicitly and effectively address those issues.

These limitations are addressed in a newly proposed QoS management framework. The proposed framework can manage multiple compositions in parallel as well as handle dynamic changes in the Web service environments. Details of the framework are described in Chapter 4 of the thesis.

Chapter 3

Distributed Constraint Satisfaction - A background

3.1 Introduction

In this thesis, distributed constraint satisfaction is the main technique proposed for QoS management of Web services. This chapter reviews the key concepts and results in the field of constraint satisfaction as a background for the proposed QoS management framework. The chapter is divided into two parts. In the first part, classical constraint satisfaction is defined and different centralized search techniques are discussed. Distributed constraint satisfaction, as an extension of the classical constraint satisfaction, is covered in the second part. The part begins with the motivation behind distributed constraint satisfaction. It then discusses applications as well as current advances in the field.

3.2 Classical Constraint Satisfaction

3.2.1 Problem Definition

The study of constraint satisfaction problems (CSPs) originated in the field of Artificial Intelligence in the 1970s. Since then the importance of the CSP has continued to grow in computer science [Tsa93]. Constraint satisfaction has been found to be a fundamental

problem in many applications including temporal reasoning, resource allocation, planning and scheduling [Mei91; FF99]. Generally, a Constraint Satisfaction Problem (CSP) is a problem comprising a finite set of variables, a finite set of possible values for each variable, and a set of constraints that restricts the values of the variables; the question is to find an assignment of values to every variable so that all the constraints are satisfied. A formal definition of a CSP is given as [RN03]:

Definition 1 [CSP]: A CSP is a triple $\langle \mathbf{V}, \mathbf{D}, \mathbf{R} \rangle$ where:

- $\mathbf{V}$ is a set of variables: $\mathbf{V} = \{v_1, \ldots, v_n\}$
- $\mathbf{D}$ is a set of discrete domains: $\mathbf{D} = \{\mathbf{d}_1, \ldots, \mathbf{d}_n\}$ in which $\mathbf{d}_i$ is the domain of v_i $\forall i = 1 \ldots n$.
- $\mathbf{R}$ is a set of constraints $\mathbf{R} = \{r_1, \ldots, r_m\}$ on possible values of variables in $\mathbf{V}$

The objective is to find *assignments to all variables* in $\mathbf{V}$ so that all the constraints in R are *satisfied*.

In the above definition, a constraint r_i is a restriction on the values that variables in $\mathbf{V}_{r_i} = \{v_{i_1}, \ldots, v_{i_k}\} \subset \mathbf{V}$ can take simultaneously. Equivalently, the constraint can be represented as a function: $r_i : \mathbf{d}_{i_1} \times \ldots \times \mathbf{d}_{i_k} \mapsto \{0, 1\}$ which maps assignments of variables in $\mathbf{V}_{r_i}$ onto 0 or 1. An assignment mapped onto 1 is said to satisfy r_i; otherwise, it does not satisfy r_i. In addition, since r_i has k variables, it is called a *k-ary* constraint. Binary constraints are 2-ary constraints. QoS constraints discussed later in this thesis typically have more than 2 variables.

Constraint satisfaction problems are often characterised by their density and tightness which are defined as follows [Tsa93]:

Definition 2 [Density]: The density of a CSP is the ratio of the number of constraints defined in the problem to the number of possible constraints[1].

[1] In this definition, two constraints are different iff they are defined on different sets of variables.

Definition 3 [Tightness]: The tightness of a CSP (or a constraint) is the ratio of infeasible assignments of the CSP (or the constraint) over the total number of all possible assignments.

In Definition 2, constraints are defined on different sets of variables. Multiple constraints on the same set of variables are considered as a single constraint after being aggregated together. The reason for this is density often being used in generating random CSPs. Generating two random constraints on the same set of variables is not necessary. For a CSP that has m constraints of *k-ary* and n variables, the density of the CSP is m/C_n^k.

In many practical applications, the description of a CSP may change over time [VS94a; MJT+01; SF02]. For instance, in resource allocation a new request for resource allocation many appear at any time while already served requests are cancelled. To describe a changing CSP, Dechter et al. [DD88] first introduced the notion of Dynamic CSP. In DynCSP, the set of variables and/or constraints of a CSP can be modified during the constraint resolution process. It has been pointed out that all the possible changes to the description of a CSP including constraint and domain modifications, variable additions and removals can be expressed in terms of constraint additions or removals [VS94b]. Constraint additions are referred to as restrictions whereas constraint removals are called relaxations. Thus a DynCSP is defined as follows:

Definition 4 [DynCSP]: A DynCSP is a sequence of static CSPs $\{P_0, P_1 \ldots\}$, each CSP resulting from a change in the preceding one. This change is a restriction (a new constraint is imposed on a subset of variables) or a relaxation (an existing constraint is removed from the CSP).

It is often assumed that only a reasonable number of constraints can be added to or removed from P_i in order to obtain P_{i+1}. The number of constraints can be measured through a *change rate* parameter [Mai05]. The parameter is computed as the average number of added or removed constraints between any two P_i and P_j over the time distance between them. Apparently the change rate is an indicator of how fast a DynCSP changes over time.

3.2.2 Algorithms for solving CSP

Considerable effort has been devoted to developing algorithms for CSPs. Those algorithms are often centralized where a central problem solver has the complete knowledge of all the variables and constraints of the CSP. Three important features of CSP algorithms are termination, correctness (or soundness) and completeness [Tsa93]:

Definition 5 [Termination]: An algorithm is terminated if its execution stops after a finite period of time.

Definition 6 [Correctness]: An algorithm is correct if every result returned by the algorithm is indeed a solution. In other words, any assignment returned by the algorithm contains the values for every variable, and the assignment satisfies all the constraints defined in the problem.

Definition 7 [Completeness]: An algorithm is complete if every solution can be found by it.

Tsang [Tsa93] classified algorithms for solving CSP into three categories: problem reduction, search, and solution synthesis. These are characterized as follows:

- *Problem reduction:* To reduce a problem into easier or known problems to solve. Arc and path consistency algorithms (e.g. AC-4 [MH86] for binary constraints, GAC4 [MM88] for non-binary constraints) belong to this category.

- *Search:* To enumerate assignments of variables and find solutions. A search is often carried out in parallel or after problem reduction. Backtracking is the most frequently used technique in this category. In addition, other strategies like lookahead (e.g. Forward Checking [HE80]) and gathering information while seaching (e.g. Dependency directed backtracking [SS77]) are also popularly used.

- *Solution synthesis:* To construct and expand partial solutions until all solutions can be found. In contrast to problem reduction, solution synthesis techniques constructively generate legal assignments rather than eliminate redundant and infeasible

assignments. Generally, solution synthesis can be seen as searching multiple partial solutions in parallel. Examples in this category are Freuder's Solution Synthesis Algorithm [Fre78] and Seidel's Invasion Algorithm [Sei81].

We note that backtracking will be referred to in different parts of this thesis. As mentioned above, the backtracking algorithms (i.e. the algorithms that use the backtracking technique to search for CSP solutions) are popular in the search category. In general, those algorithms continuously construct and expand a partial solution. The expansion follows a depth-first tree in which the search continues to find a value for the next unassigned variable in the variable list [Tsa93]. If the assignment leads to no possible further exploration, the last assignment is undone and a re-assignment is performed. This action is called backtracking. The backtracking algorithms are normally used with different heuristics. It can also be used with consistency checking (e.g. look-ahead techniques) to better prune the domains of unassigned variables. For CSP, backtracking algorithms are generally complete [Tsa93].

3.3 Distributed Constraint Satisfaction

3.3.1 Motivation and Problem Definition

In classical CSP, the sets of variables and constraints of a problem are assumed to be known and controlled by a centralized problem solver, thus the solving process is carried out in a centralized manner. This assumption is not very practical in distributed applications. Such applications are typical in distributed environments where the variables and constraints in a CSP can belong to different organizations and no single problem solver or organization can have an access to every variable and constraint. Distributed Constraint Satisfaction [YDIK92] has been introduced to handle this problem. The Distributed Constraint Satisfaction model is an extension of the classical constraint satisfaction model. A Distributed Constraint Satisfaction Problem (DisCSP) consists of a number of distributed agents (e.g service providers), each agent owns a number of variables and constraints. Similar to the classical CSP, the agents need to find an assignment for every variable so that

all the constraints can be satisfied. In DisCSP, privacy of constraints is a major concern for agents. An agent may not wish to share its local constraints with others. A popular definition of DisCSP in which an agent has only a single variable is given in [YDIK92] as follows:

Definition 8 [DisCSP-Single Var per Agent]: A DisCSP is a 4-tuple $\langle \mathbf{AV}, \mathbf{D}, \mathbf{R} \rangle$ where:

- $\mathbf{A}$ is a set of agents: $\mathbf{A} = \{A_1, \ldots, A_n\}$. Agents communicate by sending messages. Two agents can send messages to each other iff they know the addresses of each other. An agent may not know the addresses of all other agents.
- $\mathbf{V}$ is a set of variables: $\mathbf{V} = \{v_1, \ldots, v_n\}$. The variable v_i is *controlled* by the agent A_i $\forall i = 1 \ldots n$.
- $\mathbf{D}$ is a set of discrete domains: $\mathbf{D} = \{\mathbf{d}_1, \ldots, \mathbf{d}_n\}$ in which $\mathbf{d}_i$ is the domain of v_i $\forall i = 1 \ldots n$.
- $\mathbf{R}$ is a set of constraints $\mathbf{R} = \{r_1, \ldots, r_m\}$ on possible values of variables in $\mathbf{V}$. Each constraint in $\mathbf{R}$ is known by at least one agent in $\mathbf{A}$.

The objective is to find an *assignment to all variables* in $\mathbf{V}$ so that all the constraints in R are *satisfied*.

In the above definition, we say that v_i is *controlled* by A_j or equivalently A_j *owns* v_i iff A_j can set the value for v_i [YDIK92]. In this definition, an agent may know about a number of variables, however it controls only one of them. Furthermore, it is implicitly assumed in many studies (e.g. [YDIK92; Yok95; HY00; CH05; BBMMar; MSTY05; MJT$^+$01; ML04]) that only the owner of a variable is allowed to propose values for the variable. However, in other research (e.g. [SSF00; Rin01]), a slightly different model of DisCSP is used. In this model, an agent can propose a value for, and thus control, any variable that the agent knows.

In this thesis, a DisCSP is viewed as a collection of centralized CSPs owned by communicating agents. A general definition of DisCSP, based on this view, is given as follows:

Definition 9 [DisCSP]: A DisCSP is a triple $\langle \mathbf{A}, \mathbf{LC}, \mathbf{V}_c \rangle$ where:

- $\mathbf{A}$ is a set of distributed agents $\mathbf{A} = \{A_1, \ldots, A_n\}$
- $\mathbf{LC}$ is a set of (centralized) local CSPs: $\mathbf{LC} = \{lc_1, \ldots, lc_n\}$ in which lc_i is the local CSP of A_i. Some local CSPs, and thus the agents of those CSPs, may share a common subset of variables.
- $\mathbf{V}_c$ is a collection of controlled variable sets: $\mathbf{V}_c = \{\mathbf{V}_{c_1}, \ldots, \mathbf{V}_{c_n}\}$ where variables in $\mathbf{V}_{c_i}$ are controlled by A_i. In addition, $\bigcup_{i=1}^{n} \mathbf{V}_{c_i}$ is identical to the set of all variables from all lc_i.

The objective is to find solutions for all the problems in LC simutaneously.

In Definition 9, the set $\mathbf{V}_c$ distinguishes between variables controlled by different agents. Since $\bigcup_{i=1}^{n} \mathbf{V}_{i_c}$ is the set of all variables from all lc_i, each variable in lc_i belongs to some $\mathbf{V}_{c_j}$ and hence is controlled by some agent in $\mathbf{A}$. In constrast to Definition 8 where the relationship regarding the ownership between agents and variables is one-to-one, in Definition 9 the relationship is many-to-many. In other words, one variable can be owned and controlled by more than one agent and one agent can own multiple variables. Definition 9 therefore can be used to define more general DisCSPs as compared to Definition 8. For example, problems with multi-variables per agent in [SSF00; Rin01] correspond to Definition 9. In these problems, for each agent A_i, the set V_{c_i} of A_i's controlled variables is the same as the set of variables defined in lc_i.

DisCSP Example: Alice, Bob, and Carol are tutors for a university subject. Tutorials for the subject run on Monday, Wednesday, and Friday every week. Each of the tutors has his/her own time constraints. However, the tutors do not wish to reveal their constraints to others. The constraints are as follows: Alice and Bob both work for a company on Monday. Alice is also busy on Tuesday, and Carol is not available on Thursday. The objective is to assign one tutor to each tutorial so that every tutorial has a tutor and all the tutors' constraints are satisfied.

The above problem illustrates a simple example for demonstrating DisCSP. Figure 3.1 shows that it can be modelled as a DisCSP using Definition 8. In the figure, each agent (i.e.

tutor) owns a variable that represents the tutor's allocated time. For Alice , it is x_1; for Bob, it is x_2; and for Carol, it is x_3. The domains of those variables are $\{2, 3, 4, 5, 6, 7, 8\}$ where 2 represents Monday, 3 represents Tuesday and so on. Since tutorials are on Monday, Wednesday, and Friday, the constraint $r_{all} : \{x_1, x_2, x_3\} = \{2, 4, 6\}$ is known by every tutor. Alice and Bob work for the same company on Monday thus they share the constraint $r_{Alice-Bob} : x_1 \neq 2$ and $x_2 \neq 2$ which is unknown to Carol. In addition, Alice and Carol have their private constraints $r_{Alice} : x_1 \neq 3$ and $r_{Carol} : x_3 \neq 5$ respectively. This problem can also be modelled using Definition 9 by gathering the constraints known by each agent. For example, Bob's local CSP consists of r_{all} and $r_{Alice-Bob}$ as shown in the figure.

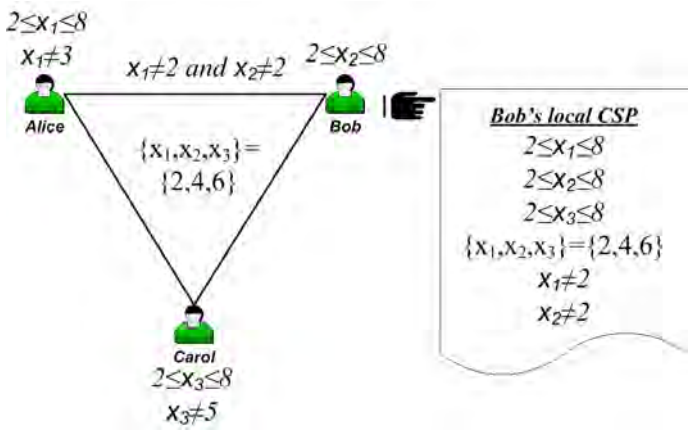


Figure 3.1: A DisCSP example: Three tutors, Alice, Bob, and Carol need to decide who will take which tutorial.

As discussed in Section 3.1.2, DynCSP [MYO03,ZUL01] has been extended from CSP to deal with uncertainties and dynamic changes. In distributed environments, a problem is even more susceptible to such uncertainties and changes. A change can occur in any part of the environment that the problem spans and at any time. For this reason, Dynamic Distributed Constraint Satisfaction Problem (DynDisCSP) has been extended from DisCSP to model the changes. DynDisCSP can be defined as follows:

Definition 10 [DynDisCSP]: A dynamic distributed constraint satisfaction problem is a DisCSP which can change over time. A change can be a restriction (i.e. a new constraint is imposed on some variables held at an agent, or a new agent is added) or a relaxation

(i.e. an existing constraint or agent is removed from the DisCSP).

3.3.2 Distributed constraint satisfaction algorithms

Many DisCSP algorithms [YDIK92; Yok95; SSF01; SSF00; MSTY05] have been extended from centralized search techniques. Typically those algorithms focus on a single variable per agent and shared binary constraints (i.e. constraints shared between two agents). Whilst some of them can be extended to multiple variables per agent, the performances for the extended version have not been well investigated.

Distributed constraint satisfaction has been a very active research field recently. One of most popular and important DisCSP algorithms so far has been the Asynchronous Backtracking (ABT) algorithm [YDIK92]. ABT is an extension of the backtracking technique used in classical CSP. In ABT, agents are first assigned different *priorities*. Priorities are used to define a temporarily hierarchical structure among agents. These priorities are helpful in guaranteeing ABT's correctness and completeness [YDIK92]. During the ABT execution, every agent attempts to solve its local CSP and find assignments to its controlled variables. The agents then communicate and exchange their assignments with their neighbors. If a solution found by an agent A_i is in conflict with some higher priority agents then A_i has to find another solution for its CSP. If such a solution cannot be found, then A_i reports the no-solution status to a higher priority agent A_j so that A_j can change its own solution[2]. This no-solution report by A_i to A_j resembles the backtrack performed in centralized backtracking CSP algorithms [YDIK92]. ABT terminates when there are no conflicts between the agents' local solutions.

Other important algorithms include Asynchronous Weak-Commitment (AWC) [Yok95] and Asynchronous Backtracking with Reordering (ABTR) [SSF01] which allow dynamic changes of agent priorities. Similar to dynamic variable ordering in centralized CSP, dynamic agent priorities are useful to improve the algorithm performance [SSF01]. In particular, they may have a shorter completion time for finding solutions. In addition, Asynchronous Aggregation Search (AAS) [SSF00] proposes to aggregate variable values

[2] A_j is chosen as the lowest priority agent among the A_i's higher priority neighbours[YDIK92].

in message exchanges to speed up the search. In AAS, instead of exchanging individual values, agents can send intervals (i.e. ranges of values) to propose the values for their variables.

More recent efforts in DisCSP research have focused on improving privacy protection. As discussed before, an agent in DisCSP has its private constraints and may not be willing to reveal them to other agents. However, during a DisCSP solving process, certain information about the constraints may be leaked. For example, in the tutor allocation example in the previous section, if Alice offers to take the Friday tutorial then it is revealed that Alice is free on Friday[3]. Generally, it is desirable that only private information from the final solution is revealed. Managing privacy to minimize the information revealed is the goal of many research including [YSH05] and [M.C03].

Distributed Constraint Optimization (DCOP) is another direction that has continued to attract attention recently. In DCOP, the goal is both to satisfy a set of distributed constraints and to maximize some objective function. DCOP problems are thus generally more difficult than DisCSPs. Important work on DCOP includes Optimal Asynchronous Partial Overlay (OptAPO) [ML04] and Asynchronous Distributed OPTimization with quality guarantee (ADOPT) [MSTY05]. OptAPO is extended from APO (Asynchronous Partial Overlay) [ML06] which is a satisfaction based algorithm. Similar to APO, OptAPO divides the agents into different subsets and centrally solves the sub-problems in each subset. Different mediator agents are elected, each for a subset, to distributively collaborate the solving processes of all agents. OptAPO thus is a mix of distributed and centralized constraint satisfaction. On the contrary, ADOPT is a fully distributed optimization algorithm and the first one to be so. It does not require any merging of sub-problems between agents. ADOPT is known to extend the iterative deepening search [RN03] into distributed environments [MSTY05].

Despite the possible complexity of the local solving at each agent, the DisCSP communication protocols[4] are often very simple. In many DisCSP algorithms including ABT,

[3]Here we assume that Alice is honest. She would not propose a day when she cannot attend.

[4]In this dissertation, we use DisCSP protocols to refer to the set of rules on message exchanges among agents which exclude the local consistency checking internally at each agent.

AAS, and ADOPT only a few message types need to be defined. Moreover, agents process the messages independently. In most of the cases, an agent does not keep old messages nor correlate messages from different senders. In other words, DisCSP protocols are lightweight and with small synchronization overhead. These characteristics are important for a protocol that needs to scale well in networks with a large number of agents. The DisCSP algorithms are thus attractive for the QoS management problem where many service providers may be involved.

3.4 Summary

This chapter has introduced classical constraint satisfaction as well as its extension for dynamic environments. The chapter has reviewed different CSP techniques consisting of the problem reduction, search, and solution synthesis. DisCSP has been introduced as an emerging version of CSP for distributed environments. In this chapter, maintaining agents' privacy is identified as a major motivation behind DisCSP. Different DisCSP algorithms have been discussed. Among them, ABT is an important algorithm from which many others extend. In Chapters 6 and 7, ABT and one of its extensions - AAS will be described in detail. Also new ABT extensions will be proposed for QoS management of service compositions.

Chapter 4

A DisCSP based QoS Management Framework

4.1 Introduction

The previous chapter discussed key concepts and important developments in the field of DisCSP. This chapter proposes a DisCSP based framework for QoS management of composite Web services. The framework is comprised of a number of distributed agents that represent organizations participating in the provision of composite services. These agents collaborate and use DisCSP algorithms to manage QoS of the Web services together.

The relationships between the QoS of a Web service and the underlying resources are examined in the next section of this chapter. In Chapter 2, QoS composition and adaptation have been considered as the two major tasks in QoS management. Section 4.3 of this chapter models these tasks as instances of DisCSP and DynCSP. Section 4.4 describes the architecture of the proposed framework and its sub-components and relates them to relevant developments in the Web service technology. Section 4.5 discusses criteria to evaluate a QoS management system that uses the proposed framework. Finally, Section 4.6 summarises the chapter.

4.2 QoS dependencies

In Chapter 2, various QoS parameters for Web services, including response time, availability, and security, have been defined and discussed. This section examines: 1) the dependencies of these parameters on the Web services' underlying resources, and 2) the relationships among the parameters for different Web services. These dependencies and relationships are modelled as constraints in CSP in this thesis. The dependencies are important for a QoS management system since they can be used in the estimation of how much resource is needed in order to attain a certain QoS level. Similarly, the relationships between the QoS of different Web services are significant for managing all the Web services collectively and successfully.

The dependencies between the QoS of Web services and the services' underlying resources originate from the design of network protocol stacks. In the classical TCP/IP model [Int89], communications and computer networks are divided into different abstract layers (see Figure 4.2). A layer is a collection of related protocols and software that provide services[1] to the layer above and use services from the layer below. Web services belong to the Application layer in the model. A Web service uses some services (e.g. TCP) from the transport layer to support its operations. Thus in order to attain a certain QoS level for the Web service, appropriate resources should be allocated to the transport services. It is worthy to note that these resources can be the QoS at the transport layer. Hence, there are dependencies between the QoS of the Web service and the QoS at the transport layer. The ability to map and inter-convert QoS values between different layers has been the focus of many research (e.g. [CK00; SC00b; YN04]). Those research study the conversion of the QoS of services at an individual layer through those at lower layers.

The relationships between the QoS of Web services are formed mainly as a result of the service abstraction principle. In the service oriented model, services are the abstraction of the computing resources. For example, computer servers, networks, storages, and software programs are modeled as services that can be combined together to form new value added Web services. On the one hand, Web services that appear to be separated at the service

[1]Services here are generic services that are not necessarily Web services or services in SOC.

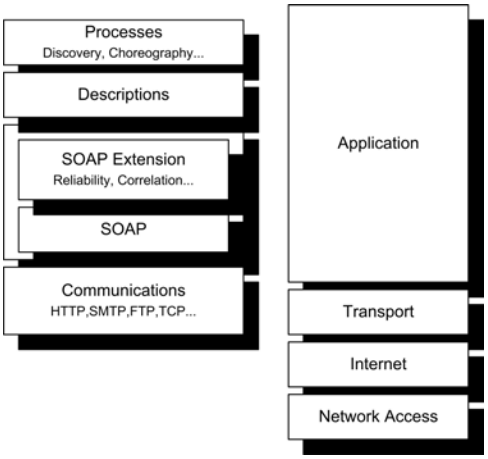


Figure 4.1: The Web service architecture [W3C] (on the left) and its relationship to the Internet protocol architecture [Int89] (on the right). The Web service stack belongs to the application layer which uses resources from lower layers.

layer due to the abstraction may share some computing resources at a lower layer. On the other hand, the QoS parameters of the Web services are dependent on the resources as discussed previously. Consequently, these QoS parameters are indirectly related to each other through their dependencies on the common resources. In other words, there exist relationships between them.

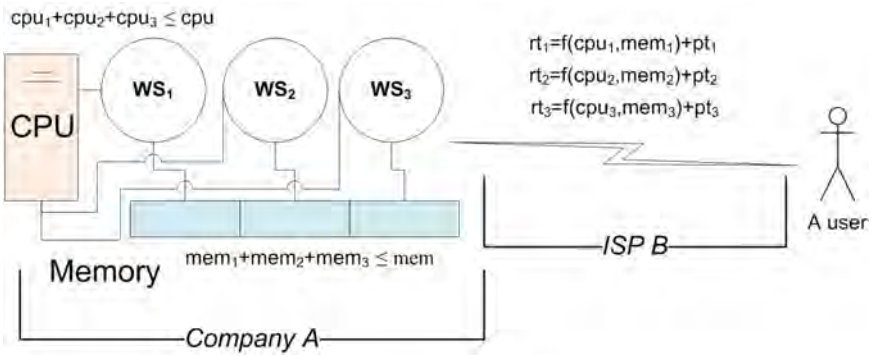


Figure 4.2: Dependencies between the response times rt_1, rt_2, rt_3 of three Web services S_1, S_2, S_3. These Web services are hosted on the same Web server and hence share the server CPU as well as memory. Constraints between different parameters are shown for concurrent executions of the services.

Figure 4.2 presents an example of three Web services S_1, S_2, and S_3, which are hosted on the same Web server and share the server's memory and CPU. The server belongs to company A which uses the communication network provided by Internet service provider B. The average response time of the Web services is the QoS parameter to be managed. The following are additional assumptions:

- The clients of the Web services also use B as their Internet service provider and the transmission times of their requests are negligible[2].
- The execution time of a Web service is a function, denoted as f, of its allocated memory and CPU share. For the sake of simplicity, we assume that this function has the same formula for all Web services in the example.

The Web services appear to be independent on the Web service layer. However, there is a dependency between the QoS (i.e. average response time) of these Web services whenever they are concurrently executed. We denote f as the function of service execution time then the dependency can be captured through the below expressions:

$$mem_1 + mem_2 + mem_3 \leq mem \tag{4.1}$$

$$cpu_1 + cpu_2 + cpu_3 \leq cpu \tag{4.2}$$

$$rt_i = f(mem_i, cpu_i) + pt_i, i = 1..3 \tag{4.3}$$

$$rt_i \leq \alpha_i, i = 1..3 \tag{4.4}$$

In the above expressions, mem_i, cpu_i, rt_i, and pt_i are the memory, CPU share, response time, and the average round trip propagation time of the Web service S_i $(i = 1 \ldots 3)$ respectively. Inequalities (4.1) and (4.2) show the constraints on the memory and CPU sharing between Web services. Equation (4.3) expresses the constraints on response time as the sum of execution time and propagation time. Furthermore, Equation (4.4) represents the client's QoS requirements on the response time of the Web services. In particular, the response time of the service number i must be smaller than or equal to α_i.

[2]Note that response time is often considered as the sum of transmission time, round trip propagation time, and execution time.

4.3 QoS management of Web services as an instance of DisCSP

Thus far, the dependencies between QoS parameters of a Web service and its underlying resources, together with the relationships between these parameters for different Web services have been discussed. Regardless of their complexity, these dependencies and relationships can be generally considered as constraints in the context of constraint satisfaction. The constraints restrict the values of the QoS parameters and resource variables in accordance with the dependencies between them. This chapter will use the constraints to model the QoS management as an instance of DisCSP.

As mentioned in Chapter 2, the main objective of a QoS management system for Web services is to guarantee the satisfaction of a number of users' QoS requirements for the services. Let us call the set of QoS parameters to be managed $\mathbf{Y} = \{y_1, \ldots, y_m\}$ and the set of QoS requirements on these parameters $\mathbf{R} = \{r_1, \ldots, r_k\}$. $\mathbf{C} = \{c_1, \ldots, c_q\}$ is a set of constraints which describes the dependencies between the QoS parameters and resource variables. Those QoS parameters and resource variables make up the set $\mathbf{X} = \{x_1, \ldots, x_n\}$. In order to achieve its objective, the management system can control and tune the values of $\mathbf{X}$ so that all $r_i \in \mathbf{R}$ are satisfied. The relationships in $\mathbf{C} = \{c_1, \ldots, c_q\}$ between resource variables and QoS parameters are preserved during the tuning process. These are illustrated in Figure 4.3

It is worth noting that:

- The variables in $\mathbf{X}$ and the constraints in $\mathbf{C} \cup \mathbf{R}$ are distributed over different organizations and administrative domains. For the example of three Web services in Figure 4.2, the memory and CPU variables belong to company A whereas the propagation delay variable is controlled by ISP B. In general, the service oriented model separates services and applications from the underlying infrastructure; abstracts the concept of computing resources and makes the overall system appear as a single set of capabilities. However, its underlying resources are in fact distributed among different business enterprises.

- The constraints in $\mathbf{C} \cup \mathbf{R}$ between QoS parameters and resource variables may not

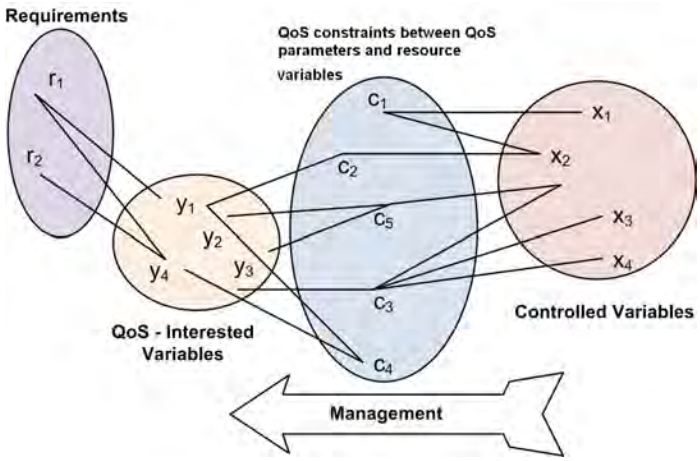


Figure 4.3: Based on a set of constraints $\{c_1, ..., c_p\}$, a set of controllable variables $\{x_1, ..., x_n\}$ can be used to control the values of QoS parameters $\{y_1, ..., y_m\}$. The objective is to satisfy all QoS requirements $\{r_1, ..., r_k\}$.

be visible to every organization. For the three Web services example in Figure 4.2, the company A can decide on the visibility of the constraints (4.1), (4.2), as well as of function f. Specifically, A can make the constraints visible as public constraints, or hide them within A as private constraints. In general, constraints between QoS parameters and resource variables are domain specific and require appropriate expert knowledge. Such knowledge can be difficult to obtain by an outsider. Also the local organization may not be willing to reveal the knowledge externally. Consequently, the constraints can remain unknown to other organizations.

- The objective of QoS management can be achieved once an assignment for X, that satisfies all constraints in $\mathbf{R}$, is found. In addition, due to the relationships between QoS parameters and resource variables, all constraints in $\mathbf{C}$ should also be satisfied.

Until now, the QoS management appears to resemble the description of DisCSP. We can make the following assumptions in order to model QoS management tasks as instances of DisCSP:

Assumptions: The two assumptions in the proposed framework are:

- QoS parameters and resource variables can be discretized to have discrete domains.

- QoS parameters and resource variables that are not controlled by the management system have constant values during the QoS (re-)composition processes. However, those parameters and values can change over time which leads to the QoS adaption

To justify these assumptions, note that in Chapter 2, numerous QoS parameters for Web services have been discussed. Typically, these parameters can be considered as having discrete domains. Similarly, in many studies (e.g. [CK00; SC00b; YN04]), different resource variables, including TCP data rate, packet lost ratio, and packet size, have discrete values. The first assumption is therefore justified due to the popularity of discrete QoS parameters and resource variables. For the second assumption, if a resource variable is not controlled by the management system then the resource is often provided by a third party. The management system can negotiate with the third party and enforce the condition that the resource level should remain constant in their agreement during a certain period of time. Thus the second assumption is also reasonable.

To model DisCSP agents for the QoS management, let us consider all organizations and companies which control the variables in $\mathbf{X}$ and constraints in $\mathbf{C}$. An organization or a company is referred to as an Autonomous System (AS) in this thesis. An AS is an independent administrative domain and is represented by an agent. An AS and its agent control the variables in $\mathbf{X}$ and the constraints in $\mathbf{C} \cup \mathbf{R}$ that belong to the AS. During the management, all the agents collaborate to guarantee that all constraints in $\mathbf{C} \cup \mathbf{R}$ are always satisfied. If we go back and revise the management tasks in Section 2 of Chapter 2, these tasks can be translated into a DynDisCSP problem as shown in Table 4.1.

In the table, the QoS composition is mapped directly to a standard DisCSP, whereas the QoS adaptation is modelled as a DynDisCSP. Note that the DisCSP of the QoS composition can be considered as an element in the DisCSP series of the DynDisCSP. If an optimization objective (e.g. to minimize the cost of the QoS composition for the client or to minimize the time taken for the QoS adaptation) is taken into account, then the QoS composition and the QoS adaptation tasks can be considered as instances of DCOP in static and dynamic environments respectively.

QoS Management Task	**Distributed Constraint Satisfaction Problem**
QoS composition: Find the QoS values of component Web services so that the requirements on the QoS of the composition are satisfied.	*Static DisCSP:* Find values for X so that all constraints in C and all requirements in R are satisfied.
QoS adaptation: Respond to events and changes that may affect service peformance so that all constraints (especially the clients' QoS requirements) can still be satisfied.	*DynDisCSP:* Update new constraints resulting from the changes and search for the values of X again

Table 4.1: QoS Management tasks as instances of DisCSP

4.4 A DisCSP based QoS Management Framework

4.4.1 Overview

In this section, a DisCSP based framework is proposed for the QoS management of Web services. In principle, the framework models QoS parameters and resource variables as distributed variables and the dependencies between them as distributed constraints. Within the framework, agents act as owners of QoS parameters and resource variables on behalf of different ASs. They collaborate and ensure the fulfillment of the client's QoS requirements for the Web services while optimizing some joint function of rewards. The agents are distributed solving entities that use DisCSP algorithms to achieve the management objective.

Figure 4.4.1 illustrates the proposed framework from a high-level view. It shows the agents as a distinctive part of the framework. The agents are distributed decision makers and belong to separate ASs that have different administrative domains. The agents can access internal constraints and use existing local management capabilities in their AS to support the QoS management.

In the remainder of this section, the architecture of the framework and its different phases of management will be described. In addition, relevant Web service technologies will be described as important instrumentations to support the framework and its imple-

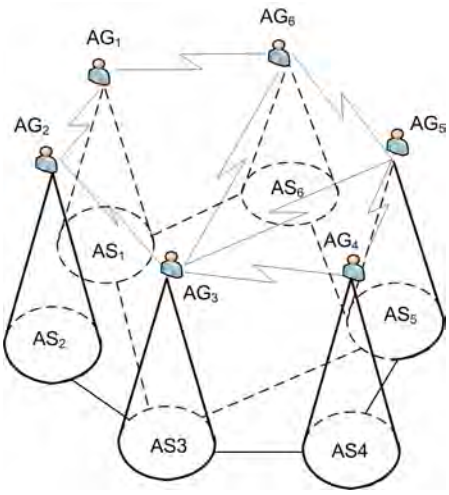


Figure 4.4: Different Autonomous Systems are represented by different management agents which collaborate to manage QoS of Web services.

mentations.

4.4.2 Architectural pattern

This section describes an architectural pattern for the proposed DisCSP based framework. An architectural pattern is a schema that expresses a fundamental structural organization for a software system. This pattern serves as a basic design principle for a management system that uses the framework. Above all, the architectural pattern gives an insight into the approach in terms of implementation as well as explains how our framework is linked to and can be built with existing technologies.

In this proposed architectural pattern, the QoS management framework consists of different ASs. Each AS is considered as having three management entities: an agent, a Constraint Manageability Interface, and a Resource Manageability Interface. These management entities are described as follows:

Distributed agents: Distributed agents are *autonomous and communicating* entities. They exchange messages asynchronously and collaborate using DisCSP algorithms to manage the QoS of Web services together. Agents are assumed to have the knowledge of different DisCSP algorithms in order to use them. An agent does not need to know every

other agent. An agent A_1 can only communicate with another agent A_2 if A_1 knows A_2's address.

In the proposed framework, it is not strictly required that an AS is represented by only a single agent. In fact, an AS can be represented by a group of agents. This is useful when the AS owns a distributed infrastructure and thus, multiple agents are needed at different places to control the AS's resources. Apparently, these agents share all variables and constraints. In such a situation, the phrase "the AS's agent" refers to the group of agents representing the AS as a whole.

Constraint Manageability Interface: The Constraint Manageability Interface is a Web service that provides SOAP based methods for accessing and querying constraints internally inside each AS. The interface is used by the representative agent in the AS to retrieve QoS related constraints. As discussed before, local constraints can be domain specific and the task of finding these constraints may require local expert knowledge. The Constraint Manageability Interface separates the task from the agent's implementation. It provides the agent with the following functionalities:

- Methods for the agent to enumerate and retrieve constraints related to a specific QoS parameter or resource variable. The returned constraints are in the XML format that potentially can be used with different agent's implementations. XCSP (XML for Constraint Satisfaction Problem [xcs07]) is an example of an XML based language that is designed to describe constraints in CSP.

- Methods to notify the agent when a new constraint related to the QoS of the managed Web services appears or an existing constraint becomes invalid. Essentially, the methods update the agent on changes in the surrounding environment so that the agent can modify its local CSP in accordance with the changes.

- Methods to find additional information associated with a constraint. The information includes the addresses of all agents which own some variables in the constraint. The methods are useful for the agent to discover other ASs' representative agents. These methods will be discussed in Section 4.4.3 of the chapter.

Figure 4.4.2 shows an example in which the constraints can be collected at the resource layer from the local resource management interfaces, or at the Web service layer in the agreement document and the structure of a service composition.

Resource Manageability Interface: Similar to the Constraint Manageability Interface, the Resource Manageability Interface is a Web service. It allows an agent to access the native resource management capabilities available in each AS. Through the Resource Management Interface, the agent can control the values for the QoS parameters and resource variables that belong to the AS, and thus have control over the outcome of the managed Web services' QoS. In particular, the Resource Manageability Interface provides the following methods:

- Methods for the agent to inspect the real value of a variable that the AS has control over. The agent can actively query the value or passively listen to any changes to the value. The value is a temporary assignment for the variable in the agent's local CSP.

- Methods for the agent to update the value of the variable that the AS has control over. The methods are often offered through some native management capabilities inside the AS. The Resource Manageability Interface exposes these capabilities through the standard SOAP based methods so that the agent can use them without major concern of interoperability with the native functions.

The Resource Management Interface can be implemented using OASIS's Web Service Distributed Management (WSDM) standard [Hew04b]. WSDM exposes the local resource management capabilities available inside each organization as Web services. Each capability has a specific manageability interface through which the capability is managed. In the context of WSDM, a resource variable that the AS has control over can be modelled as a managed resource. The native capability of monitoring and updating values for the variable can be exposed to the agent through a WSDM interface. The Resource Management Interface can finally be presented as an aggregation of those WSDM interfaces.

Figure 4.8 illustrates the interaction between the above three components in the pro-

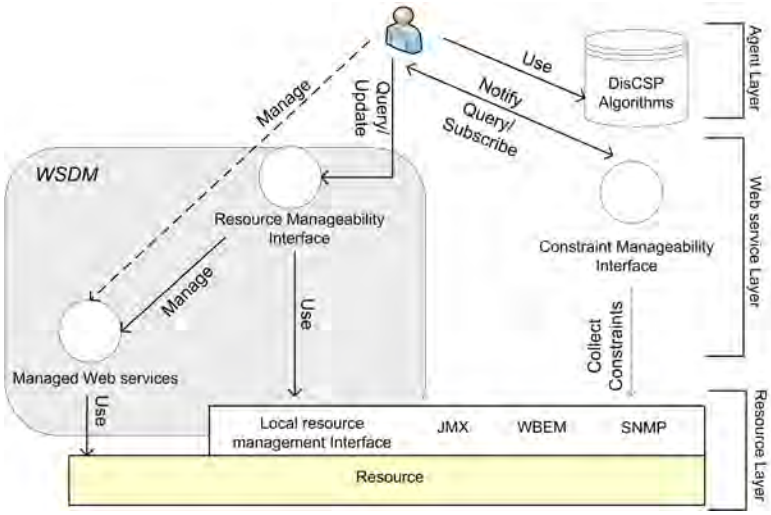


Figure 4.5: Different components of the management system inside an AS. Resource Manageability and Constraint Manageability Interface are two Web services that expose the native management capabilities to the agent.

posed framework. The interaction spans over three different layers: agents, Web services, and resources inside every AS. There is a logical separation between these layers. The agent layer is for decision making and reasoning while the Web service layer is for computing and the communication infrastructure. An agent in the upper layer carries out complex reasoning tasks using existing information and computing resources offered by the lower Web service layer. In particular, the agent can control resource variables on the resource layer through the WSDM Management Using Web Service (MUWS) [Hew04a] manageability interfaces on the Web service layer. Recall that the WSDM MUWS is a Web service abstraction over the native management interfaces such as Simple Network Management Protocol (SNMP) [Sta98], Common Information Model/Web-Based Enterprise Management (CIM/WBEM) [Dis07], and Java Management Extension (JMX) [Sun07] as depicted in Figure 4.4.2.

4.4.3 Management phases

The previous section presented an architectural pattern for the proposed framework. This section further discusses its operation through different QoS management phases. In gen-

eral, the management in the proposed framework comprises three major phases: Initial DisCSP Setup, DisCSP (Re-)Solving, and DisCSP Update. These phases are depicted in Figure 4.6. The Initial DiscSP Setup phase takes place only once whilst the DisCSP (Re-)Solving and DisCSP Update phases can occur repeatedly.

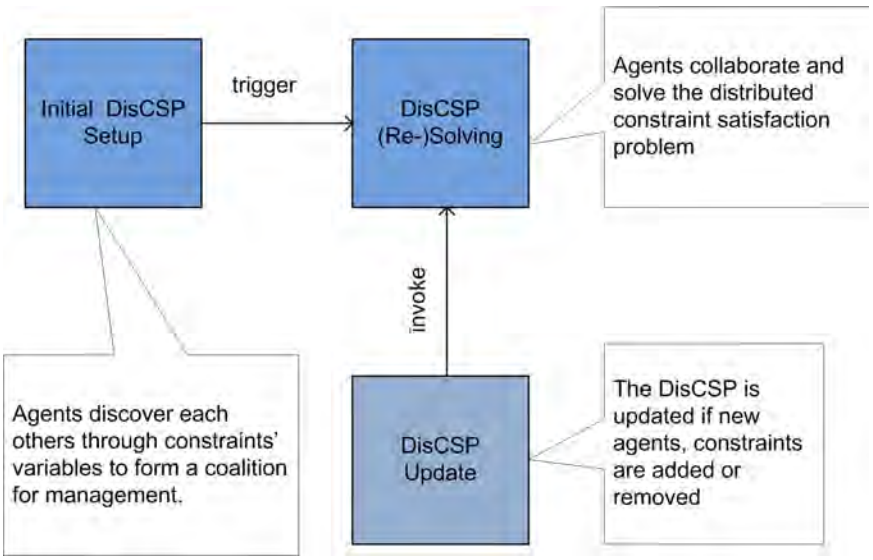


Figure 4.6: Different phases of the QoS management process carried out by management agents.

Phase 1 – Initial DisCSP Setup: In this phase, the agents discover each other to form a coalition before the management can start. The coalition is a set of all agents from different ASs which agree to take part in the management. Initially an agent in an AS is assigned the task of QoS management of some Web services. It analyses the QoS constraints available in the AS and assembles a list of QoS parameters and resource variables that the agent does not have control over. The agent uses the Constraint Manageability Interface to find the owners of these parameters and variables, and invite them to join the coalition. These steps are repeated for newly joined agents since the variables owned by the agents may have constraints with variables that are controlled by someone else. In general, an existing agent can invite several new agents. It is possible that a new agent is invited by more than one agent. In this case, the agent simply ignores repetitive invitations. The agent network in the DisCSP is gradually expanded until no more agents are added.

Figure 4.7 illustrates an example of the Initial DisCSP Setup phase. A management system initially has only an agent in AS_1. This agent then invites representative agents from AS_2 and AS_3. The agents from AS_2 and AS_3 invite new agents and so on. In this example, both AS_4 and AS_8 are invited by more than one agent.

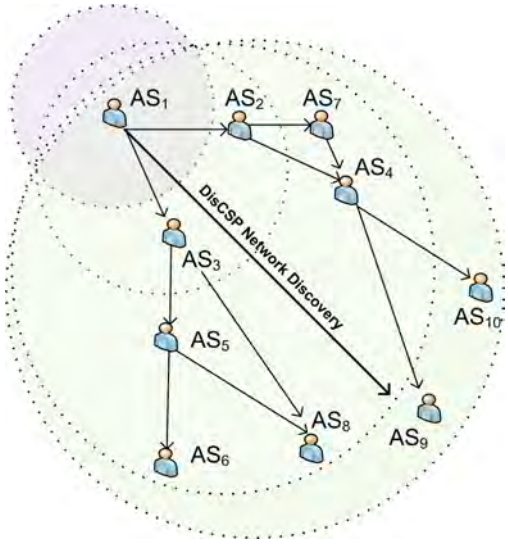


Figure 4.7: Agents discover each other and form a coalition to manage the QoS of Web services together.

Naturally, questions can be raised on the allowable size of the coalition and what to do if some newly discovered agents are rejected from joining. The answer to the first question depends on the policy that existing agents have on the coalition. Such a policy is not strictly defined and imposed in the framework. Different implementations can have different policies. A simple policy can be that the initiator agent sets up an upper limit for the coalition size at the beginning, and forbids any new agents to join once the size of the coalition has reached its limit. In order to enforce this policy, an agent is required to report its joining status to the initiator agent so that the latter can keep track of the actual coalition size. For the second question, if a new agent A_j is invited by an existing A_i; but A_j does not want to join the management coalition then A_i can do one of the following:

- Negotiate with A_j on fixed values for all variables controlled by A_i. A_j's variables

thus can be considered as constants.

- Add to its local CSP a new constraint that the final solution of the CSP must satisfy all constraints for any legal values of A_i's variables. The legal values are specified in the contract between A_i and A_j; for example, if A_i and A_j signed a contract in which a variable v_j of A_j can have any integer value between 8 and 10. A_i then adds a local constraint that the final solution must satisfy all constraints $\forall v_j \in \{8, 9, 10\}$.

Phase 2 – DisCSP Solving: Upon completion of the initial DisCSP network, the DisCSP solving can be started. During this phase, agents collaborate and perform the QoS composition and adaptation using a DynDisCSP algorithm. According to Section 4.3, the agents can model the QoS composition and adaptation as a DynDisCSP and thus it can be solved by the DynDisCSP algorithm. The algorithm is designed to handle multi-variables that are the QoS parameters and the resource variables. During this phase, the agents exchange messages and collaborate using the DynDisCSP algorithm. The framework and the DynDisCSP algorithm, do not strictly specify a local solving procedure inside each agent. Instead, the agents can use different techniques for their local CSP solving. It is important to verify the agents' conformance to the DynDisCSP algorithm during this phase to ensure that the solutions for the QoS composition and adaptation are correct. Such a verification mechanism and the corresponding DynDisCSP algorithm will be detailed in the subsequent chapters of this thesis. In particular, chapters 6 and 7 discuss different DynDisCSP algorithms in the context of QoS management. Chapter 8 presents a verification mechanism for the Web service environment.

Phase 3 – DisCSP Update: A Web service environment is by nature unpredictable with many uncertainties. In this environment, changes to the Web services to be managed and their underlying resources may occur at any time. Examples of the changes are:

- Alternation in the existing resource inside an AS. This leads to modifications of some constraints in the local CSP known by the AS's representative agent.

- Removal of some Web services under the management scope due to the expired lifetime of the services. QoS requirements for the services as well as their related

resource variables need to be removed from the DynDisCSP.

- Addition of new Web services that are to be managed. QoS requirements for the services as well as their related resource variables are added into the DynDisCSP.

In the proposed framework, this DisCSP Update phase is for the purpose of accommodating the above unexpected changes. It is executed in parallel with the DisCSP solving phase. During this phase, the DynDisCSP is updated for constraint additions and removals. Constraint additions may require new agents to join the network. If this is the case, the update process happens as in the Initial DisCSP Setup where constraints are analysed and owners of their variables are contacted and invited to join the agent coalition. Similarly, constraint removals may lead to the departure of some existing agents from the coalitions which are no longer related to the managed Web services. Once an agent is leaving the network, its neighbor agents are notified about the leaving. Also, existing constraints are revised and updated by the DynDisCSP algorithm in order to find correct solutions.

4.5 Flexibility and Efficiency

This session discusses flexibility and efficiency of a management system implemented in the proposed framework. A management system is said to be more flexible than others if it can provide more solutions for the DynDisCSP of QoS composition and adaptation. Efficiency refers to how much computational cost is incurred in order to find the solutions. In general, it is desirable for a management system to be more flexible and efficient. These characteristics are useful to evaluate different management systems in the proposed framework.

4.5.1 Flexibility

To manage the QoS for the same set of Web services, there can be different management systems which are implemented using the proposed framework. For the example of three Web services in Figure 4.2 where the response times of the Web services need to be man-

aged, a management system M_1 may comprise company A and ISP B whereas another management system M_2 may have A only. Apparently in order to fix a violation of the client's QoS requirements, M_1 can provide any solution that M_2 finds since a solution reported by M_2 is actually from A which is part of M_1. However, M_2 cannot give any solution that requires a new value of the propagation time whereas M_1 can do this through ISP B. Hence M_1 is more flexible than M_2.

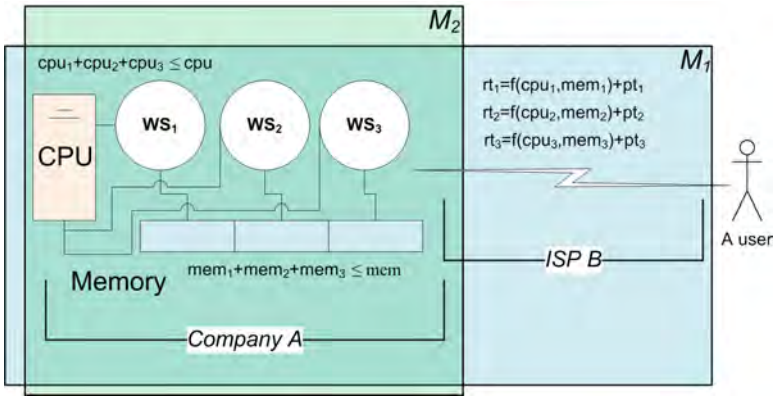


Figure 4.8: Two different management systems M_1 and M_2 can both be used to manage three Web services. However M_1 is more flexible than M_2.

Generally, for two different management systems M_1 and M_2 are used for QoS management of the same Web services. Supposing that agents in M_1 have the set of variables $\mathbf{X}_1$ and the set of constraints $\mathbf{C}_1$ and agents in M_2 have the set of variables $\mathbf{X}_2$ and the set of constraints $\mathbf{C}_2$. If the CSP problem $P_1 = \langle \mathbf{X}_1, \mathbf{C}_1 \rangle$ is a relaxation of $P_2 = \langle \mathbf{X}_2, \mathbf{C}_2 \rangle$ then any solution for P_2 is also a solution for P_1. Consequently, as long as the DisCSP algorithms used by the two management systems are complete, M_1 is more flexible than M_2. This observation illustrates the importance of the Initial DisCSP Setup phase discussed in Section 4.4.2. Since the CSP is relaxed every time a new agent joins the coalition, better flexibility for the management system can be achieved by expanding the coalition and inviting more agents to join in. However, this flexibility may be at the cost of being less efficient as discussed in the next section.

4.5.2 Efficiency

While flexibility indicates how much control a management system has over the set of QoS parameters and resource variables, efficiency is a measure of the performance of the DynDisCSP algorithm used by agents. Specifically, effciency refers to the time and computing storage required to carry out the QoS composition and QoS adaptation tasks. *A more efficient management system can find a solution for the DynDisCSP in a shorter time as well as use less computing storage in order to do this.* Designing efficient DisCSP algorithms for the agents is therefore important. This is the focus of Chapter 6 to 8 of this thesis.

It is also worthy to note that in general the completion time and computing storage required for a DynDisCSP varies according to the size of the problem. In particular, if more agents and variables are added into the DynDisCSP, more storage space is needed and longer time may be taken for the DynDisCSP algorithm to find a solution. Therefore, there can be a tradeoff between efficiency and flexibility of a management system.

4.6 Discussion

So far, the architecture as well as different management phases of the proposed framework have been discussed. This framework has the following important properties:

- ***Property 1:*** The framework is distributed. In the framework, different organizations actively participate and collaborate to carry out the QoS management of Web services.

- ***Property 2:*** The framework can protect the privacy of an organization's internal constraints from being revealed to others.

- ***Property 3:*** In the framework, multiple Web services and service compositions can be managed simultaneously.

- ***Property 4:*** The framework allows a number of new Web services to be added or existing Web services to be removed from the management scope.

The above properties are also the advantages of the proposed framework over many existing studies on QoS management including [PSL03], [POSV04],[JCZ03], [PS03], [PEWS03], and [RD05]. In particular, according to Property 1, the framework is totally distributed and hence there is no single point of failure. Property 2 ensures that organizations' internal constraints are kept invisible to outsiders. An organization does not need to know more than what it already knows about other organizations. This is an advantage over other approaches such as [PS03], [PEWS03], and [RD05] in which an organization may need permission to access and use local capabilities offered by some other organizations. Properties 3 and 4 are most important as they show that this framework is more capable than other frameworks proposed in [PSL03] and [POSV04] because it can manage multiple Web services at the same time.

Whilst the main features of the framework have been discussed, there are two issues which require further investigation to make the framework complete as follows:

- ***Constraint Collection:*** Finding the constraints on QoS parameters and resource variables is important for the proposed framework. A management system using the framework should be able to discover and gather those constraints and expose them to agents. Constraints can come from different network layers. Many works have been carried out to find such constraints at low-level network layers e.g. [CK00; SC00b; YN04]. However, a very few of them have been carried out for the Web service layer.

- ***Distributed Solving Algorithms:*** The core part of reasoning and decision making of the proposed framework is the set of distributed search algorithms and protocols available for the management agents. This framework does not assume that the agent is individually intelligent. However, these agents collectively are capable of attaining goals that are difficult or impossible to be achieved by an individual agent. These are done through distributed search algorithms. Therefore, DisCSP/DynDisCSP algorithms play a central role in the framework.

The constraint collection problem will be addressed in Chapter 5 whereas the second issue of designing new DisCSP/DynDisCSP algorithms will be solved in Chapters 6, 7, and 8 of this thesis.

4.7 Summary

This chapter has proposed a distributed framework which employs DisCSP techniques to solve the QoS management problem. Each organization or service provider in the framework is represented by an agent. These agents collect QoS related constraints inside their organization and then collaboratively find the required amount of resources to satisfy all of the QoS constraints. Essentially, the agents interact and cooperate with each other using DynDisCSP algorithms to automate the QoS composition and adaptation tasks in QoS management.

This chapter has also presented different components, as well as management phases, of the proposed framework. However, there are still open issues of finding QoS constraints and designing DynDisCSP algorithms. The next chapter studies the QoS constraints in the Web service layer whilst the remainder of the thesis focuses on designing DisCSP algorithms to support the QoS management.

Chapter 5

Constraint Formation and Representation*

5.1 Introduction

The previous chapter proposed a DisCSP based framework for QoS management. An important component in the framework is the Constraint Manageability Interface which provides access to local constraints inside each AS. These constraints need to be discovered in the AS. Finding the constraints at a network layer lower than the Web service layer is often domain dependent, and has been investigated in a number of studies including [PEH+00; SC00a; DBC04]. This chapter is about finding these constraints at the Web service layer.

As discussed in Chapter 3, QoS related constraints can reside on different network layers. Web service layer constraints are the constraints with variables in the Web service layer only. In general, a Web service layer constraint can be explicitly described in documents such as service contracts, or implicitly deduced from a functional Web service compositions. A client's QoS requirements for a Web service are examples of constraints that can be found in a contract. Extracting these constraints from the contract is not difficult since the constraints are often well defined. However, finding QoS related constraints

*A preliminary version of the work presented in this chapter has been published in [NKP06b].

from the functional Web service compositions is more difficult since such a constraint is implicitly imposed by a particular arrangement of component Web services in the compositions. Different functional compositions can have different arrangements. To give a simple example, suppose that a composite Web service A comprising of component services B and C in series. An induced constraint from the compositional structure of A is: $t_A = t_B + t_C$ where t_A, t_B, t_C are the execution time of A, B, and C respectively. The constraint can be much more complex for compositions with nested structure.

This chapter specifically aims to find QoS related constraints imposed by functional Web service compositions. In particular, it focuses on constructing the formulas and expressions of the constraints. This process is refered to as the *QoS constraint formation* process. The chapter is organized as follows: In the next section, related work on QoS computation/aggregation for Web services are discussed. Two algorithms for QoS constraint formations are proposed in Section 5.3. A comparison of these two algorithms is presented in Section 5.4. Section 5.5 discusses constraint privacy. Finally Section 5.6 summarises the chapter.

5.2 Related work

There have been many XML based languages developed to support Web service composition [Aal03]. The most popular one has been WS-BPEL (Web Services Business Process Execution Languages) [OAS06c]. WS-BPEL defines a number of primitive activities such as SEQUENCE, SPLIT, and JOINT, etc...These activities in WS-BPEL are basic *workflow controls* upon which more complex compound structures can be built. Other than WS-BPEL, WS-CDL (Web Services Choreography Description Language) [W3C06] is becoming a new standard from W3C with the focus on choreography. Despite these different languages, they share many common features in the control flows. Vander Aalst at el. in [AtHKB03] carried out a survey on different composition languages including WS-CDL and WS-BPEL, and proposed a set of generic *patterns* for workflow controls commonly used in these languages. These patterns reflect the control-flow dependencies between various tasks (i.e. service executions in a composition). The workflow control patterns are useful

as they are simple but general enough for analysing Web service compositions written in different languages.

No.	Pattern	Explanation
1	Sequence	A sequence of service executions
2	Loop	Repeating the execution of a service
3	XOR-XOR	An XOR-SPLIT followed by an XOR-JOIN
4	AND-AND	An AND-SPLIT followed by an AND-JOIN
5	AND-DISC	An AND-SPLIT followed by an m-out-of-n join (i.e. a *discriminator* workflow pattern)
6	OR-OR	An OR-SPLIT followed by an OR-JOIN
7	OR-DISC	An OR-SPLIT followed by an m-out-of-n join

Table 5.1: QoS composition patterns

There have been many studies on QoS computation for composite Web services [Men04; GNCW03; Men02; LNZ04; JRG04]. In these studies, the QoS values of a composition are calculated using the QoS values of its component services. In order to carry out the calculation, workflow controls have been used. Many existing studies (e.g. [Men04; GNCW03; Men02; LNZ04]) are limited to only a few simple workflow controls, Jaeger M. C. et al. [JRG04] consider workflow control patterns [AtHKB03] instead. They present seven QoS composition patterns as shown in Table 5.2. The authors also propose a computation formula for each composition pattern for different QoS parameters including cost, response time, encryption, throughput, and uptime probability (see examples in Table 5.2). By considering the composition patterns instead of the workflow controls themselves, Jaeger M. C. et al. are able to address many workflow controls defined in a variety of composition languages.

The difference of the work in this chapter compared to the work of others is that it focuses on QoS constraint formation in contrast to QoS computation. On the one hand, with QoS computation, the computing *formula* is not of importance as long as the QoS of the composition is calculated. The computation is often described as a black box which may consist of a series of algorithmic steps. QoS constraint formation, on the other hand, finds the computing formula and its expression. In QoS constraint formulation, we go

further than QoS computation by formulating the computation process as a mathematical expression. The expression uses mathematical operators to connect different QoS variables and shows the dependencies between them. Such an explicit expression of constraint is important for constraint satisfaction techniques in solving the QoS composition since knowledge of the constraint expressions is often crucial to speed up the search [Tsa93].

5.3 Aggregation formulas for QoS composition patterns

The QoS constraint construction in this chapter is to build the formula for constraints between the QoS value of a Web service composition and the QoS values of its component services. For our purposes:

- S_c denotes the Web service composition.
- S_1, $S_2 \ldots S_m$ denotes the components services of S_c.
- q denotes the name of the interested QoS parameter.
- q_k denotes the value of the parameter q for the service S_k, $k = 1..m$.
- q_c denotes the value of the parameter q for the service S_c.
- $f_{S_c,q} : (q_1, \ldots, q_m) \mapsto q_c$ is the constraint formula describing the relationship between q_k and q_c.

In this chapter, $f_{S_c,q}$ is constructed using simple *QoS aggregation formulas* for QoS composition patterns. A *QoS aggregation formula* for a composition pattern describes the relationship between the QoS values of various services present in the pattern. In order to make our management system non-specific to a particular domain, we assume that the aggregation formulas for the composition patterns are available from local domains. These QoS aggregation formulas can be used to build constraint formulas for more complex compositions with nested structures.

QoS aggregation formulas for composition patterns can be found in many studies and are often domain-specific. For example, in [CPE+06] Canfora G. et al. discuss some

domain-specific QoS parameters including the *color depth* of a photo service, the number of *credit cards* accepted from a payment service and the *refresh rate* of a temperature service. The aggregation formulas for the QoS parameters can be found in Table 5.2. In particular, the aggregation formula $f_{q,p}$ for the QoS parameter q and composition pattern p can be found in the cell at row labelled p and column labelled q. For instance, $f_{Sequence,colordepth} = min\{q_1, \ldots, q_m\}$. In other words, the color depth of a sequence of services is the minimum among the color depths of the services.

Pattern	Color Depth	Credit Cards	Refresh Rate
Sequence	$q_c = min\{q_1, \ldots, q_m\}$	$q_c = \bigcap_{k=1}^{m} q_k$	$q_c = min\{q_1, \ldots, q_m\}$
Loop	$q_c = q_1$	$q_c = q_1$	$q_c = q_1$
XOR-XOR	$q_c = q_k : pr(q_k) = max\{pr(q_1), \ldots, pr(q_m)\}$	$q_c = q_k : pr(q_k) = max\{pr(q_1), \ldots, pr(q_m)\}$	$q_c = \sum_{k=1}^{m} q_c.pr(q_c)$
AND-AND	$q_c = min\{q_1, \ldots, q_m\}$	$q_c = \bigcap_{k=1}^{m} q_k$	$q_c = min\{q_1, \ldots, q_m\}$
OR-OR	$q_c = q_k : pr(q_k) = max\{pr(q_1), \ldots, pr(q_m)\}$	$q_c = q_k : pr(q_k) = max\{pr(q_1), \ldots, pr(q_m)\}$	$q_c = \sum_{k=1}^{m} q_c.pr(q_c)$

Table 5.2: Aggregation formulas of constraints for simple composition patterns of domain specific QoS attributes: color depth, accepted credit cards, and refresh rate. $pr(q_k)$ is the probability that S_i will be executed in XOR and OR related patterns

In addition, the aggregation formulas of common QoS parameters for Web services can be found in Table 5.3 from [JRG04]. In this work, the upper bound value of cost, instantaneous value of throughput, and uptime probability are considered.

5.4 Algorithms for the QoS constraint formation

Given the input that consists of the composition structure and the aggregation formulas for different workflow control patterns, we propose here two algorithms to construct the constraint expressions for the composition. The first algorithm follows a bottom-up composition approach whereas the second one uses a top-down decomposition. For the composition presented in Figure 5.1, these algorithms output the constraint expressions in Figure 5.2 when the formulas in Table 5.2 are used:

Pattern	Cost	Throughput	Uptime probability
Sequence	$q_c = \sum_{k=1}^{m} q_k$	$q_c = min\{q_1, \ldots, q_m\}$	$q_c = \prod_{k=1}^{m} q_k$
Loop	$q_c = k.q_1$	$q_c = q_1$	$q_c = q_1^k$
XOR-XOR	$q_c = max\{q_1, \ldots, q_m\}$	$q_c = min\{q_1, \ldots, q_m\}$	$q_c = min\{q_1, \ldots, q_m\}$
AND-AND	$q_c = max\{q_1, \ldots, q_m\}$	$q_c = min\{q_1, \ldots, q_m\}$	$q_c = \prod_{k=1}^{m} q_k$
OR-OR	$q_c = max\{z : z = 1 - \sum_{y \in \mathbf{S}} y, \forall \mathbf{S} \in \mathbf{P}(\mathbf{T})\}$	$q_c = min\{\mathbf{P}(\mathbf{T})\}$	$q_c = max\{z : z = 1 - \prod_{y \in \mathbf{S}}(1-y), \forall \mathbf{S} \in \mathbf{P}(\mathbf{T})\}$
AND-DISC	$q_c = \sum_{k=1}^{m} q_k$	$q_c = min\{q_1, \ldots, q_m\}$	$q_c = \prod_{k=1}^{m} q_k$
OR-DISC		$q_c = min(\mathbf{P}(\mathbf{T}))$	

Table 5.3: Aggregation formulas of constraints for simple composition patterns of Sequence, AND and OR. In this table, $\boldsymbol{P}(\boldsymbol{T})$ is the power set of $\boldsymbol{T}$ - all combination of possible splits of a OR-related pattern.

5.4.1 Bottom-up QoS constraint formation

This section presents an algorithm for QoS constraint formation using aggregation formulas for QoS composition patterns. In this algorithm, constraints are constructed from the most inner compositions to the outer ones. An important notion in this algorithm is collapsibility [JRG04] of compositions (or sub-compositions). During the algorithm execution, a composition is said to be collapsible into a composite service if internally it does not contain any sub-compositions. For example, the sequences A and B, and the switch block in Figure 5.1 are collapsible sub-compositions at the beginning of the algorithm execution.

The algorithm is listed in Algorithm 1. In principle, it (i.e. CC_{bu}) searches for collapsible sub-compositions (Line 3) and replaces them with new composite services (Line 13). When a sub-composition is replaced with a composite service, the QoS values of this service are constructed (Line 9) according to the available aggregation formulas of composition patterns (Lines 5 to 7). In detail, QoS variables of these services are automatically created and assigned according to the composition patterns. The assigned expressions are then updated to contain only QoS variables of component services in the original composition. This process is repeatedly carried out until no workflow remains in the collapsible workflow candidate set (Line 4). Since there is a limited number of sub-compositions in *wf*, CC_{bu} will eventually reach Line 11 and terminate. The final formula of $c_{qos,wf}$ returned

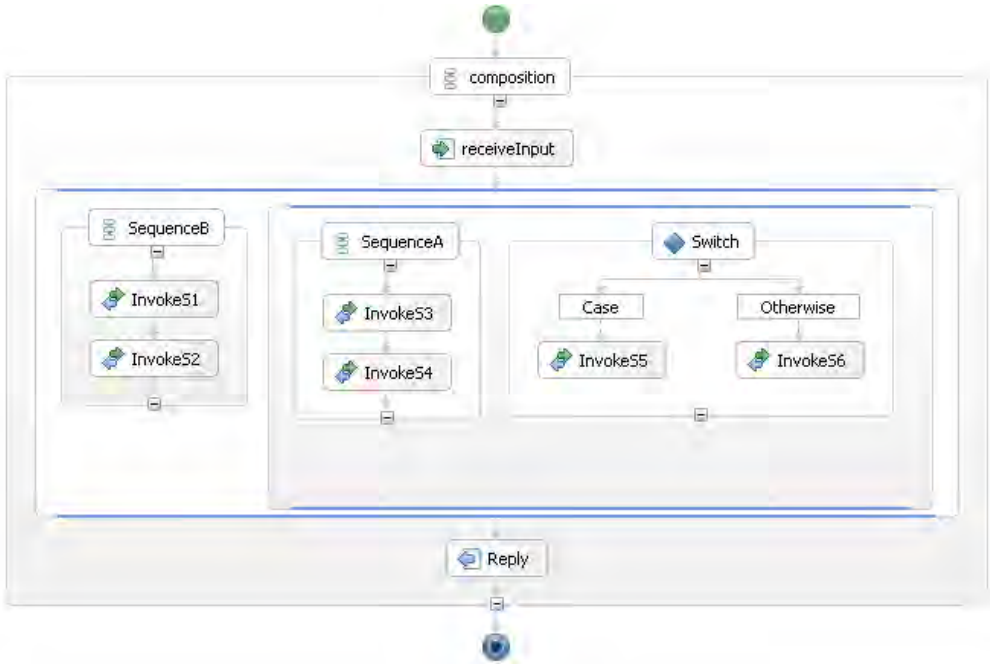


Figure 5.1: A composition visualization created with Eclipse BPEL. A flow activity (i.e. OR-OR or OR-DISC composition patterns) is enclosed by two parallel bars.

in Line 12 only contains QoS parameters of the component services in wf.

It is important to know that in order to avoid decomposing the same compositions again when checking for collapsible sub-compositions, the algorithm relies on the observation that a new collapsible composition at the *i-th* cycle (of the loop at Line 4) must contain a collapsible composition detected in the previous *(i-1)-th* cycle. With this observation, new collapsible compositions are only searched for from the set of parents of previously collapsed compositions (Line 15).

The correctness of the above observation can be easily proved by contradiction. If we assume that the observation is wrong, then there exists a collapsible composition $wf_{collapsible}$ at the *i-th* cycle, and $wf_{collapsible}$ does not contain any collapsible composition at the *(i-1)-th* cycle. However this would indicate that $wf_{collapsible}$ should have been detected and collapsed before the *i-th* step. This is a contradiction and hence the correctness of the observation is proved.

Cost constraint expression:
$q_c^{(cost)} = q_1^{(cost)} + q_2^{(cost)} + max\{q_5^{(cost)}, q_6^{(cost)}\} + q_3^{(cost)} + q_4^{(cost)}$

Throughput constraint expression:
$q_c^{(tput)} = min\{q_1^{(tput)} + q_2^{(tput)}, min\{max\{q_5^{(tput)}, q_6^{(tput)}\}, q_3^{(tput)} + q_4^{(tput)}\}\}$

Uptime probability expression:
$q_c^{(uptime)} = q_1^{(uptime)}.q_2^{(uptime)}.min\{q_5^{(uptime)}, q_6^{(uptime)}\}$

Figure 5.2: Constraint formulas for the example in Figure 5.1.

Algorithm 1: $c_{qos,wf} = CC_{bu}(qos, wf)$ – Bottom-up Constraint Construction

1 $c_{qos,wf}$=*null*;
2 **WF** = $\emptyset$; `/* A set of collapsible compositions */`
3 Add all collapsable sub-compositions in wf into **WF** ;
4 **while** $\{null\} \neq$ ***WF*** **do**
5 **foreach** $w \in$ ***WF*** **do**
6 **if** w *is collapsible* **then**
7 find the set $\mathbf{S}_w$ of all services in w ;
8 find the composition pattern $p(w)$ of w ;
9 look up the QoS aggregation formula for $f_{qos,p(w)}$;
10 construct the constraint $c_{qos,w}$ from $f_{qos,p(w)}$ by substituing c_{qos,s_w} for $q_{s_w}^{qos}$ $\forall s_w \in \mathbf{S}_w$;
11 **if** w *is* wf **then**
12 return $c_{qos,wf}$;
13 collapse w into a service ;
14 find w_{parent} - the immediate parent composition of w;
 `/*` w_{parent} `is` ***null*** `if` w `is` wf `*/`
15 **WF** = (**WF** $\setminus w$) $\cup \{w_{parent}\}$;

For the example in Figure 5.1, the algorithm will be executed through the following

steps for the QoS parameter of cost:

1. The switch composition collapses into a service S_{switch} and
   $q_{switch}^{(cost)} = max\{q_5^{(cost)}, q_6^{(cost)}\}$

2. *sequenceA* collapses into a service $S_{sequenceA}$ and
   $q_{sequenceA}^{(cost)} = q_3^{(cost)} + q_4^{(cost)}$

3. *sequenceB* collapses into a service $S_{sequenceB}$ and
   $q_{sequenceB}^{(cost)} = q_1^{(cost)} + q_2^{(cost)}$ (*Finish the first cycle of the loop* line 2 in Algorithm 1)

4. The inner flow which contains *sequenceA* and the switch collapses into a service $S_{innerFlow}$
   and $q_{innerFlow}^{(cost)} = q_{sequenceA}^{(cost)} + q_{switch}^{(cost)}$
   $= max\{q_5^{(cost)}, q_6^{(cost)}\} + q_3^{(cost)} + q_4^{(cost)}$ (*Finish the second cycle of the loop*)

5. The outer flow which contains $S_{innerFlow}$ and *sequenceB* collapses into a single service and
   $q_c^{(cost)} = q_{innerFlow}^{(cost)} + q_{sequenceB}^{(cost)}$
   $= q_1^{(cost)} + q_2^{(cost)} + q_3^{(cost)} + q_4^{(cost)} + max\{q_5^{(cost)}, q_6^{(cost)}\}$

6. The algorithm terminates and the final result agrees with Figure 5.2.

5.4.2 Top-down QoS constraint formation

The top-down QoS constraint formation algorithm builds the constraint formulas by repeatedly decomposing a composition and replacing the QoS parameters of this composition with those of inner compositions. The algorithm is shown in Algorithm 2 below. In the algorithm, a list of compositions is maintained in the variable **WF** which initially contains only the input composition wf (Line 1 - Algorithm 2). The compositions in this list will be decomposed at the next round of the *while* loop at Line 2. At any moment, the temporarily constructed formula $c_{qos,wf}$ has only variables of QoS parameters of compositions in **WF**. As those compositions are further broken down, they are removed from **WF** and new compositions are added (Line 10). At the same time, $c_{qos,wf}$ are updated respectively

according to this new list of **WF** (Line 9). When all compositions in **WF** have been decomposed (the termination condition for the *while* loop at Line 2), the final formula of $c_{qos,wf}$ will have only QoS parameters of the component services in wf.

Algorithm 2: $c_{qos,wf} = CC_{td}(qos, wf)$ – Top-down Constraint Construction

```
1  WF = {wf} ;
2  while ∃w ∈ WF: w has sub-compositions do
3      if c_{qos,wf} is null then
4          find the composition pattern p(wf) of the top composition in wf ;
5          c_{qos,wf} = f_{qos,p(wf)};
6      find the set S_w of all first sub-compositions in w ;
7      find the composition pattern p(w) of the top composition in w ;
8      look up the expression of f_{qos,p(w)} from the list of QoS aggregation formulas ;
9      in c_{qos,wf} replacing q_w^{qos} by f_{qos,p(w)} ;
10     WF = WF \ {w} ⋃ S_w ;
11 return c_{qos,wf} ;
```

For the example in Figure 5.1, the algorithm execution steps are:

1. $q_{wf}^{(cost)} = max\{q_{sequenceB}^{(cost)}, q_{innerFlow}^{(cost)}\}$ (*After the first cycle of the loop*)
2. $q_{wf}^{(cost)} = max\{q_1^{(cost)} + q_2^{(cost)}, max\{q_{sequenceA}^{(cost)}, q_{Switch}^{(cost)}\}\}$ (*After the third cycle of the loop*)
3. $q_c^{(cost)} = q_1^{(cost)} + q_2^{(cost)} + q_3^{(cost)} + q_4^{(cost)} + max\{q_5^{(cost)}, q_6^{(cost)}\}$ (*After the fifth cycle of the loop*)
4. The algorithm terminates and the final result agrees with Figure 5.2.

5.4.3 Comparison of the two algorithms

Whilst both algorithms are simple, the Bottom-up Constraint Construction algorithm has higher complexity than the Top-down Constraint Construction algorithm. To quantify the complexity of these two algorithms, we can identify two basic operations used in the

algorithm executions: UPDATE and CHECK. UPDATE is to lookup the workflow pattern formulas and update a temporary constraint formula constraint (used in lines 5 to 7 in Algorithm 1, and 6 to 9 in Algorithm 2). CHECK is to verify whether a sub-composition is collapsible (used in line 2 of Algorithm 1). Let us use $N(wf)$ to denote the number of sub-compositions in wf then the both algorithms need $N(wf)$ UPDATE operations in total to lookup and update the temporary constraint formulas. Additionally in the first algorithm, checking the new collapsible compositions from the set of parents of previously collapsed compositions requires additional $N(wf)$ CHECK operations in order to complete its execution. Therefore, while an agent can use one of these algorithms to find QoS constraints at the Web service layer, the Top-down Constraint Construction algorithm is considered to be more efficient than the Bottom-up Constraint Construction algorithm.

5.5 Constraint Privacy

In addition to the expressions and formulas of constraints, the privacy of constraints is important in DisCSP as discussed in Chapter 3. An agent does not wish to share its constraints with others unless the agent is required to do so. In the previous section, two algorithms have been proposed to build the induced QoS constraints from functional compositions. However, there has been no discussion on the privacy of the constraints. The following rules can be used to determine which parties a constraint is visible to:

- ***Rule 1:*** If the input composition (in Algorithm 1 and 2) is an orchestration then the QoS constraints are constructed by the orchestration composer and privately available to this composer only.

- ***Rule 2:*** If the composition is a choreography then the QoS constraints can be constructed by any or all participants in the choreography and available to all of them.

In practice, a Web service can be an orchestration itself and be involved in a number of choreographies. In this case, the constraints constructed from the orchestration are private to the Web service provider. At the same time, the constraints constructed

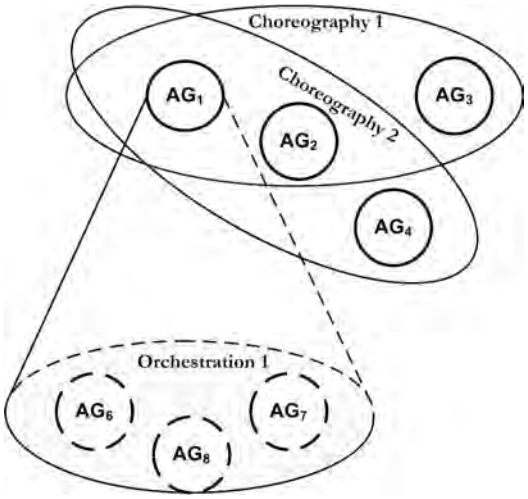


Figure 5.3: Constraints are visible differently to different agents. Denote $C(wf)$ as the set of constraints constructed from the composition wf then AG_1 knows $C(Choreography_1)$, $C(Choreography_2)$, and $C(Orchestration)$, AG_2 knows only the first two constraint sets. AG_3 knows $C(Choreography_1)$ and AG_4 knows $C(Choreography_2)$. AG_6, AG_7, and AG_8 know nothing about these constraints.

from a choreography are shared between participants in this choreography but invisible to participants in another choreography. An example is shown in Figure 5.3 where 8 agents AG_1, AG_2,...,AG_8 are involved in a number of compositions. Each agent provides a Web service. AG_1 has its service in two choreographies 1 and 2. AG_1 also has its service as an orchestration made up by services from AG_6, AG_7, and AG_8. In this case, the constraints constructed from Choreography 1 or 2 are shared between participants in the choreography, whereas the constraints constructed from Orchestration 1 are AG_1's private constraints.

5.6 Summary

This chapter has focused on one important class of constraints at the Web service layer: constraints which are formed from the composition structure. These constraints often express the relationships between the end-to-end QoS of a composition and the QoS values of component services in the composition. Two algorithms, Bottom-up Constraint Construction and Top-down Constraint Construction, have been proposed for finding these

constraints. The visibility of those constraints to DisCSP agents has also been discussed.

It is important to note that the QoS constraints discussed in this chapter constitute a part of the agents' local CSP. Thus they are an imperative part of the problem that needs to be solved by agents in order to guarantee QoS for Web services. Solving this problem with different DisCSP algorithms is the main topic for the remainder of the thesis.

Chapter 6

QoS Management using DisCSP algorithms*

6.1 Introduction

A distributed framework for QoS management of Web services has been proposed in Chapter 3. In the framework, management tasks are modeled as DynDisCSPs that can be solved using a DisCSP or DynDisCSP algorithms. A DynDisCSP algorithm is designed to solve DynDisCSPs whereas a DisCSP algorithm, in combination with a restart strategy, can also be used for the same purpose. A restart strategy solves DynDisCSPs by restarting a DisCSP algorithm everytime a change to the DynDisCSP is detected.

Designing DynDisCSP and DisCSP algorithms that can be practically used for the QoS management in the Web service domain is important for the proposed framework. This chapter investigates how existing DisCSP algorithms can be adapted for this purpose. Specifically, it studies Asynchronous Backtracking (ABT) [Yok95] – a popular DisCSP algorithm, and Asynchronous Aggregate Search (AAS) [SF05] – an extension of ABT. The chapter is organized as follows: The ABT and AAS algorithms are described in Section 6.2. Section 6.3 addresses different assumptions made for the algorithms, especially those which are impractical in the Web service environments. Section 6.4 proposes an efficient

*A portion of the work presented in this chapter has been published in [NKP06c].

heuristic that exploits the commonly monotonic preference of service providers for QoS parameters to *improve* ABT's performance. Finally, Section 6.5 concludes the chapter.

6.2 Distributed Constraint Satisfaction Algorithms

This section reviews two DisCSP algorithms – ABT (Asynchronous Backtracking Search) and AAS (Asynchronous Aggregate Search). ABT has been one of the most popular DisCSP algorithms. Many algorithms (e.g. Asynchronous Weak-Commitment (AWC) [Yok95], Asynchronous Backtracking with Reordering (ABTR) [SSF01], Asynchronous Backtracking with Dynamic Ordering (ABT-DO) [ZM06]), are extended from ABT. A successful application of ABT thus can open the way for those algorithms to be potentially used for QoS management too. With this in mind, we describe AAS in this chapter as a selected example of ABT extension. Most of the discussion for ABT is also applicable for AAS.

6.2.1 ABT algorithm:

ABT is a complete algorithm which was proposed by Yokoo [Yok95] to solve DisCSPs. In ABT, each agent owns only one variable. Two agents are neighboring if they share some constraints. Before the algorithm starts, agents are assigned different priorities. Those priorities can be integer numbers. The lower the number assigned to an agent A_k, the higher priority A_k has. For the clarity of the discussion, let $\mathbf{A}_k^+$ denote the set of A_k's neighbors whose priorities are higher A_k, and $\mathbf{A}_i^-$ the set of neighbors whose priorities are lower than A_k.

In ABT, different message types are defined as follows:

- *ok?* message: To propose an assignment for a variable. An assignment is a pair (x_i, v_i) where x_i is a variable and v_i is a proposed value for the variable. If A_i is the owner of x_i then an *ok?* message is sent by A_k to $\mathbf{A}_i^-$.

- *nogood* message: To reject a proposal. If an agent A_j previously sent a proposal to another agent A_i, A_i can reject the proposal. This happens when either A_i's

assignment is in conflict with the assignments from some higher priority neighbors of A_i or the assignment leads to overconstraints for A_i's local CSP. The rejected assignment from A_j is referred to as a *nogood*.

- *addlink* message: To add a new neighbor to an agent. It can happen in ABT that a conflict arises beween the assignments of two agents A_i and A_j which do not know each other. In addition, the conflict cannot be easily solved by other agents. *addlink* messages are used to link A_i and A_j together so that they can directly communicate and resolve the conflict.

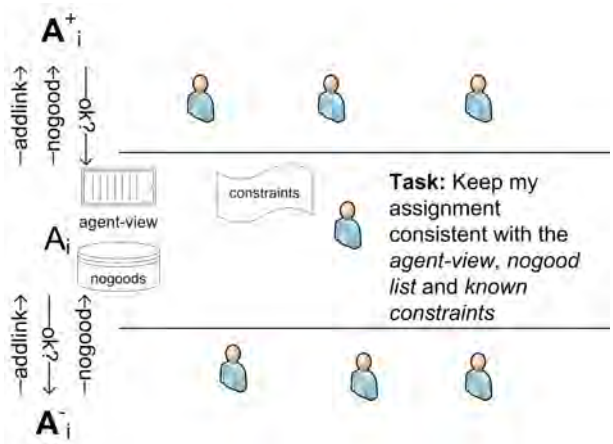


Figure 6.1: Message exchanges in ABT from the perspective of a single agent A_i. The agent receives ok? messages from higher priority neighbors and nogood messages from lower priority neighbors.

Additionally, each agent A_i in ABT keeps an *agent-view* to hold assignments received from $\mathbf{A}_i^+$, and a *nogood list* to keep nogoods (i.e. invalid assignments) reported by $\mathbf{A}_i^-$. The task of the agent is to find a *consistent* assignment for its variable (see Figure 6.1). An assignment is said to be *consistent* if it does not resemble any *nogood*, and together with the assignments in the *agent-view*, satisifies all A_i's constraints.

ABT executes as follows: In the beginning, every agent concurrently finds a valid assignment and sends *ok?* messages to propose the assignment to its lower priority neighbors. An agent A_k, upon receiving an *ok?* message, updates its *agent-view* and checks whether

its current assignment is *consistent*. If not, a new *consistent* assignment is generated and new *ok* messages containing the assignment are sent to $\mathbf{A}_i^-$. If such a consistent proposal cannot be found, A_k does backtracking by sending a *nogood* message to its lowest upper priority neighbor. Upon receiving the *nogood* message, the receiver agent may generate a new proposal or has to backtrack the inconsistency further upstream (i.e towards higher priority agents) until this inconsistency can be resolved completely. During a backtrack, an agent may receive a *nogood* which contains variables that it does not know. In this case, the agent sends *addlink* messages to become the neighbors of those variables' owners so that they can communicate directly and resolve any conflict together. A solution in ABT is reached when every agent's assignment is consistent and no further message is sent between the agents. More information about the ABT algorithm and a proof of its completeness can be found in [Yok95].

6.2.2 Asynchronous Aggregate Search:

AAS was proposed by Silaghi et al. [SF05] as an extension of ABT. AAS extends ABT in the following aspects:

- In contrast to ABT in which one agent controls only one variable, agents in Asynchronous Aggregate Search (AAS) can control multiple variables. Specifically, an agent controls any variable that it knows. AAS is thus designed for more general DisCSPs. In AAS, two agents are neighboring if they share some variables.

- Agents propose a single value for a variable in ABT. In AAS, agents propose multiple values in an interval for a variable. This can significantly reduce the communication cost between agents [SF05] since multiple ABT (*ok?* or *nogood*) messages can be aggregated into a single AAS message.

Similar to ABT, AAS agents are assigned with random priorities at the begining. *ok?*, *nogood*, and *addlink* messages are also used in AAS for the same purposes as in ABT. Morever, the search process in AAS is the same as that in ABT. During the search, each agent A_i sends assignments in *ok?* messages to $\mathbf{A}_i^-$ and rejections in *nogood* messages

to $\mathbf{A}_i^+$. It is worth noting the difference in generating assignments between ABT and AAS. An agent in ABT generates assignments for the variable it owns. An agent in AAS generates assignments for variables which satisfy the two following conditions: i) The agent can control the variables, and ii) The agent has the highest priority among agents which can control the variables. Those conditions avoid the agents repeatedly generating assignments for the same variable at the same time.

In AAS, an agent A_i also keeps an *agent-view* and a *nogood list*. A_i's task is to find an assignment for its variables so that the assignment is *consistent* with the agent's local constraints, *agent-view*, and *nogood list*. It is important to note that consistency in AAS is interval-based. In particular, an assignment generated by A_i is consistent with its local constraints, *agent-view*, and *nogood list* only if every value in the assignment's interval is consistent with those.

A detailed explanation of the algorithm can be found in [SF05]. Also, termination, correctness and completeness of AAS, are proven in that work.

6.3 Adaptation of ABT and AAS for QoS management

It is important to note that ABT and AAS have been designed for general DisCSP, whereas in our QoS management, the focus is specifically on the Web service domain. The following assumptions for ABT and AAS, that can be impractical in the Web service domain, are:

- The assumption of a single variable per agent and simple local constraints in ABT inadequately represents the QoS management problem, in which each management agent can have several variables and complex local constraints.

- Most DisCSP algorithms assume message sending and receiving are FIFO, i.e. message ordering, and no message lost in transit. This is impractical in the Web service environments.

- ABT and many of its extensions lack of a termination detection mechanism. Those algorithms terminate when there are no further messages exchanged between agents.

This condition, referred to as the agents' quiescent state [SF05], is difficult to detect in a distributed environment.

- The assumption that agents in the proposed framework are already ordered according to some priorities so that ABT and AAS can be applied directly.

In the next few sections, the above assumptions of ABT and AAS algorithms are investigated and addressed so that the algorithms can be practically used in the Web service environment for QoS management.

6.3.1 Agent Priorities

As discussed before, ABT and AAS agents are ordered according to priority. The priorities do not imply any social inequality between the agents. Nevertheless, they are important to ensure the completeness of ABT and AAS. In the proposed QoS management framework, priorities can be assigned to agents during the DisCSP Network Setup phase (see Chapter 4). In this phase, agents are invited or invite other agents to join the management coalition. The priority assignment can be carried out as follows:

- The initiator agent A_{ini} (i.e. the first agent in the coalition) is assigned the highest priority of value 1. This agent keeps a counter ξ of the current number of agents in the coalition. $\xi = 1$ at the beginning.
- If an agent A_i is invited to join the coalition and A_i accepts the invitation, A_i then sends a message to A_{ini} to ask for permission to join. A_{ini} can either:
  - Accept A_i into the coalition and set $\xi + 1$ as the priority for A_i. A_{ini} assigns this priority to A_i by sending a message to A_i together with the priority value. It increases ξ by 1 after that.
  - Reject A_i from joining the coalition. This happens when *count* is equal to the maximum allowable number of agents in the coalition.

With the above simple scheme, late joining agents have lower priorities. An agent A_i is considered for joining the coalition after another agent A_j, if its request for joining

permission arrives A_{ini} later than the request from A_j. If multiple requests arrive at A_{ini} at the same time, A_{ini} randomly assigns the priorities among their senders. Since ξ is increased every time an agent joins the coalition, different agents have different priorities in the coalition.

Another possible scheme is to convert the agents' addresses to strings and use them as the agents' priorities. Lexical comparison can be used to compare strings. Obviously the agents should have distinct addresses in order to be identified during communications. This also ensures that different agents have different priorities in the coalition. In this scheme, invited agents should still request permission from the initiator A_{ini} to join the coalition. A_{ini} does not need to assign priorities to the agents; it only needs to check whether the coalition size has reached the maximum allowable limit.

6.3.2 Multi-variables

AAS has been designed for multi-variables per agent and *n-ary* constraints. However, ABT is not. Nevertheless ABT can be extended to have multiple variables per agent. Yooko suggests two approaches in [Yok01] for such an extension:

- In the first approach, if an agent A_i owns more than one variable, new agents are introduced as internal components of A_i to control the variables. In addition, each newly introduced agent is the owner of exactly one variable in A_i. For example, if an agent A_i has two variables x_p and x_q then A_i can be considered as comprising two intra-agents A_{i_1} and A_{i_2} where A_{i_1} controls x_p and A_{i_2} controls x_q. The ABT algorithm therefore can be carried out by replacing A_i with its internal agents.

- In the second approach, multiple variables in an agent can be grouped together into a new aggregated variable. The domain of this new variable is the cross-product of the existing variables' domains. Each agent therefore, can be considered as having only one variable, thus ABT can be applied. For the previous example, A_i is considered as having one aggregated variable $x_i = \langle x_p, x_q \rangle$

Either of the above approaches can be used to adapt the ABT algorithm to our proposed framework. For the first approach, instead of one agent, multiple agents can be used to represent an AS. Each agent owns a QoS parameter or resource variable which can be controlled by the AS. This approach has the disadvantage that it requires modelling the local CSP of the AS as a distributed problem between the AS's internal agents. Centralized CSP algorithms are more suited to solving the local CSP. Therefore, it is preferred in this thesis to use the second approach. With the second approach, QoS parameters in the AS are aggregated into a single variable that is controlled by an agent representing the AS.

6.3.3 Local solvers and solution aggregation for AAS

ABT assumes that the agent's local constraints are simple. As such, there has been a lack of effort in solving DisCSP with complex local constraints [DBL04]. However, QoS related constraints in the Web service environments can be complex. This is especially true when ABT is used and multiple variables are grouped together. Consequently, the complexity of finding local solutions can no longer be neglected. To handle the complexities of local constraints, centralized CSP solvers are proposed within each agent in the framework. Every time an agent needs to find a consistent assignment, it invokes the local CSP solver. The CSP solver of an agent A_k inputs assignments from $\mathbf{A}_k^+$ and generates solutions for variables shared with $\mathbf{A}_k^-$. Examples of CSP solvers in practice are ILOG JSolver [ILO05], Choco [RHF+05], and NSolver [Chu05]. Different ASs can select different CSP solvers as long as they can convert the output from those solvers into a common format.

Using CSP solvers inside agents simplifies the local CSP solving inside each agent and provides greater flexibility for the agent's implementation. However, there are certain obstacles when using local CSP solvers with AAS. As discussed before, an agent in AAS proposes an interval (i.e. consecutive values in a domain) for a variable. Thus, an assignment for multiple variables is sent between agents in the form of a Cartesian product of intervals which represents a set of possible valuations. This product is referred to in this thesis as a box for short. An AAS agent needs a special solving algorithm, in order to produce boxes of solutions for its local CSP (e.g. [SSF99]). However, in general there is no

guarantee that output from a CSP solver will be boxes. Therefore, for any CSP solver to be used with AAS, it is neccessary to have an algorithm which can construct boxes from a set of solutions output by the solver. In addition, since only a box is sent in an *ok?* message, it is desirable that the algorithm returns a minimal number of boxes in order to reduce the number of messages sent. Such an algorithm, called *Greedy Box Construction* is proposed in Algorithm 3. In the Greedy Box Construction algorithm, the input is a set of solutions. A solution is represented as an n-dimensional point. The coordinator value for the n-th axis of the point is the value of the n-th variable in the variable list owned by an agent. The algorithm executes as follows: It picks up a point in the solution set and systematically extends the point into a box along each dimension (i.e. variable domain) as long as all points contained in this box are solutions (Lines 2 to 4). If the box cannot be extended any further, all solutions in the box's volume are removed from the solution set and the whole process is started again (in the *while* loop) until the solution set becomes empty. The algorithm returns the list of distinct boxes which cover the whole solution set.

Algorithm 3: Greedy Box Construction

```
   /* boxes = greedyBoxConstruct(S)                                  */
   Input: S
   /* S: The input of solutions.                                     */
   Output: boxes
   /* boxes:  The minimal set of boxes returned.                     */
1  while S ≠ ∅ do
2      Pick any point s in S ;
3      box = {s} ;
4      foreach dimension of s do
5          Expand box along the dimension until a point that does not belong to S is
           encountered;
6      boxes=boxes∪{box} ;
7      S = S \ box ;
8  return boxes ;
```

It can be seen from Algorithm 3 that *Greedy Box Construction* has to examine every solution in **S**. Therefore, its complexity is $\boldsymbol{O}(\log k)$ where k is the number of solution in **S**. In fact, since an agent does not have to find all solutions each time the local CSP is invoked, it can set an upper bound on k. Consequently, the complexity of *Greedy Box*

Construction in this case can be considered as a constant.

6.3.4 Message ordering, communication reliability, and failure detection

Message ordering, reliability, and network failure detection are other important issues to consider when using DisCSP algorithms in Web service environments. Most of the DisCSP algorithms, including ABT and AAS, assume that messages arrive in the order that they are sent. At the same time, they make an assumption that no messages are lost during transit. These assumptions can be questionable for applications in distributed environments. This section discusses different mechanisms to support message ordering, communication reliability, and network failure detection for Web service environments.

In order to ensure message ordering for any DisCSP algorithm in general, *identitifiers* of DisCSP solving sessions in combination with *sequence numbers* of messages are proposed as follows:

- Each DisCSP solving session (i.e. an instance of ABT or AAS execution) is assigned a distinctive identifier.
- Before an agent sends out a message in a solving session, it assigns a sequence number seq to the message. This number is 0 at the beginning of the session and increases by 1 after the message is sent. Both the session identifier and the message sequence are sent with the message.
- The agent keeps a buffer to store received messages. Messages with $seq = 0$ are processed first by the agent in a solving session. The agent only processes a message m_1 with $seq = k$ if it has already processed another message m_2 with $seq = k - 1$. In addition, m_2 needs to be from the same sender and has the same seq with m_1.

Using the above steps, messages sent from a sender agent can be processed in the right order by the receiver agent. It is worth noting that the solving session identifiers are used in combination with the sequence numbers in order to allow a group of agents to participate in multiple DisCSP solving sessions in parallel.

To handle reliability of message delivery, the *sliding window* technique developed for TCP protocol [Com95] can be used. The technique is briefly described as follows [Com95] :

- Both sender and receiver agents maintain a finite size buffer (i.e. window) to hold outgoing and incoming messages from each other.
- Acknowledgement is required for every sent message. A sender agent may send many messages continuously until the number of unacknowledged messages is as large as its window size.
- The receiving agent can send a single acknowledgement message for multiple received messages, embedded with the agent's window size. This window size tells the sender agent to increase or decrease the number of messages in the next transmission appropriately.

The above mechanisms can only assure DisCSP algorithms execute correctly in a network without failures, such as process crashes or connectivity break-downs. If one of those failures happens, the algorithms will eventually stop sending messages. A different mechanism is therefore required to detect network failures. Such a mechanism can be implemented using *keep-alive* messages. These *keep-alive* messages are periodically sent between any two neighboring agents. An agent listens for active *keep-alive* messages from its neighbors. If a neighbor stops sending this message after a pre-defined time period, an event of failure will be raised. The sender agent then propagates a failure message through its neighbors and neighbors of neighbors. Agents which successfully receive the message will stop executing. It is important to note that due to the *keep-alive* messages, eventually every agent will experience a timeout or be notified about a failure in some part of the network.

6.3.5 Termination Mechanism for ABT

In ABT a solution is reached when the agent system is in *quiescent* state. This state is achieved when every agent has its local constraints satisfied and no more messages

exchanging between the agents [Yok95]. Detecting this state is very important when ABT is applied for the QoS management task. Specifically for the management task of QoS composition, every agent should be able to learn about the algorithm termination in order to use the final solutions. Similarly, during the QoS (re-solving) process, the agents should be able to decide on the new values of QoS parameters. However the *quiescent* state cannot be easily detected by agents separately since an agent has no knowledge of other agents' solving status.

A naive approach to detecting the *quiescent* state is to ask an agent to broadcast its local solving state to all other agents whenever its local CSP is solved. This approach needs to synchronize the solving status among all agents and hence may create a huge amount of traffic. This is because during the solving, an agent often undergoes a large number of local satisfaction states. A better way to detect the *quiescent* state is thus required for ABT.

This section proposes a termination detection mechanism for ABT. This mechanism is based on the mechanism designed for AAS in [SF05]. To determine whether a running ABT algorithm has been terminated, two modifications are introduced into ABT. The first modification requires that before the search starts, every *top* agent adds a link to the *highest priority* agent (i.e. the initiator agent in the framework) using *addlink* messages. A *top* agent is an agent that has no higher priority neighbors. Recall that in ABT, normally agents are linked if they share some constraints. However, a link between a *top* agent and the highest priority agent does not force the two agents to share any constraints but is solely for the purpose of direct communications. The second modification introduces a new message type called *good*. A *good* message indicates that a proposal from some higher priority neighbor of A_i enables valid assignments to be found at A_i and all of A_i's descendants. This *good* message type is apparently a counterpart of the ABT *nogood* message type.

The proposed termination procedure of ABT is listed in Algorithm 4. In general, agents wait for *good* messages from lower priority neighbors and send *good* messages to higher priority neighbors. Every agent A_i after generating a new asignment, collects *good*

messages from its descendents and aggregates their contents into *partial_sol* (Line 6). If a *nogood* is reported for the current assignment, this *partial_sol* is cleared (Line 13) and any *good* messages reported later for this assignment are simply ignored (Line 2). If no such *nogood* message is received, and the descendent subtree rooted at A_i (referred to as *subtree* parameter in Line 7) is subsequently solved, then A_i sends *good* messages to its parents. The subtree is solved if A_i's *partial_sol* has valid assignments from all of its neighbors and these values are consistent with A_i's view (Lines 16 and 17). A solution is reached when A_i is the highest priority agent and its subtree is solved (Lines 9 and 10). Clearly, this solution is correct since in this case all other agents are A_i's descendants and their assignments are consistent with each other and with A_i's.

Algorithm 4: ABT Termination Procedure at an agent

```
1  when receiving a good message m do
2      if nogood_reported is true then
3          return;
4      else
5          if good assignment is my current solution then
6              update and aggregate this good assignment into the partial_sol;
7              if the subtree is solved then
8                  send partial_sol in good messages up to higher prio neighbors;
9                  if this is the highest priority agent then
10                     announce a solution found;
11 when receiving a nogood message do
12     if nogood assignment is my current solution then
13         clear partial_sol;
14         nogood_reported = true;
15 when check agent view do
16     if the view is consistent and subtree is solved then
17         send up a good message to every parent;
18     else
19         if a new assignment found then
20             nogood_reported = false;
```

6.4 Improving ABT performance using a monotonic QoS–based heuristic

6.4.1 The monotonic QoS–based heuristic

This section proposes a heuristic that can be used to improve the performance of ABT (and possibly its extensions) for QoS management in Web service environments. In the context of Web service composition, QoS parameters have particular characteristics. Specifically, Web service providers (and thus their representative agents) often have preferences as a monotonic function over the values of a QoS parameter. In other words, there often exists a common preference comparator $\preceq$ defined between any two values a_{k_i} and a_{k_j} in the domain of a QoS variable x_k. If $a_{k_i} \preceq a_{k_j}$ then we say that the agent prefers a_{k_i} to a_{k_j} for x_k. For example, provided every other QoS parameter is kept the same then it is often that:

- A provider agent prefers higher price for its service.
- A provider agent prefers less management overhead and hence less reliability and security levels for its service (provided that the price is kept unchanged).
- A provider agent may prefer less resource consumption on its host and hence consequently longer execution time of its service (provided that the price is kept unchanged).

In general, the preference for an agent can be defined between two assignments as follows:

$$\bigwedge_m^{k=1}(\mathrm{x}_k{=}\mathrm{a}_{k_i}) \preceq \bigwedge_m^{k=1}(\mathrm{x}_k{=}\mathrm{a}_{k_j}) \text{ if } \forall \mathrm{k}{=}1..\mathrm{m}\ \mathrm{a}_{k_i}{\preceq}\mathrm{a}_{k_j}.$$

If such a relation cannot be defined due to $\mathrm{a}_{k_j}{\preceq}\mathrm{a}_{k_i}$ for some i, then the assignments are said to be *incomparable*.

It can be observed that the above preferences measure the easiness for an agent to support certain QoS level. Particularly, the higher the preference for a QoS level is, the easier this level can be supported. The observation can be used in a heuristic that guides the agents' local searches for new assignments. When a new consistent assignment is generated the heuristic criterion could be *to choose an assignment that has lower preference*

as compared to the previous rejected assignments. This typically speeds up the searches. The reason is as follows: If an assignment sent by A_i to A_j has been rejected, A_i should avoid being rejected again by reducing its own preference and hence giving A_j and other lower priority agents higher preference and thus more chance to find their consistent assignments[1]. To execute the heuristic, A_i first adds a new constraint into its local CSP solving. The constraint is that its assignment should have less preference than all previous rejected assignments in the *nogood* lists. A_i then solves its local CSP and sends the newly found assignment to its lower priority neighbours. If no solution can be found, the constraint is removed from A_i local CSP so that ABT can be executed as normal.

6.4.2 Experiments

The monotonic QoS based heuristic is very simple, as explained in the previous section. Nevertheless, it can provide much better performance when compared to the original ABT. We have carried out experiments in which two versions of ABT are used: One with the heuristic and one without the heuristic proposed in the previous section.

In the experiments, there are n agents which are randomly allocated into compositions so that each of them belongs to 3 or 4 compositions. Each composition consists of 3 agents [2]. The number of agents n varies from 8 to 30. For a composition, three QoS parameters: *cost, reliability* (*log* scale), and *execution time* are considered. Each parameter has the domain of 10 discrete values. The client's QoS requirements for each composition are: i) the end-to-end *cost* should be at most $\frac{3}{2}c_{max}$, ii) the end-to-end *reliability* should be at least $\frac{3}{2}r_{max}$, and iii) the end-to-end *execution time* should be at least $\frac{3}{2}t_{max}$ where c_{max}, r_{max}, t_{max} are the maximum values in the domains of cost, reliability, and time respectively. With these settings, the tightness for the end to end constraint for each QoS parameter is 50%. In addition, different values for constraint tightness of the local CSPs inside the agents are used in the experiments. In particular, those tightnesses are set to 0.8 (hard), 0.5 (medium), and 0.1 (easy). A hard problem likely leads to overconstraint.

[1]Note that the end to end QoS requirements are often fixed in an ABT or AAS session. Increasing in preference of one agent potentially reduces the preference of other agents.

[2]Around *1.5n* compositions in total are required to achieved this.

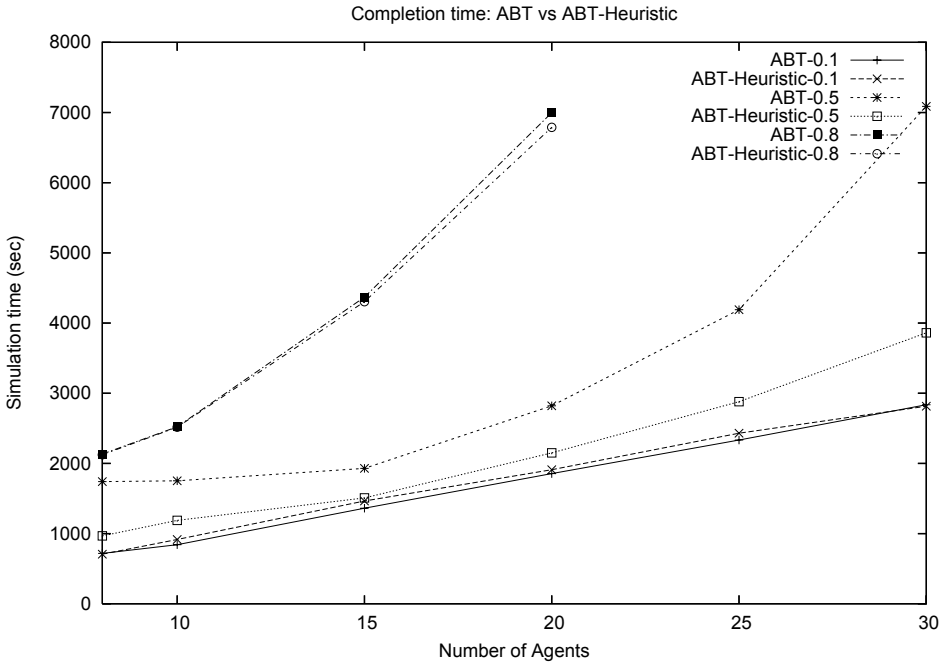


Figure 6.2: Completion time: ABT versus ABT using QoS-based Heuristic.

Figure 6.2 and 6.3 plot the experiment results. Figure 6.2 depicts completion time where as Figure 6.3 illustrates the number of local searches (i.e. number of times the local CSP solver inside each agent is invoked) for the two algorithms. Each data point is the average of 10 different runs. The figures show that the completion time and total number of local searches required in both cases are quite similar for hard and easy problems as the plotted lines for the two algorithms closely resemble each other in the figures. However, for a medium difficult problem of tightness=0.5, ABT with the heuristic can cut down around 40% for completion time and 25% for total local searches. Therefore, the heuristic can help to speed-up ABT as well as reduce its computational overhead. The experiment results can be explained as follows. For easy problems, the number of backtracks is minimal for both ABT versions and hence no significant improvement can be expected. For hard problems,

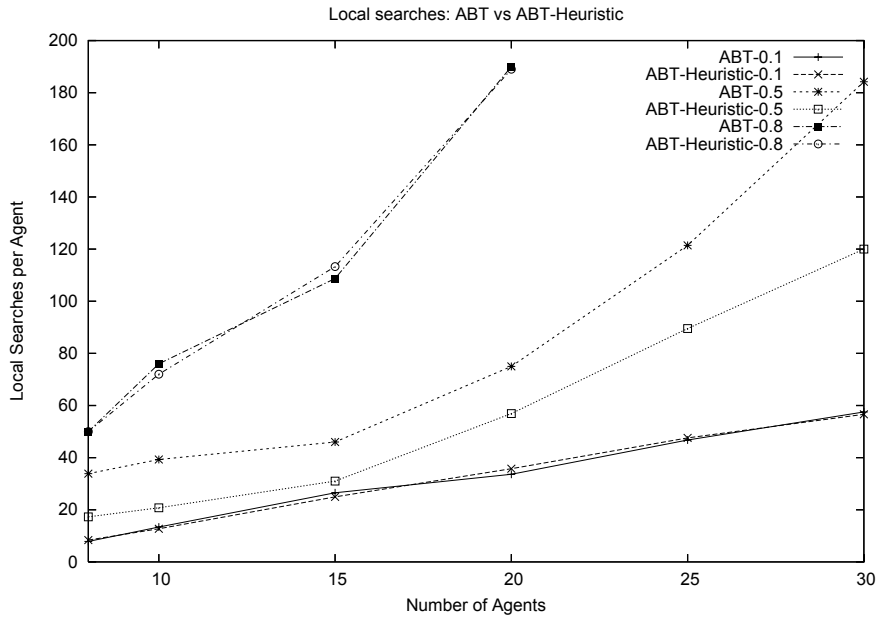


Figure 6.3: Number of local searches: ABT versus ABT using QoS-based Heuristic.

exhaustive searches are likely to occur at each agent and similarly no improvement can be made. A real improvement can only occur for medium difficult problems when there are substantial backtracks which may happen, and hence avoiding less preferred assignments becoming important during the search.

6.5 Summary

This chapter has proposed various mechanisms to *adapt* and *improve* existing distributed constraint satisfaction algorithms – ABT and AAS, so that they can be applied for QoS management in the Web service domain. For adaptation, many assumptions in these algorithms have been addressed so that they are valid for Web service environments. These include the assumptions of existing agent priority ordering, simple local constraints, and reliable communications. For improvement, a monotonic QoS based heuristic has been

proposed. The heuristic can provide better performance in terms of shorter completion time and lower computational overhead for medium-difficult QoS composition problems. As a continuation of these adaptation and improvement proposals, different important aspects of the QoS management, including dynamic environment and quality of solutions, are discussed and addressed in the subsequent chapters of this thesis.

Chapter 7

QoS Management For Dynamic Environments*

7.1 Introduction

In the proposed framework, the algorithms used by the agents play an important role in QoS management. Specifically, they provide a means for the agents to collaborate and make joint decisions on which management actions the agents should take. Chapter 6 has discussed how existing DisCSP algorithms can be adapted and extended into Web service environments in order to be used in the framework. In this chapter, a new algorithm called Dynamic Asynchronous Backtracking (DynABT) is proposed. The algorithm is designed to solve DynDisCSP - the general form of DisCSP in dynamic environments where changes can occur any time during a solving process. In constrast to existing DynDisCSP algorithms such as LD-AWC [MJT+01] and DynAPO [Mai05] which are either limited to solve a sub-class of DynDisCSP or not fully distributed, DynABT can solve general DynDisCSP and is a fully distributed algorithm. DynABT is built on top of ABT and thus can incorporate any ABT adaptation mechanism proposed in the previous chapter.

This chapter is organized as follows. In the following, a brief review on DynDisCSP is presented. This section also introduces important concepts in the context of DynDisCSP

*Preliminary versions of the work presented in this chapter have been published in [NKP06a] and [NKH06].

for QoS management of Web services. Section 7.3 discusses related work on solving DynDisCSP. Section 7.4 presents ABTRestart – a simple extension of ABT that employs a restarting strategy to handle dynamic changes. The proposed DynABT algorithm is described in Section 7.5 and Section 7.6. Section 7.7 compares the performance of DynABT against ABTRestart. Section 7.8 concludes the chapter.

7.2 Preliminaries

A description of DynDisCSP has been presented in Chapter 3 of this thesis. In this description, a DynDisCSP is considered as a DisCSP which is affected by changes over time. A change can be:

- New constraints or variables added into an agent
- Existing constraints and variables removed from an agent
- New agents joining the network
- Existing agents leaving the network

With the above general description, there has been lack of a clear definition for DynDisCSP solutions. A solution of a DynDisCSP is often considered as a solution of the DisCSP, which results from the DynDisCSP *after* some dynamic changes [Mai05; MJT+01; SF02]. This "after" notion implicitly refers to a global clock and the changes can be ordered according to the timing of this clock. However, a perfect synchronization of time across the agents is difficult, if not impossible, to achieve in practice. There is no physical global clock to accurately determine which change happens before another. To tackle this difficulty, the relation between dynamic changes in a DynDisCSP is determined using a logical clock (e.g. [Lam78]) in this thesis. In particular, the changes at the agents are partially ordered according to the causal precedence or the "happened before" relation in the logical Lamport clock [Lam78]. The "happened before" relation, denoted as $\prec$, is defined in the Lamport clock as follows: (i) For two events (e.g. changes) δ_1 and δ_2 on the same process (e.g. agent), and δ_1 comes before δ_2 then $\delta_1 \prec \delta_2$. (ii) If δ_1 is the sending

of a message by one process and δ_2 is the receipt of the same message by another process, then $\delta_1 \prec \delta_2$. (iii) If $\delta_1 \prec \delta_2$ and $\delta_2 \prec \delta_3$ then $\delta_1 \prec \delta_3$. δ_2 and δ_1 are concurrent if neither δ_1 happened before δ_2 nor vice versa.

In this thesis, a DynDisCSP P_{dyn} is viewed as being equivalent to an initial DisCSP P_0 and a set of changes Δ: $P_{dyn} = P_0 \cup \Delta$. These changes are partially ordered according to the "happened before" relation. Note that Δ can be an infinite set. Continuous changes in P_{dyn} and its solutions are defined as follows:

Definition 1 [Continuous Changes]: For a DynDisCSP $P_{dyn} = P_0 \cup \Delta$, a finite subset $\Delta' \subset \Delta$ is called a set of continuous changes (starting from the beginning) in P_{dyn} if for any change $\delta \in \Delta'$ then all changes happenning before this change must be in Δ', e.g: $\forall \delta' \in \Delta : \delta' \prec \delta$ then $\delta' \in \Delta'$.

Definition 2 [DynDisCSP Solution]: An assignment s is a solution of a DynDisCSP $P_{dyn} = P_0 \cup \Delta$ if there exists a set of continuous changes $\Delta' \subset \Delta$ so that s is a solution of the static DisCSP $P_0 \cup \Delta'$.

Note that a solution in DynDisCSP is only temporary, and it can be invalidated by the subsequent changes. Nevertheless, the solution offers an intermediate guideline for the management agents to react against changes. Whilst DynDisCSP solutions discussed in the existing DynDisCSP such as [Mai05; MJT+01; SF02] are not explicitly defined, they are equivalent to the above definition.

7.3 Related works

Over the past few years, there has been an increasing interest in the MAS community in developing different DisCSP algorithms and techniques for static environments. Unfortunately, only a few studies have been carried out on solving DisCSP in dynamic environments (i.e. DynDisCSP) where the DisCSP's description can change over time.

DynDisCSP has lately captured the interest of the AI community. DynDisCSP algorithms are often extensions of existing DisCSP algorithms. Some important DynDisCSP algorithms are:

- The Locally-Dynamic AWC (LD-AWC) algorithm proposed in [MJT+01]. LD-AWC

is extended from Asynchronous Weak Commitment (AWC) [Yok01] to handle dynamic constraints. AWC can be considered as a refinement of ABT in which agents use a min-conflict heuristic (i.e. mimimizing the number of constraint violations) to change their priorities in order to improve the efficency of the search [YH00]. In LD-AWC, an agent executes similar routines as in AWC. However, it appends the assignment of its variables into every *nogood* before sending the *nogood* out. With this appending, the agent's *nogoods* are always valid regardless of dynamic constraint removals or additions. LD-AWC has been proved to be complete provided that changes only occur to agents' local constraints [MJT+01].

- The Dynamic Distributed Breakout (DynDBA) and Dynamic Asynchronous Partial Overlay (DynAPO) proposed in [Mai05]. As the names suggest, those algorithms extend Distributed Breakout (DBA) [YH00] and Asynchronous Partial Overlay (APO) [ML06] respectively. DBA is an iterative improvement algorithm. In DBA, every agent continuously refines a solution by finding a value for its own variable so that only a minimum number of constraints are violated. DBA is an incomplete algorithm. DynDBA, as an extension of DBA, is also incomplete. In DynDBA, whenever a new constraint is added or removed from an agent, the agent performs an update and continues with normal DBA operations [Mai05].

  On the contrary, DynAPO is a complete algorithm. It combines both centralzied and distributed search techniques in order to solve a DynDisCSP. Essentially, agents in DynADPO use centralized search to solve different portions of the DynDisCSP. The agents may expand their problem portions during a solving process if neccessary. In order to do this, some agents are required to share their private constraints with each other. The extension from APO into DynAPO is fairly straight forward. However, a problem in DynAPO can be that the agents can continuously increase the size of the problem portions even when it is unneccessary to do so. An additional message type is introduced in DynAPO to remove links between agents in order to keep the problem portions from increasing their sizes. More information of DynAPO can be found in [Mai05].

- A set of consistency techniques, referred to in this thesis as DisCSP-Enhancements, designed for open environments proposed in [SF02]. These techniques may allow existing DisCSP algorithms to be extended for dynamic environments. They focus mainly on maintaining the consistency between the agents' local solving processes in the presence of dynamic changes. These techniques are discussed very generally and the work lacks a specific discussion on whether termination and completeness for an algorithm are guaranteed when the techniques are used.

The above algorithms and techniques, as discussed, whilst providing many insights into solving a DynDisCSP, do not solve the problem completely. Their limitations are summarized in Table 7.1. As shown in the table, LD-AWC can allow only local constraints. This is not suitable for the QoS management since QoS constraints of composite Web services are shared. Similarly, DynAPO is not fully distributed and may require agents to exchange constraints at some stage of the search. This carries the risk of revealing the agents' organizational privacy. Finally, whilst DisCSP-Enhancement can allow any change to the problem description, there is an uncertainty in whether it can guarantee the completeness of the extended algorithms.

Algorithms	Completeness	Distributed	Allowed Changes
LD-AWC	Yes	Yes	Local Constraints only
DynDBA	No	Yes	Any constraints
DynAPO	Yes	Partially	Any constraints
DisCSP-Enhacement	N/A	Yes	Any constraints

Table 7.1: Existing DisCSP algorithms/mechanisms for dynamic environments.

The next few sections of this chapter will discuss ABTRestart and DynABT. These algorithms are proposed to solve DynDisCSP and do not suffer from the above limitations.

7.4 ABTRestart Algorithm

It has been pointed out previously that a DisCSP algorithm in combination with a restart strategy can be used to solve DynDisCSPs. This section proposes ABTRestart, an algor-

ithm that uses this strategy. In principle, ABTRestart restarts ABT every time a change is detected. Correctness and Completeness of ABTRestart are also proved in this section.

In order to restart an algorithm executed by distributed agents, some synchronization between the agents is required. ABTRestart uses a coordinator agent to control this synchronization. The coordinator agent can be randomly selected among agents. It is responsible for listening to changes reported by any agent and broadcasting the changes to all other agents. After all the agents have updated their local CSPs according to the changes, the coordinator can ask every agent to restart the solving session. Important procedures of ABTRestart are listed in Algorithm 5. The *handleChange* method specifies how an agent should react to a local change. *processChangeUpdateMessage* is used by the agent to handle notifications of remote changes. *processChangeAckMessage* is for the coordinator to process agents' acknowledgements of a notification received. *processRestart* describes the execution at an agent when the agent is asked to restart its local solving process. Finally, the *filter* method is to check the validity of a received message. If an agent receives a message which was sent from some previous solving session, the message should then be discarded.

The ABTRestart algorithm can be described as follows: Upon detection of a new change δ_{new}, an agent records this change in a buffer Δ_{own}, and sends its notification in a *ChangeUpdate* message to the coordinator (in *handleChange*, Lines 1 to 5), the coodinator duplicates the message and broadcasts it to everyone else (Lines 7 and 8). An agent, when receiving a *ChangeUpdate* message, updates a part of the DisCSP according to the message content and sends back a *ChangeAck* message to the coordinator (Lines 9 to 12). After receiving all replies for all recorded changes, the coordinator asks every agent to restart the search in *Restart* messages (Lines 18 to 21), and clear the list of received changes. Note that the coordinator needs to keep track of all agents in the coalition. Every time a new agent joins the network, the agent is obligated to report its presence to the coordinator before any changes from this agent can be propagated.

Algorithm 5: ABTRestart Protocol

```
1  procedure handleChange(δ_new) do
2      update the local CSP according to δ_new;
3      if previous solution is consistent then
4          return ;
5      add δ_new into Δ_own ;
6      create a new ChangeUpdate message mesg with RequestAck field off;
7      send(mesg, coordinator);
8  procedure processChangeUpdateMessage(mesg) do
9      if this is the coordinator and the AckedRequest field in mesg is off then
10         broadcast mesg with AckedRequest field on ;
11     if the change δ_new in mesg is new for this agent then
12         solving_state = false ;
13         update the agent according to δ_new;
14         send( ChangeAck, coordinator);
15 procedure processChangeAckMessage(mesg) do
16     if this agent is not the current coordinator then
17         return;
18     get the change_id contained in mesg;
19     increase the number of agents responding for this change_id;
20     if every recorded change_id is acknowledged by all agents then
21         increase abt_sid ;
22         create a RestartMessage mesg embedded with abt_sid and the list of
           recorded change_id ;
23         broadcast mesg ;
24 procedure processRestartMessage(mesg) do
25     if mesg does not carry all Δ_own then
26         return ;
27     clear agent_view, nogood_list;
28     set the agent's abt_sid to the abt_sid value in mesg ;
29     solving_state = true ;
30     process and clear all messages in message_queue ;
31     send a valid assignment to all lower priority neighbors ;
32 procedure filter(mesg) do
33     get the session identity abt_sid_mesg from mesg ;
34     if abt_sid_mesg > abt_sid then
35         solving_state = false ;
36         add mesg into message_queue ;
37     else if abt_sid_mesg < abt_sid then
38         discard mesg ;
```

In ABTRestart, the coordinator identifies an ABT session with a *abt_sid* number. The number is increased every time a new ABT session is created. It is sent to all other agents through *Restart* messages (Lines 19 and 20). Due to message delay, the values of *abt_sid* at different agents may not be the same. To synchronize ABT sessions, an agent invokes the *filter* procedure on every newly received message. It compares the *abt_sid* number received in the message against its own *abt_sid*. If the received number is lower, the message is simply discarded (Lines 35 and 36). Otherwise, if the received number is higher, the agent knows that its session has lagged behind the message sender's session. The agent then stops its local solving and temporarily stores the received message (Lines 32 to 34). The message is only processed after a *Restart* message with the same *abt_sid* is received from the coordinator.

ABTRestart uses the timeout mechanism described in the previous chapter to handle possible network failures. In particular, if the coordinator is not responding then the highest priority agent among the remainder agents can be selected as the new coordinator. It is also worth noting that the content of a *ChangeUpdate* message may contain the change details. If the change is local to an agent then the change details can be omitted. Otherwise, those details are encoded so that only agents which are affected by the change can understand them so as to avoid a privacy leak. In the QoS management problem, change details are important when a change request initiated by an agent is related to some composition. There may be different ways to encode the details. A possible approach is to have agents in the same composition share a secret key to encrypt those details. More detailed information of ABTRestart is available in Appendix B of this thesis.

7.5 ABTRestart Correctness and Completeness

The following proposition confirms the correctness of solutions reported by the ABTRestart algorithm which uses the solution detection mechanism described in the Section 6.3.5 of the previous chapter.

Proposition 7.1 [ABTRestart Correctness]: Every solution reported by ABTRestart

using the solution detection mechanism described in Section 6.3.5 is correct.

Proof: For any reported solution S, the proposition can be proved if we can show that there exists a set of continous changes $\Delta' \subset \Delta$ so that S is a solution of $P_0 \bigcup \Delta'$. ABTRestart uses the solution detection mechanism described in Chapter 3. The mechanism works by having agents send upstream (i.e. to their higher priority neighbors) *good* messages. A *good* message sent by an agent A_k to another agent A_i contains a good assignment of A_i and a valid partial solution for all agents in the descendent trees rooted at A_k. The anouncement of finding S at the highest priority agent A_{ini} implies that A_{ini} was in the same ABT session with all of its neighbors in $\mathbf{A}_{t\_neighbors}$ when S was created; otherwise some of the *good* messages sent by $\mathbf{A}_{t\_neighbors}$ would have been dropped by the *filter* procedure of the ABTRestart algorithm, and hence S would not be formed. These arguments can be repeated for the partial solution reported at each $A_m \in \mathbf{A}_{t\_neighbors}$. A_m must be in the same session with its neighbors when the partial solution is created, and so on. Hence we can conclude that there was an ABT session $\wp$ where all assignments at every agent were consistent. Δ' can be taken as the set of changes in the DisCS Restart messages at the begining of $\wp$. We only need to prove that Δ' is a set of continuous changes.

By contradiction, suppose that $\exists \delta_i \in \Delta, \delta'_i \in \Delta'$ so that $\delta_i \prec \delta'_i$. Let us denote e_r as the event that the coordinator sent out Restart messages to begin $\wp$, A_m as the agent in which δ'_i occurs, and e_{A_m} as the event A_m restarts its search, we then have $\delta'_i \prec e_r$, $e_r \prec e_{A_m}$. Since the "happened before" relation is transitive then $\delta'_i \prec e_{A_m}$. This indicates that A_m has its change δ'_i before receiving a Restart message. Because this message carries Δ' which does not contain δ'_i, A_m should not have restarted its local search according to Lines 23 and 24 of Algorithm 5. Hence it is contradictory. In other words, the above proposition holds. ■

Proposition 7.2 [ABTRestart Completeness]: ABTRestart is complete if to every agent in a DynDisCSP, the changes in this problem occur one after another, and the time distance between two successive changes is long enough.

Proof: By assuming that to every agent the changes to the DynDisCSP occur one after another, we mean that when a change happens to some part of the coalition, all agents

are notified about this before a new change can appear. With this assumption and the assumption that the time distance between two successive changes is long enough, the DynDisCSP P_{dyn} can be considered as a series of static DisCSPs $P_{dyn} = \{P_i\}$ where an ABT execution for every single DisCSP P_i can terminate. Since ABT is a complete algorithm, if a solution of P_i exists, it can always be found. Therefore the proposition holds. ■

7.6 Dynamic ABT Algorithm

ABTRestart has the advantage of being simple. However, since the search is restarted for every new change, whatever information was previously achieved, is lost. It is more desirable to have this previous information reused in the new ABT session and hence less effort is required during subsequent searches. DynABT (Dynamic Aysnchronous Backtrackig Search) is designed for this. DynABT is a complete algorithm. It uses *nogood explanation* [VS94a] to reuse partial search results and employs a totally *reactive* strategy to handle dynamic changes.

The next section begins with an introduction to the concept of *nogood explanation* and its use in DynCSP. An efficient scheme for building *nogood explanations* in DynABT is then proposed. Finally, DynABT procedures are presented and disscussed.

7.6.1 Nogood Explanations

Nogood explanation is an important concept used in DynABT. It was first introduced in [VS94a] for solving dynamic centralized CSP. Generally, a *nogood explanation* is a constraint that justifies the reason why an assignment is invalid. Formally,f it is defined as follows:

Definition 3 [Nogood Explanation]: For *nogood* $n{=}\neg\{(x_1 = d_1) \bigwedge \ldots \bigwedge (x_k = d_k)\}$, a constraint ℓ is called an explanation of n iff ℓ can induce n, i.e. $\ell \longrightarrow n$.

Two immediate properties following from this definition are:

Property 1: If ℓ_1 is an explanation of a nogood n, ℓ_2 is a constraint and $\ell_2 \longrightarrow \ell_1$ then ℓ_2 is also an explanation of n. In this thesis, we say that ℓ_1 is more *optimal* than ℓ_2.

Property 2: If all ℓ_i, $1 \leq i \leq k$, are explanations of a nogood n then $\ell = \ell_1 \bigvee \ldots \bigvee \ell_k$ is also an explanation of n. Also, ℓ is more optimal than ℓ_i.

For a DynDisCSP, a constraint can be removed from, or added into, the problem. An agent A_k considers a constraint as being *valid* at a particular time locally at A_k if to A_k's knowledge, the constraint still belongs to the problem at that time. Otherwise, it is *invalid* to A_k. Consequently, validity of a *nogood explanation* can be defined as follows:

Definition [Validity of nogood explanation]: For an agent at a particular local time, a *nogood explanation* ℓ is valid if there exists a set $\mathbf{C}$ of all valid constraints so that $\bigwedge_{c_t \in \mathbf{C}_t} c_t \longrightarrow \ell$. A *nogood* is valid if any of its explanations is valid. Otherwise, the nogood is invalid.

In other words, an *explanation* of a *nogood* n is valid if it can be induced by valid constraints in the DynDisCSP. Consequently, it should be kept for further processing. Otherwise, if an *explanation* is invalid, the associated *nogood* should be removed since the constraints which caused the *nogood* have been removed.

Let us consider the example in Figure 7.1 of a map coloring problem which has the constraints that only two colors (WHITE=1, RED=2) must be used to color three different neighboring countries and any two countries are colored differently. This problem can be translated to a set of original constraints $c_0 : x_i \in \{1, 2\} \forall i$ =1 to 3, $c_1 : x_1 \neq x_2$, $c_2 : x_2 \neq x_3$, $c_3 : x_1 \neq x_3$ where x_i is the color of the *i-th* country. For this problem, the assignment $(x_1 = 1, x_2 = 2)$ cannot satisfy all c_0, c_2, and c_3 since no value for x_3 can be found. Therefore $\neg(x_1 = 1, x_2 = 2)$ is a *nogood* and $\ell = c_0 \bigwedge c_2 \bigwedge c_3$ is its explanation. In addition, $c_0 \bigwedge c_1 \bigwedge c_2 \bigwedge c_3$, whilst being less optimal than ℓ, is also an explanation for the *nogood*. In addition, if c_0, c_2, or c_3 is removed, the *nogood* may become invalid.

For DynDisCSP, constraint removals are in general more difficult to handle than constraint additions. For example, to handle new constraints, any existing assignments that invalidate the new constraints can simply be removed. Handling removals of constraints, however, is a more complex task. Any solutions which were invalidated by the constraints in the past should be recovered. *Nogood explanations* are especially useful for this task.

The basic idea of using *nogood explanations* is simple and similar in different DynCSP

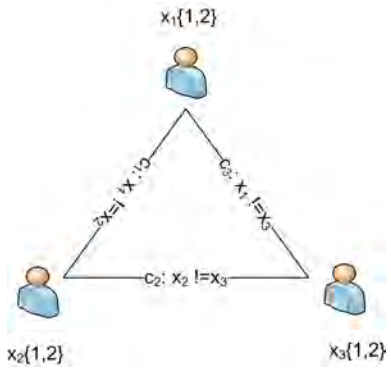


Figure 7.1: A map coloring problem for 3 neighboring countries and 2 colors. If $c_2 \bigwedge c_3 \bigwedge (x_1 = 1) \bigwedge (x_2 = 2)$ *holds then* x_3 *cannot have the value of 1 (to satisfy* c_3*) or 2 (to satisfy* c_2*) hence* $c_0 \bigwedge c_2 \bigwedge c_3 \longrightarrow \neg\{(x_1 = 1) \bigwedge (x_2 = 2)\}$*. In other words,* $\ell = c_0 \bigwedge c_2 \bigwedge c_3$ *is an explanation of the nogood* $\neg(x_1 = 1, x_2 = 2)$*.*

algorithms (e.g. [VS94a; VS94b; VJ05]). It can be explained as follows: In those algorithms, *nogood* are recorded in a *nogood* list to avoid being re-generated for assignments. A *nogood* is recorded together with its explanation. An explanation for a *nogood* n is a set of constraints[1] that conflicts with the assignment of n. Whenever a constraint in the set is removed due to dynamic changes, the *nogood* is also deleted from the recorded *nogood* list. It can be seen that with this scheme, the recorded *nogoods* are always consistent with existing constraints in the problem at any time. In other words, the *nogoods* never invalidate any potential solutions.

7.6.2 DynABT and nogood explanations

The previous section has introduced the concept of *nogood explanation* and its applications for DynCSP. This section further discusses how the concept can be used to solve a DynDisCSP. The main difficulty when using *nogood explanations* in a distributed environment is in determining the validity of a *nogood explanation*. A *nogood explanation* in a DisCSP algorithm is often constructed using both *original* constraints (i.e. constraints defined in the DynDisCSP's description) and *induced* constraints (i.e. the recorded *nogoods* in an

[1] In reference to the *nogood explanation* definition in the previous section, the actually explanation in this case is the conjunction of all constraints in the set.

agent). Due to the presence of the induced constraints, it is difficult to determine the validity of the explanation. In ABT, an agent cannot determine the validity of an induced constraint since this constraint (e.g. *nogood*) is created by one of the lower priority neighbors of the agent.

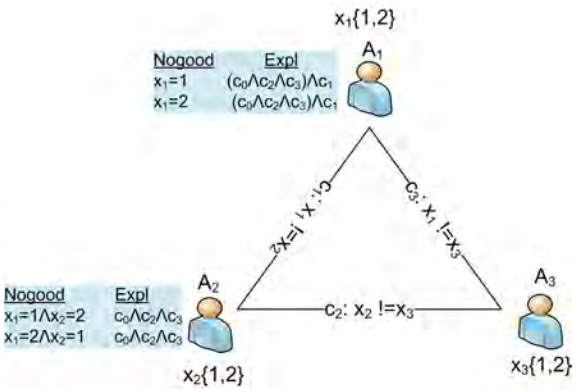


Figure 7.2: Suppose that c_1's removal is notified to A_1 and A_2. A_1 will remove its nogoods since the explanation $c_0 \bigwedge c_2 \bigwedge c_3 \bigwedge c_1$ becomes invalid. A_2 will keep all of its nogoods since their explanations remain valid.

DynABT addresses the above difficulty by allowing agents to exchange the explanations of induced constraints. DynABT uses a similar backtracking mechanism as in ABT. However, in DynABT when a *nogood* message is sent, an explanation for the *nogood* (i.e. induced constraint) is also embedded in the message. The receiver agent records both the *nogood* and its explanation. Later, when the agent needs to use this *nogood* to create some other explanation, it uses the *nogood*'s explanation instead. By doing so, eventually every explanation can be constructed from original constraints only. The validity of an explanation then can be easily determined based on the validities of the original constraints.

Following the above steps, this section proposes two different schemes – the *optimal explanation building scheme* and the *possibilistic based explanation building scheme* – which can be used by agents in DynABT to create *nogood explanations*. Whilst the first scheme can create an optimal explanation for a *nogood*, it comes with the cost of high computational complexities. The second scheme creates a less optimal explanation, however, it requires only a linear space complexity.

The optimal explanation building scheme

This scheme focuses on improving the optimality of the returned *nogood explanation*. For two explanations ℓ_1 and ℓ_2, if ℓ_1 is more optimal than ℓ_2 then ℓ_1 is less likely to be invalid than ℓ_2. This is because whenever ℓ_2 is valid, ℓ_1 is also valid. Consequently, the more optimal the explanation that a *nogood* has, the lower the chance is for the *nogood* to be removed. The scheme uses this principle to reduce the removal chance of a *nogood*. It is presented in Algorithm 6. When a *nogood* is detected, an agent A_k attempts to find a list of explanations ℓ for this *nogood*. Also, each explanation in ℓ is a conjunction of some constraints and *nogoods* that are held by A_k (Line 2). The agent then creates a new explanation by taking the disjunction of those explanations (Line 3) and replaces any *nogood* with its corresponding explanation (Line 4). By taking the disjunction of existing explanations, the newly returned explanation is more optimal than any existing one according to Property 2 in the previous section.

Algorithm 6: Explanation Building for a *nogood* at an agent A_k

1 **procedure** *create_opt_expl(n)* **do**
2 Find a list of explanations $\boldsymbol{l} = \{\ell : \exists \ell_{\mathbf{c}}$ a set of constraints and nogoods of A_k so that $\ell = \bigwedge_{c \in \ell_{\mathbf{c}}} c\}$ for the nogood n ;
3 Create a new explanation $\ell_{temp} = \bigvee_{\ell \in \boldsymbol{l}} \ell$;
4 Construct ℓ_{final} by replacing all nogoods in ℓ_{temp} with their explanations ;
5 Return ℓ_{final};

The following proposition confirms the correctness of the explanation building scheme:

Proposition 7.3 [Correctness of explanation building]: The *optimal explanation building scheme* in Algorithm 6 returns an explanation for the input *nogood* n. This explanation consists of only original constraints.

Proof: According to Algorithm 6, any *nogood* in ℓ_{temp} is replaced by its explanation. If this explanation contains a *nogood* then the *nogood* is again replaced by its explanation. This replacement occurs recursively until no *nogood* remains in ℓ_{temp}. An infinite loop cannot happen since an explanation of a *nogood* recorded at A_k can only contain *nogoods* recorded at A_k's decendants and hence no cyclic references can take place. The algorithm is thus terminated after a finite period of time. At Line 3 of Algorithm 6, ℓ_{temp} is an explanation of *nogood* according to Property 2. Line 4 replaces *nogoods* by their explanations,

hence $\ell_{final} \longrightarrow \ell_{temp}$. ℓ_{final} is thus an explanation of n. ■

Since *nogood explanations* in DynABT contain only original constraints, they can be applied in the same way as they have been used in DynCSP algorithms. In particular, whenever an explanation becomes invalid, its associated *nogood* is removed. Figure 7.2 illustrates this for the same example of map coloring with three agents, presented in the previous section. Suppose that some *nogoods* have been partially constructed as shown in the figure. If c_3 is removed for some reason then all *nogoods* held at A_1 and A_2 will automatically be removed since the explanations of these nogoods become invalid. However, if c_1 is removed, only nogoods held at A_1 will be removed.

With the *optimal explanation building scheme*, a DynABT nogood explanation is a boolean formula using disjunctions and conjunctions only. Finding all "conjunction-form" explanations in Line 2, whilst making the final explanation more optimal, can result in high time complexity. Therefore, to avoid the complexity, only a limitted number of explanations[2] should be found in Line 2 of Algorithm 6. One way to do this is to find the sets $\ell_{\mathbf{c}}$ mutually nonoverlapped. Basically, a set of $\ell_{\mathbf{c}}$ is found first. After that, all elements in this $\ell_{\mathbf{c}}$ are removed from the agent's constraints and the next $\ell_{\mathbf{c}}$ are created from the set of remaining constraints. This process is repeated until no such $\ell_{\mathbf{c}}$ can be found. Algorithms for finding such an $\ell_{\mathbf{c}}$ can be found in [LMD06]. Whereas the time complexity can be adjusted for the scheme, its space complexity cannot. The space complexity refers to the length of the explanation formula returned from Algorithm 6. For ABT, a *nogood explanation* at an agent A_k is composed from its own *original* constraints and those of its decendants. Apparently, the explanation length (i.e. the number of original constraints in the formula) grows exponentially in the worst case for the highest priority agent. To address this problem, the *possibilistic based explanation building* scheme is proposed next. It makes use of possibilities of change events estimated by agents. Only a part of an explantion that has the lowest necessity to be removed is kept in the scheme.

The possibilistic based explanation building scheme

[2]In the worst case, only one explanation is found. This results in no extra complexity introduced by DynABT into ABT.

This scheme uses the possibility/neccessity concepts from possibility theory [Zad75] to measure the uncertainties in future validities of constraints. For each constraint c, the event that c is removed from the constraint network, is denoted as e_c. Each e_c is associated with a pair of possibility and neccessity measures $\pi(e_c)$ and $n(e_c) = 1 - \pi(\neg e_c)$. In particular, $\pi(e_c)$ is the degree of possibility and $n(e_c)$ is the degree of certainty that e_c will happen. We assume that $\pi(e_c)$ and $n(e_c)$ are estimated by agents who own c_k. The assumption is reasonable as the agents possess certain domain knowledge over their own contraints and have evidence regarding the plausibility of e_c's appearance. Such evidence can be the agents' knowledge of c_i's lifetime. For example, a constraint with a higher estimated lifetime has a lower possibility to be removed during the execution of a DynABT session.

With the introduction of possibility/necessity measures into every constraint removal event, an agent can compute the possibility and neccessity that a *nogood explanation* remains valid. In the *possibilistic based explanation building* scheme, an explanation is a conjunction of original constraints. This scheme is similar to the *optimal explanation building* scheme in that a list of explanations needs to be found (Line 2 of Algorithm 6 and 7). However, instead of taking the disjunction of these explanations, their neccessity measures are computed (line 4 - Algorithm 7) and compared against each other. The explanation with the lowest measure is selected as the final explanation. Note that the computation is based on the property that $n(\mathbf{U} \bigcap \mathbf{V}) = min(n(\mathbf{U}), n(\mathbf{V}))$ for any disjoint subsets $\mathbf{U}$ and $\mathbf{V}$ of events.

Algorithm 7: Possibilistic based Explanation Building for a nogood

1 **procedure** *create_poss_based_expl(n)* **do**
2 Find a list of explanations $\boldsymbol{l} = \{\ell : \exists \ell_{\mathbf{c}}$ a set of constraints so that $\ell = \bigwedge_{c \in \ell_{\mathbf{c}}} c\}$ for the nogood n ;
3 **foreach** $\ell \in \boldsymbol{l}$ **do**
`/* denote` e_ℓ `as the event that` ℓ `becomes invalid */`
4 compute $n(e_\ell) = min\{n(e_c) : c \in \ell_{\mathbf{c}}\}$;
5 Select $\ell_{nec-max} \in \boldsymbol{l}$, $n(e_{\ell_{nec-max}}) = min\{n(e_\ell) : \ell \in \boldsymbol{l}\}$;
6 Construct ℓ_{final} by replacing all nogoods in $\ell_{nec-max}$ with their explanations ;
7 Return ℓ_{final} ;

Since Algorithm 7 uses the same method as in Algorithm 6 for aggregating explanations, it always returns an explanation for the input *nogood*. Hence Algorithm 7 is correct.

In addition, this algorithm does not suffer from exploding of explanation lengths as confirmed in the following proposition:

Proposition 7.4 [Space complexity]: In the worst case, the length (i.e. number of attributed constraints) of a valid explanation in Algorithm 7 is equal to the total number of valid and original constraints from all agents.

Proof: We note that explanations built with Algorithm 7 use only the conjunction (i.e. $\bigwedge$) operator and hence duplicated constraints in an explanation can always be removed. Also an invalid constraint will invalidate the whole explanation. Therefore, any explanation can only contain valid constraints and the number of these constraints cannot be higher than the total available constraints from agents. If there is an upper limit on number of constraints per agent, then the complexity of explanation length using Algorithm 7 is $\boldsymbol{O}$(n) where n is the number of agents. ■

Figure 7.3 presents a scenario that demonstrates the advantage of using Algorithm 7. In this figure, a number of explanations at agents A_1, A_2, A_3 were constructed using explanations at agents A_4 and A_5. Also, the explanations at A_4 and A_5 were reported by A_6. A_6 can use either c_{61} or c_{62} in the boolean formulas for its explanations. In other words, the roles of these constraints are interchangeable in the formulas, e.g. both $c_{61} \bigwedge c_{41}$ and $c_{62} \bigwedge c_{41}$ are explanations of n_{41}. A_6 selects a constraint beween c_{61} and c_{62}, whichever has a lower neccessity level to be removed. This is to increase the certainty that all shown *nogoods* will be kept in the future.

7.6.3 Intra-Agent Consistency

This section describes how changes can be handled at each agent in DynABT. Obviously, in order to execute an algorithm correctly, an agent needs to ensure that its local solving state is consistent after every change. This consistency is referred to as *intra-agent consistency*. In DynABT, the consistency is between the agent's local constraints, assignments, *agent view* and recorded *nogoods*.

DynABT defines six different types of *primary change*. Any change can be considered

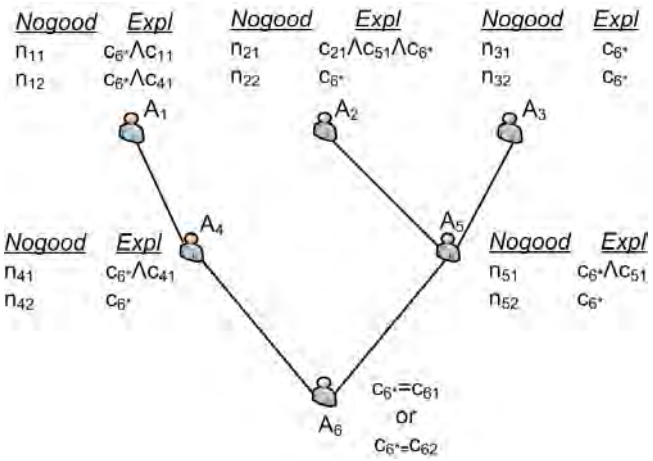


Figure 7.3: An example of agents using the possibilistic based explanation building scheme.

in general as a collection of these *primary changes*. A *primary change* can be one of the following:

- *NeighborAddition*: New neighbors are added for an agent.
- *NeighborRemoval*: Existing neighbors of an agent are removed.
- *ConstraintAddition*: New constraints are added into an agent.
- *ConstraintRemoval*: Existing constraints are removed from an agent.
- *VarAddition*: New variables are added to an agent.
- *VarRemoval*: Existing variables are removed from an agent.

Note that changes of the last two types – *VarAddition* and *VarRemoval* can be constructed from *ConstraintAddition* and *ConstraintRemoval* changes. Nevertheless, *VarAddition* and *VarRemoval* are presented as types of *primary change* for the sake of clarity and simplicity. Also, *NeighborAddition* and *NeighborRemoval* changes only add or remove neighbors from an agent. They neither add nor remove any shared variables or constraints between the neighbors and the agent.

For example, a new Web service composition formed among a set of agents $\mathbf{A}_c$ is equivalent to the following set of *primary changes*:

- *NeighborAddition* – changes at every agent in the composition which does not know some other agents. In other words, if $A_i \in \mathbf{A}_c$ and A_i does not know $A_{j1}, \ldots, A_{jk} \in \mathbf{A}_c$ then A_i adds $A_{j1}, \ldots, A_{jk}$ as new neighbors.

- *VariableAddition* – changes at agents which are affected by the *NeighborAddition* changes. For the agents in the discussion above, A_i adds new variables (into its list of known variables) which are the variables controlled by $A_{j1}, \ldots, A_{jk}$.

- *ConstraintAddition* changes at every agent in the composition. Each agent adds new constraints representing *end-to-end* QoS requirements for the composition.

Algorithm 8: DynABT Protocol - Change Management for Local Consistency

```
1  procedure consumeChange(Δ_new) do
2      Δ_{A_k} = Δ_new ⋃ Δ_{A_k} ;
3      affected = false ;
4      foreach δ ∈ Δ_new do
5          if δ is not for A_k then
6              continue ;
7          update A_k' local CSP according to δ ;
8          if δ is VarAddition then
9              get the set of controlled variables vars in δ;
10             foreach v in vars do
11                 current_assignmt = current_assignmt ⋀(v=min(D_v));
12         else if δ is VarRemoval then
13             get the set of variables vars in δ ;
14             r_constraints = {c in constraints: var(c)∩vars != ∅};
15             consumeChange(ConstraintRemoval[r_constraints]);
16             delete all assignments of vars in current_assignment;
17             delete all nogoods containing any of vars in the nogood storage;
18         else if δ is ConstraintRemoval then
19             get the set of constraints r_constraints in δ;
20             foreach nogood in A_k's nogood storage do
21                 if nogood is invalid after removing r_constraints then
22                     remove nogood;
23         if δ is only for A_k then
24             empty the content of δ;
25         affected = true ;
26     return affected ;
```

Algorithm 8 illustrates the procedure (i.e *consumeChange*) that an agent A_k executes to maintain its intra-agent consistency in the presence of a new change. The change is represented in the algorithm as a set Δ_{new} of primary changes. For each primary change δ in Δ_{new}, the agent first updates its local CSP according to the description of the change (Line 7). If the change is one of *VarAddition*, *VarRemoval*, or *ConstraintRemoval* then further processing needs to be carried out as follows:

- If δ is a *VarAddition*, A_k updates its set of controlled variables, and expands its current assignment for local variables if necessary (Lines 9 to 11).
- If δ is a *VarRemoval*, A_k finds the set of contraints that are subject to removal as a consequence of δ (Line 14). Those constraints are removed by recursively calling the *consumeChange* procedure (Line 15). In addition, A_k's current assignment is shortened and invalid *nogoods* are deleted (Line 16 and 17). Validity of nogoods have been discussed in Section 7.3 of this chapter.
- If δ is a *VarRemoval*, A_k removes all invalid *nogoods* in its *nogood* list. The validity of a *nogood* is determined by its explanation as detailed in Section 7.5.1.

With the above steps, the local solving inside an agent can always be kept consistent after the change. It is worth noting that the procedure *consumeChange* returns a flag to indicate whether A_k's local state is affected. If a *true* value is returned then a new assignment for A_k's variables may need to be generated again at some later stage of DynABT.

7.6.4 Inter-Agent Consistency

So far, we have discussed the mechanism to achieve local solving consistency internally at every agent but not between agents. On the one hande, in a distributed environment, two agents can be notified about and hence are made aware of the same change but at different times due to transmission delay. On the other hand, consistency between the agents in viewing the problem is important in order for the agents to reach a final solution. Agents which know the same set of changes have consistent views of the problem. This

consistency is referred to as inter-agent consistency in this thesis. Inter-agent consistency is about synchronizations of changes noted by different agents.

Algorithm 9: DynABT Protocol

```
1  procedure dyn_abt(mesg) do
2      A_m = the sender of mesg ;
3      Δ_mesg = the set of changes embedded in mesg;
4      if neighborSyn(Δ_mesg, A_m ) then
5          if mesg is an ABT message then
6              abt(mesg);
7  procedure neighborSyn(Δ_mesg, aid) do
8      Δ_new = Δ_mesg \ Δ_A_k ;
9      Δ_for−neighbors = Δ_A_k \ Δ_mesg ;
10     if Δ_for−neighbors ≠ ∅ then
11         updateNeighbor(aid) ;
12         return false;
13     if Δ_new ≠ ∅ then
14         updateMyself(Δ_new) ;
15         clear partial_sol ;
16     return true;
17 procedure updateNeighbor(aid) do
18     if AID has higher priority than this agent then
19         create an update message mesg ;
20         send(mesg, aid);
21 procedure updateMyself(Δ_new) do
22     if Δ_new = ∅ then
23         return;
24     affected = consumeChange(Δ_new);
25     create an update message mesg;
26     send(mesg, lowestPriHigherNeighbor) ;          /* notify upstream */
27     if local_view is consistent then
28         foreach lower priority neighbor nb do
29             send(ok?, nb) ;                        /* notify downstream */
30     else
31         if affected then
32             check_agent_view ;    /* skip check local consistency inside */
```

In contrast with ABTRestart, which uses a coordinator to facilitate the synchronization, DynABT employs a fully distributed mechanism to propagate changes. In particular,

each agent A_k keeps a list of changes Δ_{A_k} that A_k is aware of. Every message sent by A_k is embedded with this Δ_{A_k}. The message receiver compares Δ_{A_k} against the changes it knows, and attempts to synchronize itself with A_k. DynABT essentially adds a new synchronization procedure, i.e. the *neighborSyn* method, on top of ABT, as shown in Algorithm 9. This procedure is called on every time A_k receives a message. Its task is to synchronize A_k with the message sender. If such a synchronization succeeds, *neighborSyn* returns *true* and ABT is called on to process the received message (Lines 4 to 6 of Algorithm 9). The synchronization may fail if the sender is not as updated as A_k (i.e. A_k knows more changes than the sender). In this case, the sender should attempt to synchronize itself with A_k instead. A_k triggers this by sending an *update* message to the sender (Lines 18 to 21). Since the *update* message contains the set of changes known by A_k, the sender can synchronize itself with A_k through this set of changes.

The *neighborSyn* method can be described in detail as follows: whenever an agent A_k receives a DynABT message $mesg$, the embeded set of changes Δ_{mesg} in the message is extracted and compared with Δ_{A_k} – the set of changes noted by A_k (Lines 8 and 9). If new changes are detected in Δ_{mesg}, those changes will be processed by A_k using the method *updateMyself* (Line 11). This method in turn invokes the *consumeChange* method discussed previously (Line 24) in order to maintain the consistency of A_k's local solving.

A change may happen locally in a remote part of the agent network. Therefore, upon receiving new changes, an agent should propagate the changes further. It does this by sending an *update* message embedded with the changes to the highest priority neighbor. At the same time, the agent resends *ok?* messages to every lower priority neighbor. These steps ensure that the new changes will eventually be propagated to all agents in the coalition. We note that sometimes a new change may not affect an agent's local solving status (the *affected* variable is *false* at line 25). In this case, reinvoking *check_agent_view* method is unnecessary (line 32).

In algorithm 9, DynABT is shown to use the *check_agent_view* procedure from ABT. However, it is important to note that there is a small modification to this procedure (which is not listed in Algorithm 9). Specifically, as much as possible every agent avoids

generating an assignment resembling previous *instantaneous solutions*. A simple way to achieve this is to follow a specific path in the agent's variable domains when searching for new assignments. The QoS heuristic described in Chapter 6 can be used to identify the path. In particular, the agent attempts to find assignments with lower preference than the last *solution*. If this cannot be done, the agent then starts again from the highest preference assignment.

7.6.5 Addition and removal of agents

When an agent receives an event of addition or removal of other agents, the agent uses Algorithm 8, presented previously, to maintain the consistency of its local CSP. The events are messages generated by the joining or leaving agents. When an agent joins the coalition, it sends *join* messages to its future neighbors. A *join* message carries shared constraints and variables between the joining agent and a neighbor.

The agents leaving event is more difficult to manage, especially if the departures are due to agent functional failures (e.g. unexpected shutdowns). In the normal case of no functional failures, an agent sends *leave* messages to its neighbors to indicate its withdrawals. Upon receiving a *leave* message, a neighbor agent executes the procedure in Algorithm 8 to maintain its local consistency. If there is a functional failure of an agent, it may prevent the *leave* messages from being delivered. In such a case, message *timeout*, discussed in Chapter 6, is used to detect those failures. If the highest priority agent is leaving the network, it should notify every other agent about the new highest priority agent. Note that in the proposed QoS management framework, these agents are neighboring. When an agent removal is notified, all other agents revise their neighbor links and attempt to connect to the highest priority agent, if the agents do not have any higher priority neighbors. In this way, DynABT always maintains a connected network of agents through the highest priority agent. Whilst DynABT always maintains the coalition, it can also split the coalition into separated smaller coalitions and hence can achieve better efficiency in solving provided some conditions are met. These conditions are stated as follows:

Proposition 7.5 [Coalition splitting conditions]: If A_k – the highest priority agent

receives good messages from all neighbors $\mathbf{A}_k^n$ and:

- $\mathbf{A}_k^n$ can be divided into two non-overlapping subsets $\mathbf{A}_k^n = \mathbf{A}_k^{n1} \bigcup \mathbf{A}_k^{n2}, \mathbf{A}_k^{n1} \bigcap \mathbf{A}_k^{n2} = \emptyset$ so that the *good* messages received from $\mathbf{A}_k^{n1}$ contain no common agent names [3] with good messages received from $\mathbf{A}_k^{n2}$
- A_k has no shared constraints with $\mathbf{A}_k^{n2}$

A_k can then disconnect itself (using *leave* messages) from $\mathbf{A}_k^{n2}$. The original DynABT session will be divided into two separated smaller DynABT sessions running at two separated sub-coalitions.

Proof: We consider the two sub-coalitions $\mathbf{N}_1$ and $\mathbf{N}_2$. $\mathbf{N}_i$ are formed by agents, excluding A_k, which can reach any agent in $\mathbf{A}_k^{ni}$: $i = 1, 2$. By contradiction, suppose that $\mathbf{N}_1$ and $\mathbf{N}_2$ have a common agent A_j then there are some agents in $\mathbf{N}_1$ and $\mathbf{N}_2$ which are ancestors of A_j or A_j itself. Since a good message reported by an agent carries the local solution of every descendant of the agent, A_j's solution must be included and reported by agents in both $\mathbf{N}_1$ and $\mathbf{N}_2$. This is contradictory to the first condition. Hence, $\mathbf{N}_1$ and $\mathbf{N}_2$ have no shared agents except A_k. Consequently, $\{A_k\} \bigcup \mathbf{N}_1$ and $\mathbf{N}_2$ can be solved separately as different problems since A_k has no constraints shared with $\mathbf{N}_2$. ■

7.7 Correctness and Completeness of DynABT

This section proves the correctness and completeness of the proposed DynABT algorithm.

Proposition 7.6 [DynABT Correctness]: every solution reported by ABTRestart using the solution detection mechanism described in Section 6.3.5 is correct.

Proof: When a solution is detected at the highest priority agent A_h, every *good* message reported to A_h by A_h's neighbors in $\mathbf{A}_{h_{nb}}$ must carry the same set of changes with A_h. Otherwise, the solution kept in A_h should have been cleared as shown in the *neighborSyn* procedure in Algorithm 9. The argument can be repeated for *good* messages reported by neighbors of every agent in $\mathbf{A}_{h_{nb}}$ and so on. Hence, we can conclude that at some point during the solving process, every agent sends a *good* message which is embedded with the

[3] Recall that a good message contain a list of agents with their corresponding local solutions.

same set of changes. Let us denote this set as Δ', and the set of changes in the DynDisCSP as Δ. Obviously, the solution satisfies the local constraints of every agent once all changes in Δ' are enforced. The remaining part is to prove that the changes in Δ' are continuous changes of Δ.

By contradiction, assume that there exist two changes δ and δ' so that $\delta \in \Delta$, $\delta' \in \Delta'$, $\delta \prec \delta'$, and $\delta \notin \Delta'$. Let us denote A_k and $A_{k'}$ as the two agents where δ' and δ took place respectively, e_r as the event when $A_{k'}$ first knew about δ. A_k and $A_{k'}$ cannot be the same since an agent includes the whole set of its changes in every sent message. Two continuous changes generated by the agent should always be in the set. Since A_k and $A_{k'}$ are different agents, $A_{k'}$ only knows about δ through some exchanged DynABT message. Hence e_r is the event that $A_{k'}$ receives the message. This message must contain δ, and $A_{k'}$ should update its list of noted changes $\Delta_{A_{k'}}$ to include δ. Because $e_r \prec \delta'$, $\Delta_{A_{k'}}$ must either contain δ alone or δ and δ' together. It is a contradiction that $\Delta_{A_{k'}}$ becomes Δ' at some point during the search, since $\delta' \in \Delta'$ and $\delta \notin \Delta'$. Hence, the proposition holds. ■

Proposition 7.7 [DynABT Completeness]: DynABT is complete if to every agent in a DynDisCSP, the changes to this problem can be considered as one after another, and the time distance between two successive changes is long enough.

Proof: Similar to the proof for the completeness of ABTRestart, with the above assumptions, the DynDisCSP $P_{dyn} = P_0 \cup \Delta$ can be considered as a series of DisCSP $\{P_i\}$ from the perspective of every agent. If the time distance between P_i and P_{i+1} is long enough, eventually all changes in P_i are propagated to and known by every agent. Consequently, ABT (Line 6 in Algorithm 8) is executed. With the use of *nogood explanations*, the local CSP solving processes in all agents are always consistent after the changes from P_i. Therefore, the agents can be considered as using the ABT algorithm to solve P_i. The ABT solving is, however, not from scratch as in ABTRestart. Since ABT is complete, a solution of P_i can be found if one exists. Thus the proposition holds. ■

7.8 Experiments

An experiment has been carried out using our MAS-SIM toolkit [IAM06] to compare ABTRestart against DynABT. In the experiment, dynamic changes are simulated as random events which can be generated and scheduled before the experiments start up. This enables the same DynDisCSP to be tested with both ABTRestart and DynABT. The experiment focuses on the effect of composition additions/removals on the algorithms' performance. The experiment setup is similar to the one in Chapter 6. At the begining, a number of compositions are formed randomly among agents so that one agent participates in 3 or 4 compositions. Each composition consists of 3 agents. During the experiment, compositions were randomly formed or removed among 3 agents. The interesting QoS parameters are *availability, response time* and *cost*. Each parameter has a domain of 10 discrete values. A composition has end-to-end QoS constraints with tightnesses of 50% on the parameters. The tightness of the local CSP inside each agent is also set to 50% .

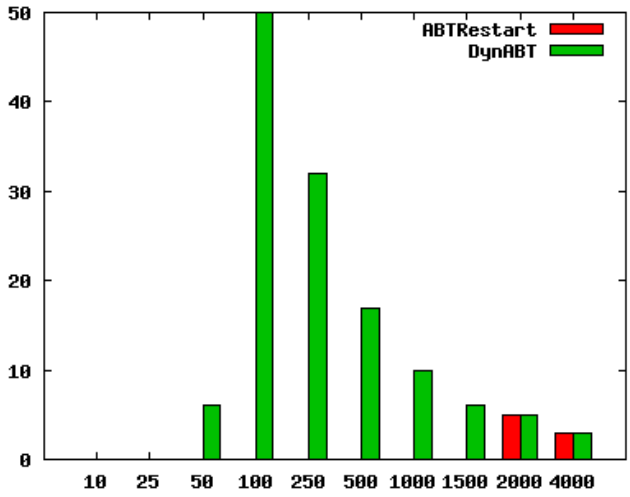
A network of 15 agents with the average transmission delay of 100ms is used in the experiment. The changes introduced to the network vary at different rates to simulate DynDisCSPs. Specifically, a change is introduced at every t seconds where t=10s, 25s, 50s, 150s, 250s, 500s, 1000s, 1500s, 2000s, 4000s during a period of 10000s.

Figure 7.4 compares the performance of ABTRestart against DynABT for different DynDisCSPs. It can be seen that DynABT outperforms ABTRestart in finding solutions for the DynDisCSPs. In particular, for a DynDisCSP, where a change appears at every 1500s (i.e. change interval = 1500ms) or more frequently, ABTRestart fails to find any solution, whereas DynABT can find many solutions. For the change interval of 1000s, DynABT can find 10 solutions. It can be seen that these solutions are all possible solutions for the DynDisCSP since there are only 10000s/1000s = 10 changes into the DynDisCSP.

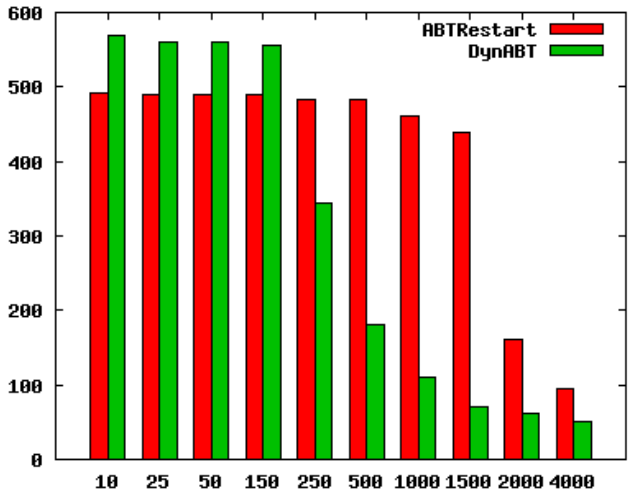
If the changes are more frequent, with change intervals of 500s, 250s, and 100s, DynABT can still find more than half of all possible solutions[4]. DynABT fails to find solutions when the changes are too frequent.

[4]The numbers of all solutions for the DynDisCSPs with change intervals of 500s, 250s, and 100s are 10000s/500s=20, 10000s/250s=40, 10000s/100s =100 respectively.

It is also interesting to note that with the help of *recorded nogood explanations*, the average number of local searchs (i.e. to find own assignments) for DynABT is significantly lower than ABTRestart for DynDisCSPs with infrequent changes (i.e. a change happens at least every 500s). However, for problems with more frequent changes, DynABT does not show any improvement. It is infact slightly worse than ABTRestart for change intervals



(a) Number of solutions.



(b) Number of local searches.

Figure 7.4: DynABT versus ABTRestart.

less than 150s. Nevertheless, this higher computational overhead is compensated for by the high number of solutions returned, as discussed in the previous paragraph.

7.9 Summary

In this section, ABTRestart and DynABT have been proposed as new algorithms to solve DynDisCSPs. Whereas existing DynDisCSP algorithms such as LD-AWC and DynAPO are either limited to solving a sub-class of DynDisCSP, or are not fully distributed, ABTRestart and DynABT can solve general DynDisCSPs and are fully distributed algorithms. Both algorithms are built on top of ABT. ABTRestart is a simple extension of ABT in which a *restart* strategy is used to handle new changes. DynABT is more complex. It uses *nogood explanation* concepts to maintain intra-agent consistency. At the same time, a distributed mechanism is used in DynABT to maintain inter-agent consitency. Both ABTRestart and DynABT are correct and complete.

Whilst being simple, ABTRestart repeats the search whenever a new change is detected. DynABT does not have this limitation. It is more efficient in reusing information during the solving process. An experiment was carried out to confirm this. In the experiment, DynABT was shown to outperform ABTRestart. DynABT can find many more solutions for DynDisCSPs at a similar computational cost as compared to ABTRestart.

Chapter 8

QoS Management with the different preferences of agents*

8.1 Introduction

In the previous chapter, we have considered the dynamic characteristics of the Web service environments and proposed two algorithms, namely ABTRestart and DynABT that can handle changes and uncertainties. These algorithms are satisfaction based. In other words, whilst there are different possible solutions, agents always take the first solution to solve the problem. In this way, the agents' preferences of outcome have not been taken into accounts. Obviously, it is useful to have these preferences considered in order to increase the joint benefit among the agents and provide an incentive for them to participate in the management.

This chapter focuses on agent preferences. It is organized as follows: The QoS management with agents' preferences is modelled as a Distributed Fuzzy CSP (DisFCSP) in Section 8.2. Section 8.3 discusses related work on solving DisFCSPs. Section 8.4 proposes an iterative based approach to solve DisFCSPs using available DisCSP algorithms. This iterative approach reuses the consistency maintenance techniques in DynABT and the existing methods in centralized Fuzzy Constraint Satisfaction (FCSP). Section 8.5 proposes another

*Preliminary versions of the work presented in this chapter have been published in [NK07] and [NKP06c].

algorithm to solve DisFCSPs. This algorithm is extended from ADOPT (Asynchronous Distributed constraint Optimization with quality guarantees) - a new DCOPT (Distributed Constraint Satisfaction) algorithm. In this section, ADOPT is also re-examined for its limitation of message flooding. A performance comparison of the proposed algorithms is presented in Section 8.6. Finally, Section 8.7 concludes the chapter.

8.2 Distributed Fuzzy Constraint Satisfaction

Fuzzy constraint satisfaction problems extend the classical constraint satisfaction problems with a more flexible notion of constraints. A fuzzy constraint can have different degrees of satisfaction represented by a fuzzy relation. Fuzzy constraints, with a fuzzy set based representation, allows expression of partial satisfaction rather than enforcing binary satisfaction/dissatisfaction. Satisfaction of individual constraints can be aggregated via fuzzy operators to obtain the global satisfaction degree for a solution.

A Distributed Fuzzy Constraint Satisfaction Problem (DisFCSP) is a DisCSP where all constraints are fuzzy. The Fuzzy DisCSP can be formally defined as follows:

Definition [Distributed Fuzzy CSP]: a Distributed Fuzzy Constraint Satisfaction problem $\mathbf{P}^f$ is a 4-tuple $\langle \mathbf{V},\mathbf{D},\mathbf{C},\mathbf{A} \rangle$ where $\mathbf{V}=\{\mathrm{v}_1,...,\mathrm{v}_n\}$ is a set of variables. $\mathbf{D}=\{\mathbf{D}_1,...,\mathbf{D}_n\}$ is a set of discrete domains of variables in $\mathbf{V}$, i.e. $\mathbf{D}_i$ is the domain of v_i: i=1,n. $\mathbf{C}=\{\mathrm{c}_1,...,\mathrm{c}_m\}$ is a set of fuzzy constraints. Each constraint restricts possible values of some or all variables in $\mathbf{V}$ to some degree. $\mathbf{A}$ is a set of agents. Each agent holds one or more variables in $\mathbf{V}$ and has a set of constraints in $\mathbf{C}$.

Each constraint c_i in $\mathbf{C}$ represents a fuzzy relation defined on the Cartesian product of its variable domains and is characterized by a membership function. The membership function expresses preferences among assignments by ranking them as being more or less acceptable to the satisfaction of c_i:

$$\mu_{c_i} : \prod_{v_i \in var(c_i)} D_i \longmapsto [0,1], i = 1, ..., m. \tag{8.1}$$

where $var(c_i)$ defines the set of variables of c_i. For an assignment S of variables in $\mathbf{V}$,

we use $S \downarrow c_i$ to denote the assignments of the variables of the constraint c_i in S. The satisfaction degree of this assignment S for an agent A_k is:

$$sat(S_{A_k}) = \oplus \{\mu_{c_i}(S{\downarrow}c_i) : c_i \in \mathbf{c} \ and \ A_k \ has \ c_i\} \tag{8.2}$$

We are interested in the min-DisFCSP problem for which the global satisfaction degree of this assignment **S** is defined as:

$$sat(S) = \oplus \{\mu_{c_i}(S{\downarrow}c_i) : c_i \in \mathbf{c}\} \tag{8.3}$$

in which $\oplus$ is a T-norm operator [Zad75] and commonly used as the *min* function. In general, **S** is a solution of **P** if it maximizes the global satisfaction degree, or it has a global satisfaction degree greater than or equal to a predefined threshold value.

8.3 Related Work

DisFCSP has been of interest to the MAS community, especially in the context of distributed resource allocation, and collaborative negotiation [LLY05; LJS+03]. In those works, a central coordinating agent is often required. For example, the work in [LJS+03] promotes a rotating coordinating agent which acts as a central point to evaluate different proposals sent by other agents. Hence, the algorithms proposed in those studies are not fully distributed.

Generally, DisFCSP can be considered as a case of DisCSP with Optimization. Asynchronous Distributed constraint Optimization with quality guarantees (ADOPT) [MSTY05] and Optimal Asynchronous Partial Overlay (OptAPO) [ML04] are two of the most successful DCOP algorithms recently. OptAPO is not a totally distributed search algorithm (i.e. it attempts to collapse different small parts of a distributed search together into centralized searches), while ADOPT is. ADOPT uses greedy search locally at every agent together with a mechanism to propagate lower and upper bounds on costs (i.e. estimations of the objective function at different agents). ADOPT can also allow optimal solutions to be found within a predefined level of accuracy [MSTY05]. More details of ADOPT are

available in Section 8.5.1 of this chapter.

8.4 FABT - An iteration based algorithm

This section proposes an iterative based algorithm , called FABT (Fuzzy constraint solving using Asynchronous Backtracking), for solving DisFCSPs. FABT uses the iterative principle [ML97], which models a DisFCSP as a series of DisCSP, in conjunction with the *nogood explanations* described in the previous chapter. The basic idea behinds FABT is to use ABT to search for a solution at different levels of the agents' preferences. In addition, moving from one preference level to another is modelled as a dynamic change in which a new preference related constraint is added or an existing one is removed.

8.4.1 The iterative principle

It is well known that a fuzzy set can be represented as a weighted set of α-cuts (i.e. the principle of the resolution identity [Zad75]). Similarly, a Fuzzy CSP can be modelled as a collection of crisp CSPs at different levels of constraint satisfaction:

$$c_j(x) = \sum \alpha c_j^{\alpha}(x) \tag{8.4}$$

in which c_j^{α}(x) is a crisp constraint which is satisfied iff x belongs to the α-cut: $\{x : \mu_{c_j}(x) \geq \alpha, 0 \leq \alpha \leq 1\}$.

Iterative methods (e.g. [ML97]) in centralized CSP use this model to solve a FCSP as a series of CSPs. Similary, a DisFCSP $\mathbf{P}^f$ in general, can be considered as a union of Dis(Crisp)CSPs at different levels of constraint satisfactions: $\mathbf{P}^f = \sum \alpha_i \mathbf{P}_{\alpha_i}$ where a P_{α_i} results from $\mathbf{P}^f$ by replacing all fuzzy constraints in $\mathbf{P}^f$ with their crisp constraints at the α-cut level of α_i. A solution of the DisFCSP is to find a solution at the highest α-cut level. Using this principle, FABT selects and solves a series $\{\mathbf{P}_{\alpha_1}, \ldots, \mathbf{P}_{\alpha_m}\}$ of DisCSP instances at different α-cuts: $\alpha_1, \ldots, \alpha_m$. The decision for selecting α_{i+1} partly depends on the outcomes of previous searches at $\alpha_1, \ldots, \alpha_i$. In addition, a strategy similar to the binary search is used to determine α_{i+1}. In particular, FABT continuously computes and updates

a global α_{LB} (lower bound) and a α_{UB} (upper bound). The α_{LB} is the satisfaction level at which we know that a solution can be found. The α_{UB} is the satisfaction level above which we cannot find any solution. The search level α_{i+1} is computed as the nearest preference level to the middle point between the α_{LB} and α_{UB}. These are shown specifically in the following equations:

$$\alpha_{LB} = max\,\{\alpha_k : k \leq i, P_{\alpha_k}\ solved\} \tag{8.5}$$

$$\alpha_{UB} = min\,\{\alpha_k : k \leq i, P_{\alpha_k}\ overconstrained\} \tag{8.6}$$

$$\alpha_{i+1} \approx (\alpha_{LB} + \alpha_{UB})/2 \tag{8.7}$$

With this binary search-based selection strategy, in the worst case for random constraints, FABT executes a number of $log_2 N_l$ ABT sessions.

8.4.2 Moving between α-cut levels

In FABT, when a solution is found or overconstraint is detected, an agent in the coalition computes the next α-cut level according to the equation (8.7). The agent then propagates a request to other agents asking them to move their local searches to the new α-cut level. This moving between different α-cut levels is modelled as a dynamic change in FABT. An increase in the α-cut level is equivalent to the addition of the constraint $c_{pf_{i+1}}$: $sat_{own} \geq \alpha_{i+1}$ to all agents. Similarly, a decrease in the α-cut level corresponds to removals of the existing constraint c_{pf_i}: $sat_{own} \geq \alpha_i$ from all agents. FABT uses existing techniques in DynABT to handle these changes.

In a similar way to DynABT, the consistency of agents' local solving, when switching between different α-cut levels, is maintained with the help of *nogood explanations*. We note that for two *nogood explanations* $\ell_i = c \bigwedge c_{pf_i}$ and $\ell_j = c \bigwedge c_{pf_j}$ and $\alpha_j \leq \alpha_i$ (i.e. c_{pf_j} is more restricted than c_{pf_i}), then $\ell_j \longrightarrow \ell_i$. This indicates that ℓ_i can be made more optimal (to become ℓ_j) if c_{pf_i} is replaced with c_{pf_j}. Using this property, in FABT, every *nogood explanation* is created using the most restricted preference constraints.

As discussed in the previous chapter, *nogoods* can be better reused if more optimal

explanations of the *nogoods* are recorded. Consequently using the most restricted preference constraints in *explanations* can bring improvement in FABT since this increases the optimality of *nogood explanations*. To give an alternative explanation for the improvement, let us compare the *nogood explanations* with, and without, optimizing the explanations. Without optimizing the explanation, every *nogood* which has a preference constraint at the α-cut level i (in the explanation) is re-created and removed during searches at every preference level $\alpha_j \geq \alpha_i$. In contrast, by using *nogood explanation* with the most optimal preference constraints, we can efficiently retain the appropriate *nogoods* during successive ABT searches without removing and re-creating them.

8.4.3 Correctness, completeness, and termination of FABT

FABT is complete and terminated as stated in the following proposition:

Proposition 8.1 [Termination and Completeness]: FABT is a terminated and complete algorithm.

Proof: We can see that there is only a limited number of ABT executions called during the execution of FABT since the number of preference levels is finite. ABT is known to be terminated and thus the termination of FABT is consequently proved.

The completeness of FABT can also be proved using the inequalities (8.5) and (8.6). According to these inequalities, the final preference level α (after the algorithm terminates) should be the maximum satisfaction degree that can be found. FABT, by using ABT - a complete algorithm, can always find a solution on the preference level α if such a solution exists. Therefore FABT is complete. ■

8.5 ADOPT-am and ADOPT-fm – ADOPT based algorithms for DisFCSPs

8.5.1 ADOPT Algorithm

A brief summary of the ADOPT algorithm is presented in this section. In ADOPT, a number of n agents collaborate to solve the optimization problem:

$$min \sum_{1 \leq i \leq j \leq n} f_{ij}(v_i, v_j) \tag{8.8}$$

in which for every pair, i, j, $f_{ij}(d_i, d_j)$ is a function which defines the *cost* incurred when agents A_i and A_j select the values d_i and d_j for their variables v_i and v_j respectively.

Before ADOPT can proceed, a Depth First Search (DFS) tree is set up among the agents. Two agents share a constraint only when they both belong to a branch of the tree. Priorities are also assigned to agents as in ABT. The higher position an agent has in the DFS tree, the higher priority the agent has. ADOPT defines four types of messages: *VALUE*, *COST*, *THRESHOLD*, and *TERMINATE*. *VALUE* messages are used by an agent to inform its children about the agent's own assignment. Those messages are equivalent to the *ok?* messages in the ABT algorithm. For an agent, the assignments from its parents are kept in the agent's *view* (i.e. *CurrentContext* Agents exchange messages during the search [MSTY05]). *COST* messages are sent from an agent to its DFS parent (i.e. the agent at the parent node in the DFS tree) to report on the temporarily lower *LB* and upper bounds *UB* of the accumulated cost of the subtree rooted at this agent. *THRESHOLD* messages, each of them contain a threshold level, are sent by a parent to its children to hint for a more efficient local search at the children. *TERMINATE* messages notify agents about the terminations of their DFS parents. Upon finding a solution or detecting an overconstraint, termination of ADOPT occurs top-down, the agent rooted at any subtree terminates before agents lower in the sub-tree.

During ADOPT execution, an agent A_i holding v_i continues to update its *LB* and *UB*

as follows:

$$\delta(d_i) = \sum_{(v_j,d_j)\in View} f_{ij}(d_i, d_j) \tag{8.9}$$

$$\forall d_i \in D_i, LB(d_i) = \delta(d_i) + \sum_{A_l \in Children} lb(d_i, v_l) \tag{8.10}$$

$$\forall d_i \in D_i, UB(d_i) = \delta(d_i) + \sum_{A_l \in Children} lb(d_i, v_l) \tag{8.11}$$

In which $\delta(d_i)$ is the local cost incurred by shared constraints between A_i and its parents. $v_i = d_i$ is the assignment selected by A_i, and $lb(d_i, v_l)$ is the LB reported previously from A_l-a children of A_i for this assignment.

In general, A_i tries to select an assignment of its own value so that LB or UB is smallest. LB has higher precedence than UB:

$$LB = min_{d_i \in D_i} LB(d_i) \tag{8.12}$$

$$UB = min_{d_i \in D_i} UB(d_i) \tag{8.13}$$

Algorithm 10: ADOPT algorithm skeleton

```
1  procedure adopt-initialize do
2      reset costs, LB, UB, and threshold;
3      adopt-backtrack;
4  procedure adopt-process-threshold do
5      adopt-pre-threshold;
6      adopt-backtrack;
7  procedure adopt-process-terminate do
8      adopt-pre-process-terminate;
9      adopt-backtrack;
10 procedure adopt-process-value do
11     adopt-pre-value;
12     adopt-backtrack;
13 procedure adopt-process-cost do
14     adopt-pre-cost;
15     adopt-backtrack;
16 procedure adopt-backtrack do
17     adopt-local-search;
18     send VALUE messages to children;
19     MaintainAllocationInvariant;
20     if threshold = UB and (parent is terminated or this agent is the root) then
21         send TERMINATE to children and terminate;
22     send COST to children
23 procedure adopt-local-search do
24     if threshold=UB then
25         assign d_i to the assignment that minimizes UB;
26     else
27         if LB(d_i)>threshold then
28             assign d_i to the assignment that minimizes LB;
```

As shown in Algorithm 10, there are three main ADOPT procedures *adopt-process-value*, *adopt-process-cost*, *adopt-process-threshold*, which process incoming *VALUE*, *COST*, and *THRESHOLD* messages respectively. The details of these procedures are not relevant to our discussion and their pseudocode is omitted here. It is important to note that all these procedures call the *adopt-backtrack* procedure at the end. In Figure 1, we define the code block in *adopt-process-** (i.e. *adopt-process-value*, *adopt-process-cost*, or *adopt-process-threshold*), which is placed before the line that calls *adopt-bactrack*, as *adopt-pre-*

*process-**. We will refer to them in succeeding algorithms. These procedures update the agents on the partial global solution and maintain the following relationships:

$$LB \leq threshold \leq UB \tag{8.14}$$

$$lb(d, x_j) \leq t(d, x_j) \leq ub(d, x_j) \tag{8.15}$$

$$threshold = \sum_{x_j \in Children} t(d, x_j) \tag{8.16}$$

(8.14) and (8.15) must be enforced if the costs reported from children or thresholds sent by the parent are not in the local permitted ranges. (8.16) is important for ADOPT's termination through ensuring the condition $UB = threshold$ in the *adopt-local-search* procedure will eventually be met.

8.5.2 ADOPT and message flooding

Whilst ADOPT incorporates new ideas into solving DCOP, its original version suffers from message flooding, which has been overlooked so far. Interestingly, this message flooding is not difficult to show. Every time an agent receives a *COST*, *VALUE*, *TERMINATE*, or *THRESHOLD* message, it does some pre-processing and then calls the *adopt-backtrack* procedure (see Alogirithm 10). This backtrack call can be seen in Line 2 of *process-threshold*, *process-terminate*, *process-value*, and *process-cost*. The *adopt-backtrack* procedure then sends a *COST* message to the parent and *VALUE* messages to all children (line 2 and 5 of the *adopt-backtrack* procedure) provided that neither the agent is the root, nor the agent's parent has terminated. In other words, every time a non-root agent consumes a message, it generates $m + 1$ messages, in which m is the number of its children. Because of this unbalance between message consumption and generation, assuming that the DCOP problem is non trivial, a large number of messages will be built up quickly. Figure 8.1 shows the average numbers of generated messages before the root agent terminates for the map coloring problem with a varied number of agents and the link density of 3 (link density = number of links/number of agents). These numbers are very significant. For example, in a network of 22 agents, the number of messages before termination of any agent is over

200,000, and grows exponentially with the number of agents. It is important to point out

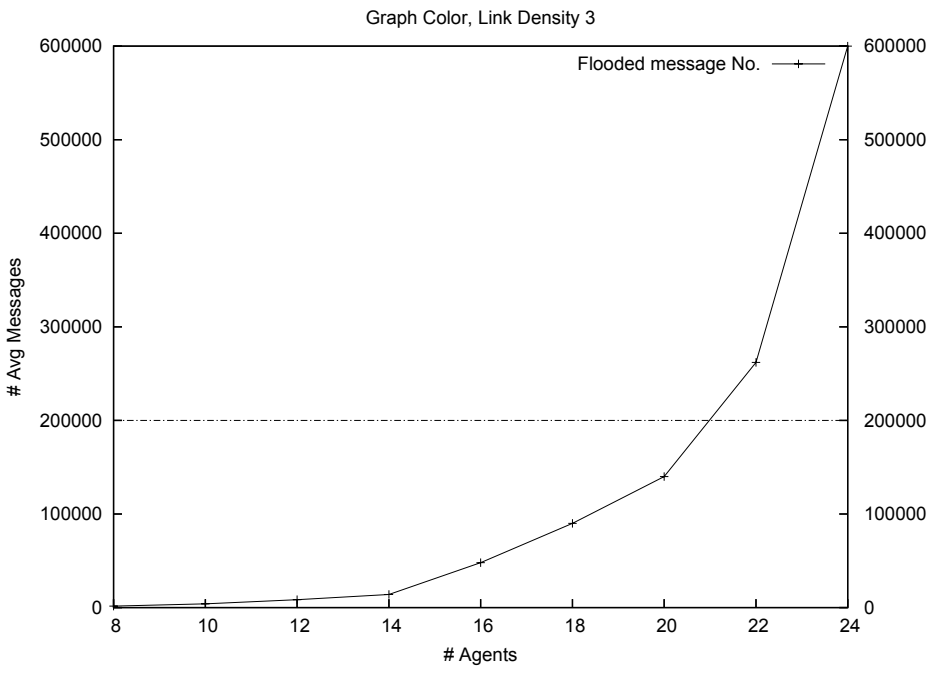


Figure 8.1: Message Flooding in ADOPT.

that message flooding happens as long as the number of generated messages is greater than the number of consumed messages on average, regardless of how much greater. This is because, in general, a DisCSP optimization problem is NP hard and there is no polynomial upper bound guaranteed on the completion time with respect to the number of agents and domain sizes. Over this period of time, generated messages will flood the network links or agents' local queues. With message flooding, an algorithm may still work if flooding messages are redundant or their contents do not contribute to the search process, and hence it is not necessary to process them. However, a further investigation into ADOPT reveals that this is not the case. In fact, the flooding behaviour may cause important messages to be dropped out and consequently affect the correctness of the algorithm. It

is important to note that this flooding problem cannot be solved by using the mechanism described in Chapter 6, since the problem originates from the design of ADOPT itself. Let us consider a particular case where every link of the network has some delay and this delay is negligible compared to the message processing time at each agent. With this assumption, messages travel quickly and are queued at receivers. The solving process can be divided into different *cycles* [YDIK92]. For an agent, a cycle consists of receiving messages, processing the messages, and sending out messages. Since m+1 messages are received and only the first message is processed, m messages are added to a buffer, which is an on-waiting-list queue of messages to be processed, after a cycle is passed. We note that ADOPT is similar to any other ABT based algorithm in the principle that they need to propagate the effect of a local change (e.g. a new assignment) in an agent to others through its neighbours. For an agent A_p and any of its ancestor A_q (i.e. A_p and A_q are in the same DFS branch) in ADOPT, a change in value of A_p (i.e. new values of its *lb* and *ub*) needs to reach and be processed by A_q. Figure 8.2 illustrates this. Assuming that there are k agents between A_p and A_q and at the cycle i-th from the beginning, a new *COST* message is sent from A_p. Because m messages are added after each cycle, at the i-th cycle, A_q already has $m.i$ messages queuing. For the change in the COST message to arrive at A_q, at least k-cycles are needed for k new *COST* messages generated at each intermediate node. Therefore, by the time a new *COST* message which reflects the change in the original *COST* message reaches A_q, there are already $m.(i+k)-k$ messages queued at A_q. For low constraint density i can be very large (at least 20,000 cycles for the map color problem with total variables=15 and link-density =3 as shown in [MSTY05]) and our extended experiment illustrated in Figure 8.1. Again, since a DisCSP optimization problem in general is NP hard, there is no guaranteed polynomial upper bound for i. In other words, it is impractical to build a buffer which can hold $m.(i+k)-k$ messages for every case. Consequently, the information of a change in A_p is not guaranteed to reach A_q.

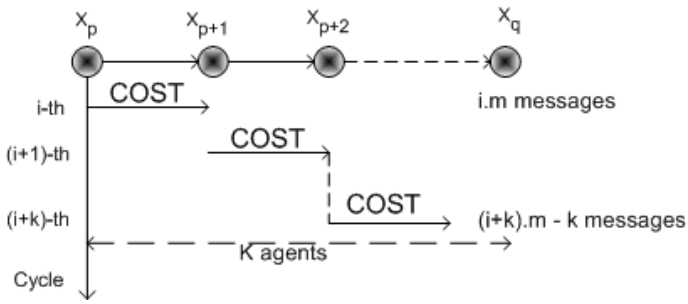


Figure 8.2: Local buffer flooding.

8.5.3 Message balancing for ADOPT

The first common response for resolving the flooding problem is the suggestion that *every agent processes all messages available in its queue before calling the backtrack.* Such a naive solution, in fact, does not solve the problem. To explain this, let us consider an example of a network where transmission delays between any two different agents are different and these differences are significant as compared to the time required for an agent to process a single message. In this network, an agent eventually has the maximum of only one message in its queue at all times. Consequently, the agent processes only one message at a time. Therefore, the same arguments in the previous section are still valid in showing that message flooding cannot be avoided if ADOPT is used in this network.

In general, there has been little consideration that message flooding may severely affect the use of DisCSP algorithms in real life application. In asynchronous communications, an agent does not block its execution thread to wait for incoming messages from other agents. Instead, the agent keeps a local queue to store messages destined to the the agent. It then retrieves and processes the messages when necessary. Often DisCSP algorithms assume that all link delays are the same. The same assumption is made for message processing delays at agents. Consequently, all sent messages are assumed to arrive at the same time and are consumed immediately by the agents. There are no queuing messages. To date, the size limit of message queues held at the agents has not been seriously considered in DisCSP literature.

Taking into account the fact that there are limits on the size of the agents' message queues and link throughputs, message flooding can inevitably cause:

- overloading of links: Messages in a link are dropped if the rate of transmitting messages bypasses the link's *throughput*. The *throughput* of a link is defined in this thesis as the average number of messages that can be delivered through this link over a pre-determined time interval.

- overbuffering of local queues: local queues of an agent are over-buffered if the agent's *message processing rate* is smaller than the total rate of incoming messages from neighbours. The *message processing rate* of an agent is defined as the average number of messages an agent can process over a pre-determined time interval.

For simplicity, we assume that an agent A_i has the average processing rate of ρ_i (over every time interval $\Delta\tau$). A_i's average message sending rate to an agent A_j during the solving process is $\zeta_{i,j}$. Let us denote the throughput of a link between the agent A_i and A_j as $\vartheta_{i,j}$. These parameters are also defined over $\Delta\tau$. Often in DisCSP algorithms, new messages are sent by an agent only after the agent has completely processed all messages in its queue, i.e:

$$\sum_{\forall j} \zeta_{i,j} \leq \rho_i \tag{8.17}$$

To avoid link overloading, the sending rate should be no more than the transmission rate:

$$\forall i, j : \zeta_{i,j} \leq \vartheta_{i,j} \tag{8.18}$$

To avoid overbuffering of local queues, the incoming message should be no more than the processing rate:

$$\sum_{\forall j} \zeta_{j,i} \leq \rho_i \tag{8.19}$$

The condition (8.18) is not of interest to us since it does not indicate a possible problem

arising within the protocol design. Here we assume that any $\vartheta_{i,j}$ is significantly greater than all ρ_i and the condition (8.18) always holds true. Also, we assume that there is a fixed value on the lower and upper bounds of all link delays. The condition (8.19) is a critical requirement to avoid message flooding. In the following subsection we propose two modified versions of ADOPT so (8.19) can always be satisfied.

8.5.4 ADOPT-*

This section proposes two modified versions of ADOPT, ADOPT-am and ADOPT-fm. Pseudocodes of ADOPT-am and ADOPT-fm are listed in Algorithms 11 and 12, in which most of the procedures are extended from ADOPT and can be referred back to Algorithm 10.

ADOPT-am algorithm

ADOPT-am is a simple modification of ADOPT. It is based on the concept of *cycles* in DisCSP [YDIK92]. ADOPT-am balances the number of messages generated and consumed in every cycle. Each agent keeps a counter called *cycle number* on the number of executions of the backtrack procedure. This number is embedded in every exchanged message. During the solving process, each agent performs a local search only after it has received the messages sent in the same cycle (i.e. having the same *cycle number*) from all of its ADOPT neighbors. Here, the postfix *am* stands for *all messages* in the same cycles.

The algorithm is described in Algorithm 11. ADOPT-am introduces a new procedure called *adopt-am-backtrack* as a replacement for *adopt-backtrack*. This procedure checks whether all messages in the same cycle from ADOPT neighbors have been received (Line 20) before calling the *adopt-backtrack*. For the algorithm to work, this condition should not be enforced at the very beginning (*in adopt-am-intitialize* procedure). Also, the value of cycle numbers recorded (in *adopt-am-process-terminate* procedure) is set to ∞ for terminated agents.

As mentioned before, ADOPT local search is carried out in the *adopt-backtrack* procedure. Therefore, a major portion of the processing time is spent on this procedure. Denote d_{max} and d_{min} are the upper and lower bounds respectively of link delays. An agent fin-

ishes one cycle in at least a time of d_{min}. Hence, during a period of d_{max}, there are no more than d_{max}/d_{min} messages sent to this agent from each of its neighbors. Therefore, as long as $\rho_i \geq r = m.d_{max}/\mathrm{d}_{min}$, in which m is the number of neighbors of the agent, the condition (3) holds. We note that r has a fixed value that can be calculated before ADOPT-am execution.

Algorithm 11: ADOPT-am algorithm

```
1  procedure adopt-am-initialize do
2      cycle-number=1;
3      expected-msg-count=total ADOPT neighbors;
4      adopt-initialize;
5  procedure adopt-am-process-threshold do
6      if sender A_j is not my DFS parent then
7          return;
8      adopt-pre-threshold;
9  procedure adopt-am-process-terminate do
10     record cycle number of the sender as ∞;
11     adopt-process-terminate;
12 procedure adopt-am-process-value do
13     adopt-am-process-threshold;
14     adopt-pre-value;
15     adopt-am-backtrack;
16 procedure adopt-am-process-cost do
17     adopt-pre-cost;
18     adopt-am-backtrack;
19 procedure adopt-am-backtrack do
20     if cycle numbers recorded for all neighbors≥my cycle number then
21         backtrack;
22         cycle-number = cycle-number +1;
```

Our investigation into the existing implementation of ADOPT reported in [MSTY05], which is available at [Pra03], reveals that it uses an equivalent algorithm to ADOPT-am, not as the original algorithm specified in the original paper [Pra03]. Therefore, message flooding did not happen in that implementation. The disadvantage of ADOPT-am (and hence the original ADOPT implementation), is that, with link delays, the time needed for each cycle is the maximum delay of the network links.

ADOPT-fm algorithm

ADOPT-fm is our second modified version of ADOPT to resolve the problem of the maximum link delay in ADOPT-am. The postfix *fm* stands for *first message*. In ADOPT-fm, any inconsistencies between assignments are propagated immediately upon being detected by an agent. This idea is closer to the original ADOPT version [MSTY05] than its reported implementation [Pra03]. However, because this can potentially lead to message flooding, a message balancing mechanism is used in addition. The idea is to set an upper bound on the number of messages sent by an agent to any of its neighbours in every time window $\Delta\tau$. To ensure the condition (3) is met, this upper bound is set by the receiver agent. In other words, ADOPT-fm assumes that before the algorithm starts, every agent A_i informs each of its neighbours A_j about the maximum number of messages $\zeta_{j,i}(\Delta\tau)$ it can receive from that agent in a time window $\Delta\tau$.

As shown in Algorithm 12, during ADOPT-fm operation, an agent A_i keeps track of the number of messages it sends to another agent A_j in the variable *msg-sent-count(j,*$\Delta\tau$*)*, which is initialized to 0 (Line 3 in *adopt-fm-initialize* procedure) at the beginning. The procedure *adopt-fm-backtrack* is a modification of *adopt-backtrack* in which message are not sent immediately, but by calling *adopt-fm-message-send* (Lines 34, 38, and 40 in *adopt-fm-backtrack*). Before sending any message to A_j, the *adopt-fm-message-send* checks if no more than $\zeta_{j,i}(\Delta\tau)$ messages have been sent to A_j (Line 23 in *adopt-fm-msg-send*). This guarantees that there is a maximum of $\zeta_{j,i}(\Delta\tau)$ messages sent during $\Delta\tau$. Consequently, some updates received by an agent may not be propagated to its neighbours during a $\Delta\tau$ time window. To handle this, the backtrack procedure is called, if the current $\Delta\tau$ window expires (in Lines 29-30 of the *adopt-fm-msg-send* procedure), to propagate these changes as soon as possible.

Algorithm 12: ADOPT-fm algorithm

```
1  procedure adopt-fm-initialize do
2      forall a_j ∈ my-neighbors do
3          msg-sent-count(j,Δτ)=0;
4          last-sent-msgs(j)= empty array;
5      adopt-initialize;
6  procedure adopt-fm-process-threshold do
7      if the sender A_j is not my DFS parent then
8          return;
9      adopt-pre-threshold;
10 procedure adopt-fm-process-terminate do
11     adopt-process-terminate;
12 procedure adopt-fm-process-value do
13     adopt-fm-process-threshold;
14     adopt-pre-value;
15     adopt-fm-backtrack;
16 procedure adopt-fm-process-cost do
17     adopt-pre-cost;
18     adopt-fm-backtrack;
19 procedure adopt-fm-msg-send(A_j,msg-type) do
20     generate new message msg with type of msg-type;
21     if msg has the same content as the last element of last-sent-msg(j,msg-type)
       then
22         return;
23     if msg-sent-count(j, Δτ) ≤ ζ_{j,i}(Δτ) then
24         send message of type msg-type to A_j;
25         update and append a copy of the previous sent message to last-sent-msgs(j);
26         remove the first message from last-sent-msgs(j);
27     else
28         first-msg = the first element of last-sent-msgs(j);
29         t=received timestamp of first-msg+Δτ;
30         (re-)schedule to send to A_j a message of msg-type;
31 procedure adopt-fm-backtrack do
32     adopt-local-search;
33     forall child A_j do
34         adopt-fm-msg-send(A_j,VALUE);
35     MaintainAllocationInvariant;
36     if threshold =UB and (parent is terminated or this agent is the root) then
37         forall child A_j do
38             adopt-fm-msg-send(A_j,TERMINATE);
39         cancel any scheduling tasks and terminate;
40     adopt-fm-send(parent-id,COST);
```

Another improvement of ADOPT-fm is the removal of duplicated messages. The original version of ADOPT generates many duplicated messages as discussed in Section 8.5.2. In ADOPT, messages which arrive earlier are processed first. If these messages are duplicated then not-duplicated messages that arrived late are forced into the next time window and hence the solving process is slowed down. To improve this, ADOPT-fm cuts off any consecutively duplicated messages. Each agent keeps a list of last sent messages (i.e. the *last-sent-msgs* variable in Algorithm 12) to any of its neighbours. Before sending any message to an agent, it checks whether the content of this message is the same as the previously sent message to this agent. The sending operation is aborted if such a duplication is detected. These actions are shown in Line 1 of *ADOPT-fm-msg-send* procedure. Note that this detection rule is not used if no messages have been sent in the last $\Delta\tau$.

Unlike ADOPT-am, ADOPT-fm does not suffer from experiencing the longest link delay. It also has another significant advantage as compared to ADOPT-am. ADOPT-fm can exploit available network resources to the best extent. ADOPT-fm uses the maximum processing power available at an agent, by encouraging the agent to process as many messages as it can. In contrast, ADOPT-am always processes fewer than $r=$ m.$delay_{max}$/delay$_{min}$ messages. An agent is not allowed to process more than that even if it is capable at doing so. Therefore, ADOPT-fm is able to provide a solution in a shorter time. However, it requires more processing resources. Our experiments presented in Section 8.8 confirm these observations.

8.6 ADOPT-*'s completion

We use OPT(x$_i$,*context*) to denote the cost of the optimal solution in the subtree rooted at x$_i$. ADOPT termination and correctness are based on three properties which were proven in [MSTY05]:

Property 1:

$\forall x_i \in V : LB \leq OPT(x_i, CurrentContext) \leq UB$

Property 2:

$\forall x_i \in V$, if $CurrentContext$ is fixed, then $threshold = UB$ will finally be met.

Property 3:

$\forall x_i \in V$, x_i's final $threshold$ value is $OPT(x_i, CurrentContext)$

Properties 1 and 2 ensure ADOPT's termination since the condition at Line 20 in Algorithm 10 will eventually be true. Property 3 guarantees the correctness of the ADOPT algorithm.

To prove the termination and correctness of ADOPT-*, first we need to prove that no deadlock can happen to ADOPT-*. This is important as ADOPT-* uses weak synchronizations as compared to ADOPT. A deadlock happens in ADOPT-* when an agent, which has not terminated yet, has to wait for the fulfillment of some *synchronization* condition in order to carry out its local search and send out messages; however, this condition, which is triggered by other agents, can never be met.

Proposition 8.2 [ADOPT-am Deadlock free]: ADOPT-am is free from deadlock.

Proof: We denote the cycle number in ADOPT-am for an agent A_i as $c(A_i)$. By contradiction, suppose that a deadlock occurs to an agent A_k during its ADOPT-am execution. This deadlock happens only when A_k is starved by some other agent A_{k_1}, i.e: $c(A_k)>c(A_{k_1})$, at any time $\tau \geq \tau_k$. From Line 20 in Algorithm 11$\Longrightarrow$ $c(A_k)$ is unchanged from the time τ_k and since $c(A_{k_1})$ cannot increase forever, (its upper bound is $c(A_k)$)$\Longrightarrow \exists \tau_{k_1}$: $c(A_{k_1})$ is constant $\forall \tau \geq \tau_{k_1} \Longrightarrow$a deadlock happens to A_{k_1}.

By applying the above arguments to A_{k_1} over again and repeating this process, we obtain an infinite sequence of positive integer numbers starting from a constant $c(A_k)$

$c(A_k)>c(A_{k_1})>c(A_{k_2})$...

This cannot happen, hence the above proposition is proved. ■

Proposition 8.3 [ADOPT-fm deadlock free]: ADOPT-fm is free from deadlock.

Proof: Since an agent in ADOPT-fm calls the backtrack procedure every time a message is received, a deadlock happens only when no message is sent to an unterminated agent. We prove that this cannot happen by contradiction. Suppose that an unterminated agent A_i is not going to receive any messages after the cycle k-th. If A_i is a leaf of the DFS tree and its DFS parent has terminated, then the agent will eventually receive a TERMINATE

message from this parent since each TERMINATE message is sent only once and is not subjected to being deleted due to duplication. Therefore, this cannot happen and there exists a neighbor of A_i which also has not terminated. This neighboring agent is either one of A_i's DFS children if A_i is not a leaf, or A_i's DFS parent if otherwise. If the neighbor receives a message in the *(k-1)*-th cycle then it will eventually send A_i a message, since even if the synchronization condition (Line 23 of Algorithm 12) fails, a message will be scheduled to be sent later to A_i (Line 30 of Algorithm 12). Consequently, no message is sent to the neighbor agent in the *(k-1)*-th cycle. On repeating all above arguments over and over again, eventually, it becomes clear that there exists one agent which receives no messsage at the beginning (the first cycle) and this is contradictory. Therefore, the proposition is proved. ■

Proposition 8.4 [ADOPT-* Completeness]: ADOPT-* is terminated and completed.

Proof: Since ADOPT-* is free from deadlock, *adopt-backtrack* code is guaranteed to be called after an agent receives new messages. Therefore, Properties 1,2, and 3 can still be applied for ADOPT-am and ADOPT-fm using the same proof in [MSTY05]. These properties ensure the termination and completion of ADOPT-*. ■

8.7 ADOPT adaptation for DisFCSP

ADOPT adaptation for DisFCSP consists of two steps. The first step is to compute ADOPT cost using the fuzzy constraints at each agent in a DisFCSP. In particular, the cost for an assignment of v_i at the agent A_k can be defined as:

$$f_{ij}(v_i, v_j) = \begin{cases} 1 - sat_{A_i}(v_i) & \text{if } i{=}j \\ 0 & \text{if } i{\neq}j \end{cases}$$

where sat_{A_i} is defined in (3). The global objective of the DisFCSP is:
$max\ min_{1 \leq i \leq n} \{sat_{A_i}(v_i)\} \Leftrightarrow min\ max_{1 \leq i \leq n} \{f_{ij}(v_i, v_j)\}$ for ADOPT adaptation. Note that we have a maximization for the aggregator instead of a summation as in the original objective function (8.8) of ADOPT .

The second step for ADOPT adaptation for DisFCSP is to redefine the computation of LB and UB, since our global objective function is a *min max* operation. Instead of (11) and (12), we redefine these as:

$$\forall d_i \in D_i, LB(d_i) = max\,\{\delta(d_i), max_{\forall: A_l \in Children} lb(d_i, v_l)\} \tag{8.20}$$

$$\forall d_i \in D_i, UB(d_i) = max\,\{\delta(d_i), max_{\forall: A_l \in Children} ub(d_i, v_l)\} \tag{8.21}$$

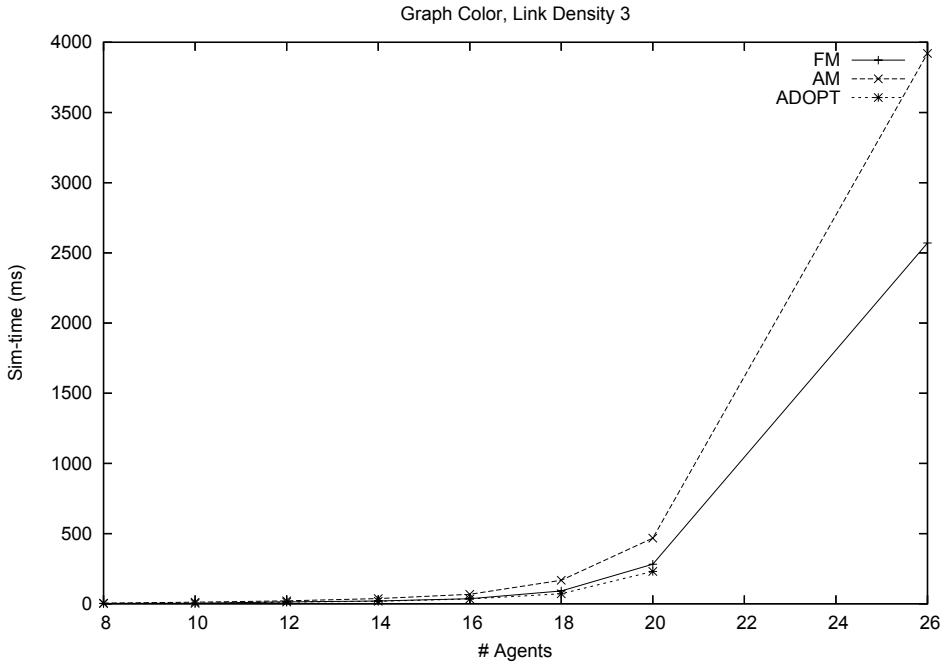
For the changes above in the objective function and formulas of LB and UB, the same arguments in [MSTY05] for ADOPT's termination and correctness can still be applied. In other words, our ADOPT adaptation can be used to solve the DisFCSP problem.

8.8 Experiments

This section is divided into two subsections. In the subsection, ADOPT-* is compared against ADOPT. The purpose of this subsection is to illustrate the scalability of ADOPT-* for DisCSPs with large numbers of agents. ADOPT does not scale well for such problems due to message flooding. Instead of using a QoS management problem, the map coloring problem - a popular benchmark problem for DisCSP algorithms, is used in the experiment. This is to replicate the original test on ADOPT performance in [MSTY05] and to show that ADOPT-* has better performance than ADOPT for the same experiment. In the second part, an experiment using ADOPT-* and FABT to solve QoS management problems is carried out. In addition, a comparison of the performance of these algorithms is reported.

8.8.1 ADOPT-fm versus ADOPT-am

An experiment was carried out to compare the two improved versions of ADOPT versus the original version of ADOPT, and to show the tradeoff between computational complexity and traffic load, versus completion time for these algorithms. In the experiment, the simulated message delays are generated randomly between t_{min}=*5ms* and t_{max}=*10ms*. To ensure the message ordering, the time delay t of a message sent from an agent A_i to another agent A_j is generated randomly between $max\{t_{last\_arrival}$ - $t_{current}$, $t_{min}\}$ and



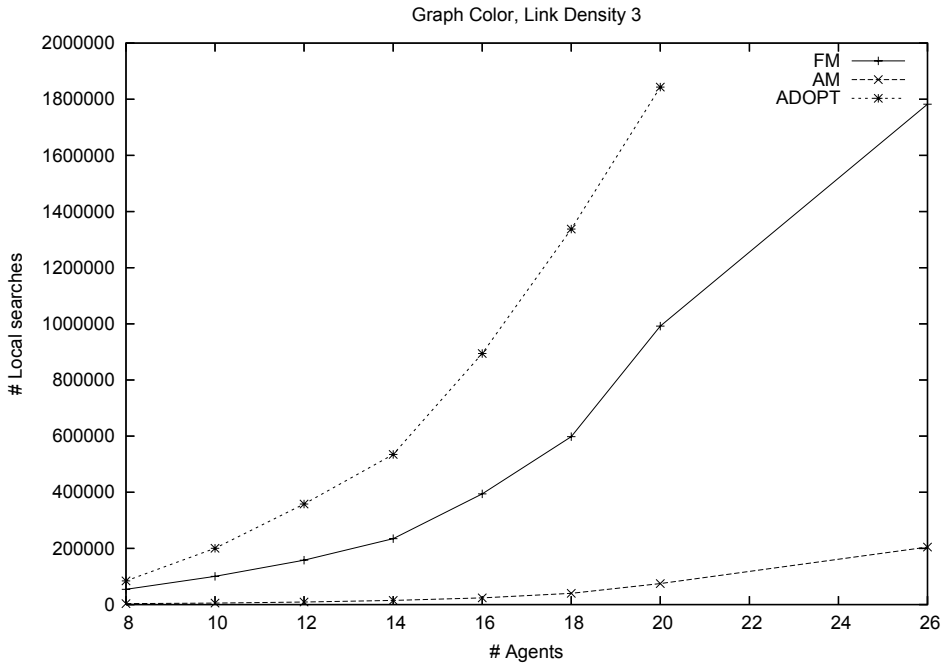


Figure 8.3: Time: ADOPT-FM versus ADOPT-AM.

t_{max}, in which $t_{last\_arrival}$ is the time that the previous message sent by A_i reaches A_j, *current_t* is the current time. This random generation can be explained as $t + t_{current} \geq max\{t_{last\_arrival} - t_{current}, t_{min}\} + t_{current} \geq t_{last\_arrival}$, or the current generating message is guaranteed to arrive A_j after the arrival of a previously sent message.

ADOPT-* was tested on the Map Coloring problems of which the number of variables (agents) are increased from 8 to 26 with a link density of 3. The message sending rate on each link is set to 50 per t_{max} for ADOPT-fm. Note that this number is significant as compared to that of ADOPT. It is approximately 16.6 folds of 3–the average number of messages per t_{max} an agent sends in ADOPT-am. As shown in the first graph at the top of Figure 8.3, all algorithms (ADOPT, ADOPT-am, ADOPT-fm) achieve a similar completion time with up to 20 agents. *For a bigger number of agents (26), the experiments on ADOPT cannot be completed due to message flooding* (as shown in the graph - this is consistent with the report that the computational complexity for ADOPT is too high [PF07], however, no specific reason is given in this study) while ADOPT-fm and ADOPT-am were able to solve the problem and ADOPT-fm has 25% shorter time than ADOPT-am. However, this improvement is at the expense (or better exploitation) of computation and network load. The second graph, at the bottom of Figure 8.3, shows an increasing of 10 folds in computation for ADOPT-fm as compared to ADOPT-am. This was predictable as ADOPT-fm is running with a message sending rate (and hence processing rate) of 16.6 times that of ADOPT-am. More importantly, ADOPT-fm computation is only around half that of the original ADOPT. This can be explained as the result of duplicated message removal in ADOPT-fm.

8.8.2 FABT versus ADOPT-* for QoS of Web service compositions

The experiments so far had been tested preliminarily on Map Colouring problems. The reason is that the original ADOPT was tested only on the Map Colouring problems. This section describes the experiment carried out for the QoS management problem.

In the experiment, the total number of agents (i.e. service providers) – n varies from 8 to 35. Similar to the experiments in Chapter 6, compositions are added between 2 or

3 randomly chosen agents. The total number of compositions is $1.5n$. This ensures that each agent is involved in an average of 3 to 5 compositions. Average link delay is 20ms. Three different QoS parameters were considered: Cost, Service Availability, and Response Time. The aggregation formulas for these QoS parameters can be found in Chapter 5. Each parameter has a domain of 10 discrete values. Five levels of preference: 0.00, 0.25, 0.50, 0.75, and 1.00 are of interest to the agents. The preference is consistent with the observation in Chapter 6: contracting higher cost and response time, and lower service availability are more preferred by a service provider. The tightness of each end to end QoS constraint for every composition is 50%.

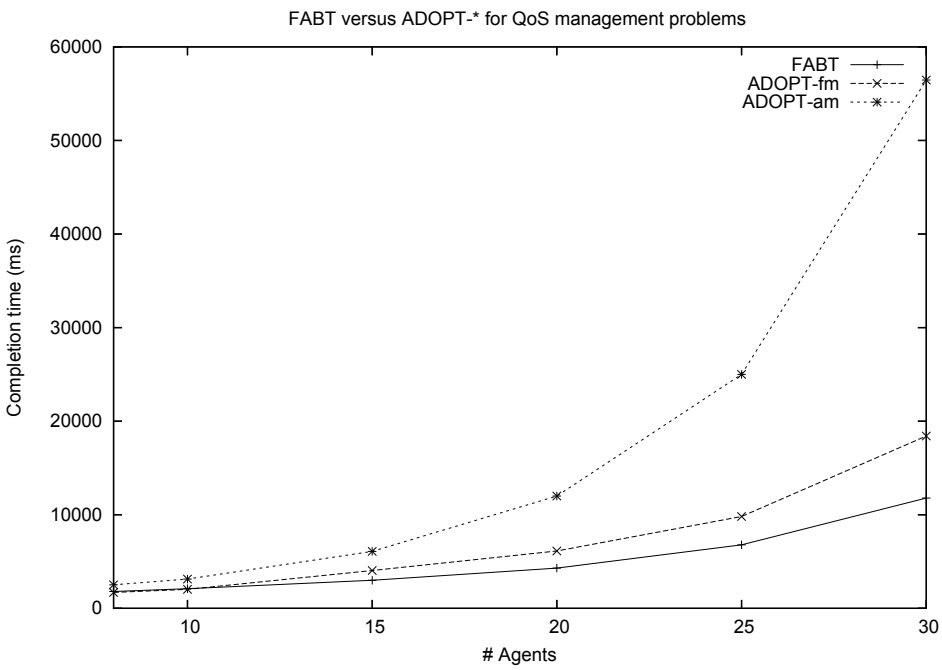


Figure 8.4: FABT vs ADOPT-.*

Figure 8.8.2 shows the performance of FABT versus ADOPT-* in solving the QoS problem with service providers' preferences in terms of completion time. It can be seen

that, in general, FABT performs better than ADOPT-*. If the number of agents is greater than 15, the completion time of ADOPT-fm is at least 30% greater than FABT. ADOPT-am is much worse. The completion time of ADOPT-am is double that of FABT for 20 agents, and six folds of FABT for 30 agents.

For an approximation to real life with a network of 30 agents, taking into account the processing delay, we can increase the link delay to 10s instead of 20ms. This number is quite reasonable as most of the local solving took under 10s in real time during the experiment using the Choco solver [RHF+05]. The expected completion time is roughly 12000ms*(10000/20)≈ 1 hour and 40 minutes for FABT. This number is approximately 2 hours and 20 minutes for ADOPT-fm, and 8 hours for ADOPT-am respectively. Nevertheless, the experiment was run on an average Intel computer with 1 Gigabyte RAM (allocated to the simulation) and 1.3GHz CPU. Therefore, it is concluded that the experiment can always be carried out for real computers only with that configuration and with similar results.

8.9 Summary

In this chapter, two approaches have been proposed to solve DisFCSP. In the first approach, the iterative technique in centralized FCSP is extended and used in distributed environments. The resulted FABT algorithm can solve DisFCSPs using successive ABT searches and the *nogood explanation* indexing technique described in Chapter 7.

In the second approach, an adaptation of the ADOPT algorithm is proposed. ADOPT needs to be modified to avoid message flooding. Two improved versions of ADOPT: ADOPT-am and ADOPT-fm were proposed. ADOPT-am balances messages by synchronizing the number of completed cycles between agents. It is simpler than ADOPT-fm. However, ADOPT-am is affected by the longest delay between neighboring agents and cannot effectively exploit network resources to the full extent. ADOPT-fm is the second improved ADOPT version which cuts off duplicated messages, meters and controls the rate of message sending. Experiments were carried out and show that both ADOPT-am and ADOPT-fm can solve larger problems where ADOPT fails.

A comparison between FABT, ADOPT-am, and ADOPT-fm shows that FABT has shorter completion time than ADOPT-fm, which is, in turn, better than ADOPT-am for QoS management problems with five different levels of agents' preferences.

Chapter 9

Verification in untrusted environments*

9.1 Introduction

Different DisCSP/DynDisCSP algorithms for QoS management have been proposed and discussed in the previous few chapters of this thesis. In the algorithms, an assumption which has been made so far is that agents are collaborative and trustworthy. This assumption is questionable in the Web service environment where management agents are from different companies and businesses. Some of the agents might even be corrupted or defect during the solving process. Such agents can cause incorrect execution of a DisCSP/DynDisCSP algorithm. Clearly, in a distributed and competitive environment like that of Web services, maintaining business privacy and detecting untrustworthy partners, are important for every organization. Whilst DisCSP algorithms successfully address the privacy problem, they leave the untrustworthy issue unsolved.

In this chapter, a novel verification mechanism is proposed for ABT [Yok95], and AAS [SSF00]. This explicit verification mechanism eliminates the impractical assumption of fully collaborative agents (i.e. providers) in DisCSP and hence enables our DisCSP based QoS management to be used practically with ABT, AAS, and their extensions in

*A preliminary version of the work presented in this chapter has been published in [NK06].

the real Web service environment, where full collaboration between providers may not be always guaranteed. The chapter is organized as follows: the difference between the tasks of maitaining agents' privacy and detecting untrustworthy agents is discussed in the following section. Section 9.3 describes the proposed verfication mechanism and presents a proof to confirm the correctness of the mechanism. Section 9.4 discusses how the verification mechanism can be further extended for future work. Section 9.5 concludes the chapter.

9.2 Related work

Privacy has been the only focus of DisCSP algorithms so far. In an ideal case, agents in DisCSP solve a satisfaction problem distributively whilst maintaining their privacy except for private information revealed in the final solution[1]. Achieving this is a difficult task. It has attracted a number of studies which use cryptography techniques. Secure multiparty computation is employed in [M.C03] where the authors propose using the properties of Paillier cryptosystem to secure the exchanged data. However, it is a requirement that at least 1/2 of the participants are trustworthy for an absolute guarantee of agent privacy. In other words, as long as 1/2 of the participants are not corrupted by any adversary, no further information except the final solution is revealed to the remaining set of the participants. In [YSH05], M. Yokoo et al. proposed another method with trusted servers in order to maintain the agents' privacy.

In general, the primary concern of these studies on privacy is information leaked by a DisCSP algorithm during its run. In contrast, we propose a verification mechanism which can be used at runtime to detect whether an agent follows the algorithm specification. If the agent does not, we can enforce some penalty on the defective agent, replace it, and restart the solving process. To our knowledge, there have not been any previous studies on this problem to date.

[1] A solution reveals to an agent that other agents can satisfy the values in the solution. This is one form of privacy leak.

9.3 Verification network and verification mechanism

For a DisCSP algorithm, an agent does not conform to the algorithm if it processes messages incorrectly according to the algorithm specification. Since DisCSP algorithms are often specified as a set of message processing procedures, a verification mechanism must be able to verify the correctness the execution of these procedures inside each agent. Here we argue that such a *strict* verification of an agent's internal execution is unnecessary from other agents' points of views. Agents in general, are interested in verification mechanisms which can ensure that their final search results (i.e. the final values of variables that the agents are interested in) cannot be manipulated by other agents. We introduce here a notion of *weak conformance* to address these. An agent is said to weakly conform to the AAS algorithm if it *appears* to operate correctly as seen by other agents. Formally, we can define *weak conformance* as follows:

Defintion [DisCSP weak conformance]: An agent A_i is said to *weakly conform* to a DisCSP algorithm specification *if there exists an agent $A_i^{'}$ which has been known to strictly follow the DisCSP algorithm specification* and it has the same settings (i.e. variables, constraints, etc...) as A_i. Also, a replacement of A_i by $A_i^{'}$ in a solving process *would produce the same input/output messages* after every processing cycle and hence does not change the final results for other participating agents.

9.3.1 Monitoring Overview and Initial Setup

As presented in the previous sections, an agent may act differently from the specification of an algorithm for some reasons. Instead of making the impractical assumption of fully collaborative agents, this section proposes a distributed monitoring system which can check whether an agent weakly conforms to the protocol specification. However, to ensure privacy, agents' constraints and domain values should not be revealed to the monitoring system. Here we propose such a monitoring system based on the following assumption:

- The monitoring system knows about the priority arrangement among agents and their neighboring relationships.

- The monitoring system is able to "sniff" messages exchanged between agents. That is, it can analyze and read different fields of a message but not the values of variables in the messages.

The overall idea of the proposed monitoring system is to capture every incoming message of an agent A_i and simulate A_i's execution (by following ABT/AAS specification) for this input message. The simulation outputs of messages are compared against A_i output to detect any inconsistency. However, instead of operating directly on the variable domains of A_i, the simulator operates on the encrypted values of those domains. This helps A_i to protect its private information from the monitoring system. Also, it is important to note that the simulator does not attempt to search for any solution during its execution, but to verifiy the correctness of such a solution reported by A_i. We note that DisCSP(s) are NP-complete and hence have the property that solutions to problems are quick to check, yet current methods to find exact solutions are not scalable. Therefore, the verification of a CSP solution in general is simpler and requires far less resources as compared to a solving process.

Before proceeding into a detailed discussion, we define *local solutions* and *initial valid solution set* as follows:

Definition [Local solution]: A local solution of an agent A_i is an assignment Γ that has the assignment for every variable that A_i is interested in. Γ must also be consistent with A_i's local constraints, agent-view, and the current *nogood* list.

For ABT and AAS, an agent searches for a new assignment if the agent detects that its agent view is inconistent. This assignment is a local solution. A local solution is temporary since the *nogood* set is continuously updated after each cycle.

Definition [Intial valid solution set]: An initial valid solution set of an agent A_i, defined as $S(A_i)$, is the set of A_i's local solutions before any communications, i.e. when A_i's view and nogood list are empty.

In contrast to local solutions, an initial valid solution set only changes when new variables are added through *add-link* messages. We note that instead of specifying the agent's constraints and its variable domains, we can use the agent's initial valid solution

set for a DisCSP search, since the same information is presented.

The setting of the monitoring network is as follows. For a DisCSP network of n solving agents $A_i, i = 1..n$, the monitoring system consists of n monitoring agents $M_i, i = 1..n$. The monitoring agent M_i is installed next to A_i and can sniff messages sent to and from A_i. ABT/AAS verification and solving processes are executed in parallel. Initially, agents are assumed to have the following knowledge:

- Every solving agent A_i shares with its neighbors a secret key k. This key is used to encrypt the values proposed in *ok?* message and assignments of *nogood* messages generated by the agent.
- Every solving agent A_i enumerates and encrypts values in its *initial valid solution set* $\mathrm{S}(\mathrm{A}_i)$ (see Definition 6) using the above secret key and then the agent's public key p_{Ai}. These encrypted values are sent to the monitoring agent M_i and are used by M_i as its initial solution set $\mathrm{S}(\mathrm{M}_i)$.

To better explain the above relationship between $\mathrm{S}(A_i)$ and $\mathrm{S}(\mathrm{M}_i)$, we assume that the set $\mathrm{S}(A_i)$ is represented as:

$$\mathrm{S}(A_i)=\{\left\langle x_{i_1} = S^p_{i_1}, ..., x_{i_k} = S^p_{i_k}\right\rangle : p = 1..m\}$$

in which $x_{i_1}, ..., x_{i_k}$ are variables of A_i and $S^p_{i_j}$ is a set of values for x_{i_j}. The set $\mathrm{S}(\mathrm{M}_i)$, which is also the M_i's valid solution set, can be obtained as:

$$S(M_i) = \{\left\langle x_{i_1} = f(S^p_{i_1}), ..., x_{i_k} = f(S^p_{i_k})\right\rangle : p = 1..m\} \tag{9.1}$$

The function f above is used for encryption purpose, and is defined as:

$$f(S^p_{i_j}) = \{f(s) : \forall s \in S^p_{i_j}\}, j = 1..k \tag{9.2}$$

$$f(s) = p_{Ai}(k(s)) \tag{9.3}$$

For example, if A_i has two variables x_1 and x_2 and

$$S(A_i) = \{\langle x_1 = \{1,4\}, x_2 = \{1\}\rangle, \langle x_1 = \{3\}, x_2 = \{3\}\rangle\}$$

then

$$S(M_i) = \{\langle x_1 = \{f(1), f(4)\}, x_2 = \{f(1)\}\rangle, \langle x_1 = \{f(3)\}, x_2 = \{f(3)\}\rangle\}$$

During the solving process, M_i operates similarly to A_i but on the encrypted domains of A_i's variables. This is to avoid privacy leaks from A_i to M_i. Messages generated by M_i are then used to compare against A_i to detect any discrepancies (i.e. disconformance). We can see that because the values in the initial solution set are encrypted when presenting to M_i, M_i knows neither A_i's valid solutions nor its constraints. Also the monitoring agent cannot know the proposed values in the sniffed messages because it does not know the secret keys shared by A_i and its neighbors. Only the cardinality (i.e. number of elements) of the valid solution set of the solving agent A_i, can be deduced by the monitoring system.

9.3.2 Monitoring Algorithms

As discussed previously, each monitoring agent M_i executes a similar process to A_i's. This section presents a detailed monitoring algorithm for AAS. A similar algorithm can be constructed for ABT. For reference purposes, important procedures of AAS are shown in Algorithms 13 and 14. Corresponding procedures of AAS verification are shown in Algorithms 15 and 16. In particular, the *verify_message_processing* procedure in Algorithm 15 for M_i is similar to the *message_processing* procedure in Algorithm 13 for A_i. From Line 2 to Line 12, the *verify_message_processing* procedure resembles *message_processing*. However, the *verify_message_processing* procedure has additional operations to capture A_i's incoming and outgoing messages in $m_{in}^{encrypted}$ and $\mathbf{m}_{out}^{encrypted}$ at Line 1. It later compares those messages against its generated messages of $\mathbf{m}_{out}^{self}$ at Line 13. Also, the procedure invokes the *verify_check_agent_view*, instead of the *check_agent_view*.

The difference in executions between A_i and M_i in the *check_agent_view* (Algorithms 13) and *verify_check_agent_view* (Algorithms 15) procedures can be explained as follows: Algorithm 15 shows that instead of computing a new local solution at M_i (as in Line 3 of *check_agent_view*-Algorithm 13), the verification requires that A_i encrypts and reports its

satisfied assignment to M_i. This encrypted assignment, as proved later in Proposition 1, is a local solution of M_i. This difference in *verify_check_agent_view* and *check_agent_view* is important for the following reasons:

- For local constraints, it is easier to verify a solution than to find one. M_i does not need to search for a new solution but to verify a solution found by A_i. This verification requires less processing resources as compared to a solving process.
- If A_i reports no solution, there is also less processing required for M_i to verify this by using the minimal *nogood* set sent out by A_i.
- There are many possible valid assignments found in Line 3 of the *check_agent_view*. If A_i does not report its assignment to M_i, there is no guarantee that there is a correspondence between their assignments.

Algorithm 13: AAS_message_processing(m_{in})

```
1  if m_in is an ok message then
2      update agent_view;
3      check_agent_view;
4  else
5      if m_in is a nogood message then
6          update agent_view;
7          if consequence of the nogood in m_in is not covered by other nogood then
8              send addlink messages to owners of variables which are not connected
               with this agent;
9              add the nogood into the nogood list;
10     check_agent_view;
11     resend an ok? message to the nogood sender if the assignment for this sender is
       unchanged ;
```

Because of the similarity between *verify_processing_message* and *processing_message* procedures, the following two properties are guaranteed:

Property 1: The agent view $V(M_i)$ maintained by M_i is an image of $V(A_i)$, the agent view of A_i, under the function f. In other words:

$$\forall v_a \in V(A_i), v_a = \left\langle x_{i_1} = S^p_{i_1}, ..., x_{i_k} = S^p_{i_k} \right\rangle \text{ then}$$

Algorithm 14: AAS_check_agent_view(m_{in})

```
1  if current_view is consistent with local constraints and nogoods then
2      if no aggregate in variable domains is consistent with local constrains and
       nogoods then
3          backtrack;
4      else
5          find an aggregate which is consistent with local constraints and nogoods;
6          update agent_view with the aggregate;
7          send ok? messages to lower priority agents;
```

Algorithm 15: AAS verify_message_processing(m_{in})

```
1  record A_i's incoming message m_in^encrypted and outgoing message set m_out^encrypted;
2  if m_in is an ok message then
3      update agent_view;
4      verify_check_agent_view;
5  else
6      if m_in is a nogood message then
7          update agent_view;
8          if consequence of the nogood in m_in is not covered by other nogood then
9              generate addlink messages for owners of variables which are not
               connected with this agent and add these messages to m_out^self;
10             add the nogood into the nogood list;
11     verify_check_agent_view;
12     generate ok messages for any repeated assignments (from the previous local
       solution) with new histories and add these messages to m_out^self
13 if m_out^self is inconsistent with m_out^encrypted then
14     report AAS violation;
15 if m_out^self is inconsistent with A_i's reported solution then
16     report AAS violation;
```

Algorithm 16: AAS *verify_check_agent_view*

```
1  if A_i reports no solution then
2      if find an assignment which is consistent with local constraints and nogoods
       then
3          report AAS violation;
4      backtrack;
5  else
6      eIfA_i reports a solution  encrypt the reported solution with A_i's public key if
       the solution is inconsistent with local constraints and nogoods then
7          report AAS violation;
8      update the agent_view with the solution;
9      generate ok messages for lower priority agents and add them to m_out^self;
10     report AAS violation;
```

$$\exists v_m \in V(M_i), v_m = \left\langle x_{i_1} = f(S^p_{i_1}), ..., x_{i_k} = f(S^p_{i_k}) \right\rangle$$

And vice versa:

$$\forall v_m \in V(M_i), v_m = \left\langle x_{i_1} = S^p_{i_1}, ..., x_{i_k} = S^p_{i_k} \right\rangle \text{ then}$$
$$\exists v_a \in V(A_i), v_a = \left\langle x_{i_1} = f^{-1}(S^p_{i_1}), ..., x_{i_k} = f^{-1}(S^p_{i_k}) \right\rangle$$

Property 2: The assignment sets in N(A_i), which is the *nogood* set of A_i, is an image of N(M_i) under f.

From the above properties and the initial setting that M_i's initial valid solution set is an image of A_i's under the function f, we have the following proposition:

Proposition 9.1 [Correctness of monitoring solutions]: For any processing cycle, $f(\Gamma)$ is a local solution of M_i iff Γ is a local solution of A_i

Proof: We note that A_i and M_i determine whether an assignment is a local solution in the same way that it is based solely on their agent views, *nogood* sets and valid solution sets. We say an assignment Γ_1 *fully contains* another assignment Γ_2 if for every variable, the value set of this variable in Γ_2 is a subset of that in Γ_1. We denote this relationship as $\Gamma_1 \widetilde{\supset} \Gamma_2$. An assignment Γ, which has the same set of variables as A_i does, is a local

solution of A_i iff:

$$V(A_i)\widetilde{\supset}\Gamma \tag{9.4}$$

$$\exists s \in S(A_i) : s\widetilde{\supset}\Gamma \tag{9.5}$$

$$\exists! n \in N(A_i) : n\widetilde{\supset}\Gamma \tag{9.6}$$

The condition (9.4) says that Γ must be consistent with the agent view, (9.5) states that Γ must be consistent with A_i's local constraints, and (9.6) confirms that no *nogoods*' assignment can be induced from this assignment. Because these conditions can be checked by matching of the values, they are preserved after f transformation. For this reason and Properties 1 and 2, we only need to prove that M_i's valid solution set is an image of A_i's under f. Initially this is true. These solution sets are only changed after *addlink* operations. If any new variable is added to A_i through an *addlink* operation, the variable is also added to M_i with its corresponding encrypted domain. Since A_i's constraints do not restrict the values of this new variable, the Cartesian product of the existing valid solution set and the new variable's domain, form a new valid solution set for A_i and M_i. This Cartesian product preserves the f relationship, therefore M_i's valid solution set is maintained as an image of A_i's after any *addlink* operation. ■

The following proposition ensures that our monitoring system can always detect any non-compliant behavior of service providers.

Proposition 9.2 [Guarantee of ABT/AAS conformance]: If A_i does not *weakly conform* to an AAS algorithm then M_i is able to detect this by using Algorithms 15 and 16.

Proof: We consider two different scenarios. In the first scenario, suppose that A_i correctly reports its local solution to M_i. It can be seen that the correctness of A_i's local solving process is then ensured in Algorithm 16 (Lines 1 to 7), in which M_i can verify A_i's local solution and the existence of such a solution according to Proposition 1. In addition, because M_i essentially executes the same deterministic procedure as A_i, if we encrypt all

values in every assignment in the output messages of A_i, we must be able to get the output messages of M_i. Therefore Algorithm 15 (Line 4) can detect if the input/output of A_i does not *strictly conform* to ABT/AAS.

In the second scenario, suppose that A_i incorrectly reports its local solution. According to Proposition 1, this solution must be a valid solution otherwise a violation is detected. However, this solution is not the same as the one A_i found. Now, if an agent $A_i^{'}$ processes messages in the same way as M_i does, but using decrypted values in all of its messages, then $A_i^{'}$ strictly conforms to AAS specification. In this case, either no difference between input/outputs of $A_i^{'}$ and A_i is found or a conformance violation is detected. Therefore the proposition is proved. Also, Lines 13 and 14 of Algorithm 15 disallow A_i from incorrectly reporting the assignments of shared variables in its local solution. Therefore A_i can only report wrongly the values of its own non-shared variables. This also confirms again that A_i cannot change the result of other variables' values which are of other agents' interests. ■

9.4 Discussion

It is worthy to note that the presented verification mechanism assumes every agent to be honest when passing the encyption of its initial solution set (or equivalently its encrypted local constraints) to the monitoring agents. This assumption can be made more practical if severe penalties are applied for any agent which cannot carry out the management according to the solution. This enforces an agent to report its initial solution set correctly since the agent has no idea about the final solution at the beginning. In addition, it is worthy to note that:

- The agent does not wish to miss any initial solution in the set that it reports to its monitoring agent. By doing this a valid solution is discarded and possibly over-constrain is detected.

- The agent does not wish to include any invalid initial solution into the set that it reports to its monitoring agent. By doing this the invalid solution can be a part in

the final solution that the agent cannot satisfy and have to pay a high penalty for.

Since the verification mechanism is desgined specifically for AAS and ABT – two DisCSP algorithms, further investigations are required for DynDisCSP algorithms such as ABTRestart and DynABT presented in Chapter 7. In general, verification in a dynamic environment is difficult unless the verification system can distinguish between real changes and false requests of changes made by malicious agents. This ability to distinguish is a very difficult to achieve as it requires the verification system to have the inside knowledge of every agent's organization and its surrounding environment. Nevetheless, one possible direction to solve this problem is to classify changes into being verifiable and non-verifiable. A change is verifiable if it can be verified by external process using some means; otherwise, it is non-verifiable. Agents can only notify and propage verifiable changes and should be responsible for non-verifiable changes. In other words, agents only participate in finding a new solution if some verifiable changes happen at an agent. However, they do not respond to non-verifiable changes. Using these concepts of verifiable and non-verifiable changes as well as developing new algorithms based on these concepts are work we propose to undertake in the future.

9.5 Summary

This chapter has proposed a verification mechanism for ABT and AAS algorithms. The verification mechanism requires only one trusted organization of the monitoring agents. This assumption of a single trusted organization is reasonable in practice. For example, it is equivalent to the assumption of a trusted public key verifier like Verisign [Ver06] in network security. The idea in this verfication mechanism can also be applied to ABT as well as its extensions. It can thus be used for the management framework described in Chapter 4 of this thesis. Future work will investigate and extend the verification so that it can be used for DynABT and other DynDisCSP algorithms in general.

Chapter 10

WS2JADE - Web service and FIPA agent integration*

10.1 Introduction

The framework for DisCSP based QoS management proposed in this thesis consists of different parts which can be grouped into three main components as shown in Figure 10.1:

Distributed Agents: The brain of our management system. These agents are individually reactive but can be collectively intelligent. The agents use DisCSP and DCOP algorithms to manage the QoS of Web services.

Protocol Verifier: This internal component verifies whether all agents conform to the DisCSP algorithms that they use. It is important to have this component since agents are from different organizations and they may not be trustful to each other. An agent, in order to gain some advantage, might not conform to an algorithm that it is supposed to follow.

Communication Adaptor: If the local resource manager requires proprietary interaction mechanisms then a communication adaptor is required. In our framework, we require a local resource manager to expose its manageability interfaces as Web services. This can be done using OASIS Web Service WSDM [Hew04b; Hew04a]- one of the key

*Preliminary versions of the work presented in this chapter have been published in [NK05b] and [NK05a].

standards on distributed management. The communication adaptor can enable the communication between the management agents and the manageability Web services.

As can be seen from Figure 10.1, distributed agents play an active role in the management process. Distributed agents can be implemented as asynchronous Web services, agents in Multi-Agent Systems (MAS), or any peer to peer software. In this chapter, we explore the possibility that distributed agents are agents in MAS. Whilst MAS is different technology to Web services, using MAS brings certain advantages. The focus of MAS is on distributed reasoning and intelligence. This focus is complimentary to the goal of Web services on system interoperability. M.N. Huhns et al. has pointed out many benefits when using multiagent systems for successful service-oriented computing [HSB+05]. Using MAS technology in our management system brings the following specific benefits:

- In MAS, every agent is an autonomous entity with an individual goal. Our management agent is naturally equivalent to an agent in MAS, where the agent's goal is to manage the QoS of all compositions that its organization takes part in.

- Cooperation and coordination for distributed problem solving is one of the topics of MAS [NJE03]. We can use existing techniques and approaches from this field for building our management system.

- Agents in MAS use ontology to describe concepts. There have been many studies (e.g. [D'A06; TRPA06]) on ontology representation and translation including work for QoS concepts. Our management system can use these studies for achieving a common understanding between different QoS concepts defined by different organizations.

- There is a growing number of studies on SLA negotiation and workflow management using MAS (e.g. [JUY04, STW05]). The work in these studies can be integrated into our system if MAS technology is adopted.

In order to provide a practical solution, and utilize the existing MAS implementations for our management agents, there is a need to interoperably connect, especially FIPA compliant MAS and Web services together. Figure 10.1 illustrates the coexistence of both

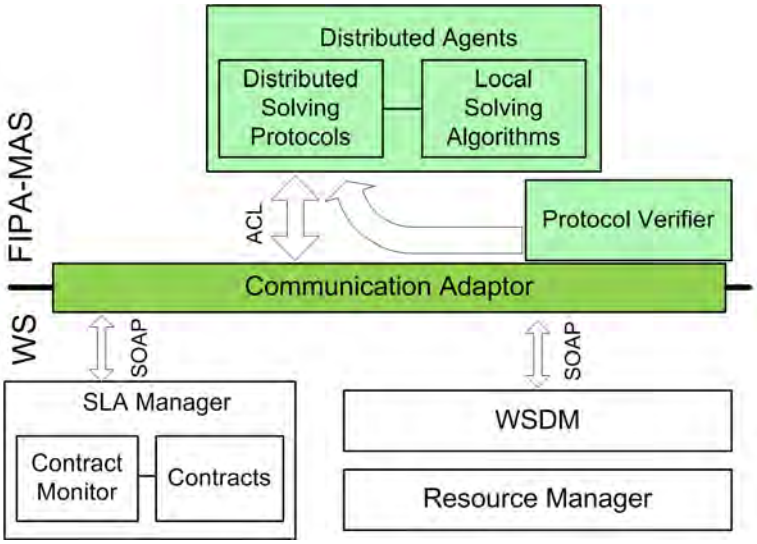


Figure 10.1: Different supported components inside the organization of a management agent.

agents and Web services in the same management system where the agents and Web services need to communicate with each other.

This chapter proposes a solution for FIPA compliant agents and Web services integration problem, called WS2JADE. WS2JADE has been developed for seamless integration of agents and Web services. With the support of WS2JADE, the management framework can integrate with WSDM [Hew04a; Hew04b]; in particular, the tool can enable an agent to manage a group of Web services in terms of execution and resource allocation. WS2JADE is a specific implementation technically dealing with mismatch between communications of JADE agents and communications of Web services. The chapter is organized as follows: Related work on FIPA compliant agents and Web services is presented in the following section. Section 10.3 describes WS2JADE in detail. Section 10.4 briefly summarizes some important remarks concerning on this toolkit. Section 10.5 gives an example of WS2JADE in action. Finally Section 10.6 concludes the chapter.

10.2 Related work

In the area of theoretically-related work, a symmetric integration of Web services and FIPA-compliant agent platforms has been proposed in [Age05] as a high-level architectural recommendation from the AgentCities. The reason for this symmetry is that Web services were developed without the concept of agents (i.e. FIPA agents) and can exist without agents. The symmetric architecture takes into consideration that many Web Service clients, though they may have autonomous characteristics of agents, do not conform to the FIPA specifications. A proxy-based approach allows the two platforms to be evolved in parallel without imposing any restrictions. This approach accepts the equity between the roles of agents and Web services, which is different to the traditional view that agents are considered one level up from Web services, and agents take solely the roles of Web services providers and consumers. As can be seen from Figure 10.2 taken from [Age05], a

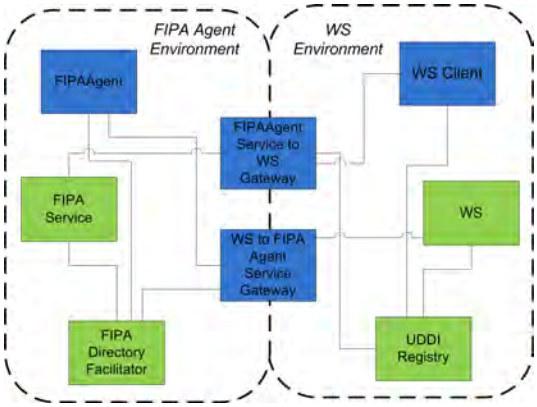


Figure 10.2: Agents and Web services can communicate through language translation gateways [Age05].

FIPA agent service environment exists in parallel with a Web service environment. The *FIPA Agent Service to Web service Gateway* on the border between the two environments allows FIPA agents to access Web services by translating ACL messages to Web service invocations. In the reverse order, the *Web Service to FIPA Agent Gateway* exposes and registers agent services in UDDI Registry Server so that any Web service client can use them. Following AgentCities recommendations, two separate pioneer implementations

have been proposed to solve two ends of the problems for a FIPA compliant Jade Agent system: Exposure of Web services to Jade agents by Sztaki [VH03] and exposure of Jade agent services to Web services by Whitestein Technology [Whi05]. Whitestein Technology has released their tool, WSAI (Web service Agent Integration), as an open source code in its first version. Another tool from Whitestein, WSIGS (Web Service Integration Gateway Service), is under development and its architecture has been published in [GC04a; GC04b]. WSDL2JADE has been released as an online program that converts WSDL file to Jade classes. It takes an input of a Web service address and generates outputs of Jade agent code and agent ontology for the Web service. It has no run-time deployment capability. Based on some test cases we have carried out with Sztaki's WSDL2JADE, there are also some problems with that tool. A sample of an input WSDL test file can be found at the WSDL2JADE homepage [IAM05]. For example, some XML enumeration values and type information are missing in the generated ontology files with XML Enumeration data type. This prevents client agents from using operations related to this data type. WSAI and WSIGS have been proposed by Whitestein Technology [Whi05; GC04a]. WSAI [Whi05] allows Web service clients to use JADE agents' services. In order to do this, WSDL files are generated for these agent services. Technically, at this stage WSDL files are created manually from these agents' behaviors. "interface agents" are also required to communicate with a target agent. These "interface agents" are created and destroyed per Web service client invocation of the agent service. However, it appears that the applicability of WSAI in practical situations is limited because of two major obstacles. Firstly, Jade agent service specification is specified loosely and not based on message levels. This makes it difficult to automatically generate WSDL interfaces. Secondly, the default single-threaded mode of Jade agents, and the asynchronous and stateful nature of agent communication do not fit well in the current implementation stage of Web service communication which mostly focuses on synchrony. There are discussions of asynchrony versus synchrony in [Whi05]. However, how to translate stateful communications of agents, in which conversations and history of past interactions with other agents are remembered, to stateless communications of Web service, is not discussed. We believe that the WSRF (WS-Resource Framework)

specification [OAS06b], where a stateless service can have stateful resources, could be applied here for a solution.

WSIGS is under development at the writing time of this thesis. WSIGS proposes an architecture for bi-directional integration with no special Agents [GC04a; GC04b]. WSIG is a set of codecs that do the translation between agents' ACL (Agent Communication Language) and Web Service calls. To be visible in both environments, WSIG is registered as a special agent service in FIPA DF (Directory Facilitator) and a special Web Service endpoint in UDDI directories. When an agent wants to invoke a service (Web service) registered in the WSIG registry, the request is passed on to the *WebServiceInvocation*, a component of WSIG, to perform the actual Web service invocation. The requirement for services in one environment to be registered by their owners in a public directory before they can be seen in other environments, reflects the assumption that Web services need to be registered before they can be discovered. This is true for a model like UDDI but in more recent P2P models the assumption is no longer valid.

10.3 WS2JADE Approach

This section describes a proposed approach for the integration of Web services and Jade agents with WS2JADE. From an architectural perspective, WS2JADE, in accordance with [Age05], forms a Web service to FIPA Agent Service gateway. There are two distinct layers in WS2JADE: an interconnecting layer and a management layer. The interconnecting layer contains interconnecting entities that glue Web Service and agents together. The management layer, being static, creates and manages those dynamic interconnecting entities. In WS2JADE, the interconnecting entities consist of special agents, ontology and protocol specifications. We call these special agents WSAG (Web Service Agents). WSAG are the agents capable of communicating with, and offering, Web Services as their own services. The combination of the static and dynamic layers is a distinct feature of WS2JADE as compared to different tools mentioned in the previous section. The WS2JADE static management layer is capable of active service discovery and automatically generating and deploying WSAG at runtime. This is an advancement over WSDL2JADE since it can au-

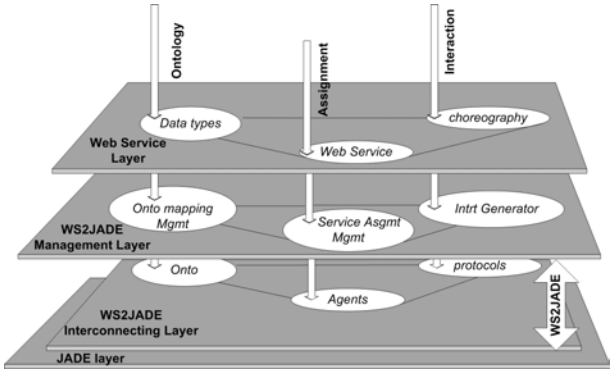


Figure 10.3: Web services are connected to JADE agents via a management layer and a communication interconnecting layer.

tomate the agent deployment process. Instead of being passive like WSIG, the WS2JADE is designed to actively discover Web services and registers them on DF if required. Also, each WSAG is multi-threaded and has its own Web service invocation module.

Figure 10.3 shows that WS2JADE management can be looked at from a different perspective as a layer which is capable of projecting Web services or any Service Oriented environment layer into the JADE agent layer. The output of this projection is artifacts which connect the Web services and agents together. As depicted in Figure 10.3, three mappings are carried out by WS2JADE during the projection: ontology mapping, interaction mapping, and assignment mapping. These mappings are handled by three main components that form the WS2JADE management layer: ontology generator and management, interaction generator and management, and service assignment management.

Figure 10.4 presents different components within the WS2JADE system and how they are linked to JADE. The vertical rectangular box is WS2JADE, the horizontal one is JADE. Note that the overlap between WS2JADE and JADE consists of the components in the WS2JADE interconnecting layer: generated interaction protocols, ontologies, and WSAGs. Figure 10.4 also illustrates a scenario for WS2JADE operation, in which a client agent searches for some service on DF. The DF can trigger WS2JADE to look up for available services in the Web service environment. If some Web services are found, their

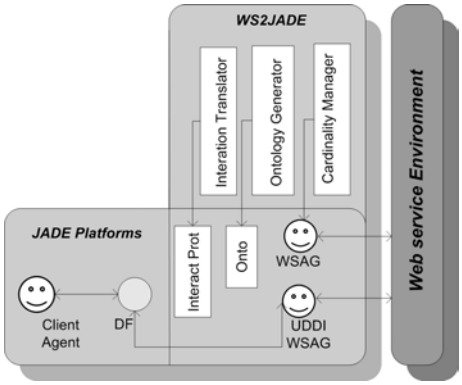


Figure 10.4: Different system components in WS2JADE.

corresponding ontology and interaction models are generated. Also, a WSAG capable of accessing the Web service is generated. This WSAG registers the Web service as its service on DF and communication between the client agent and this WSAG can start if the client agent wants the service. The following section discusses each WS2JADE component in turn.

10.3.1 Ontology Generation and Management

The ontology generator is responsible for ontology generation and ontology management. It translates data and its structure from Web service WSDL interfaces into meaningful information for Agents. A detailed explanation of a WSDL document can be found in WSDL specification [W3C05b]. WSDL describes abstract concepts and concrete entities. Abstract concepts are port type, operation, message and data type. Concrete entities are data encoding style, transport protocol and network address. In WS2JADE, the abstract concepts are relevant for the ontology mapping management as Agents need to know how to invoke the operations of a Web service. The concrete entities are handled by the interaction translator and management component. The WS2JADE ontology translator and management component converts Web services' data types and operation inputs and outputs into agent ontologies. The corresponding WSDL port type is tagged in the structure of the ontologies. According to JADE documents [BPR99], JADE ontologies can be repre-

sented as Java classes, which are convenient for Jade agent's manipulation and processing. Alternatively, they can be in other formats such as RDFS and OWL for interoperability with other FIPA compliant agent platforms. Our WS2JADE toolkit supports Jade native ontology and OWL. To generate ontologies in Java, a WSDL data type is converted to a concept in agent ontologies. Two concepts are generated for each WSDL operation. One is for the operation input message and the other is for the output message. WSDL data types can be built-in XML types. The list of built-in simple XML data types are defined in the XML schema specification. We map these built-in simple data types to Java primitives that are supported by Jade ontology representation. For XML data types that are not built-in, special customized Java classes are used, for example, Beans, Enumeration Holders and Facet classes.

Generating ontologies in the OWL format is simpler than in Java classes because OWL and WSDL both use XML. Similar to the Jade ontology generation approach, data types and messages in WSDL are mapped to concepts in OWL. There is a one-to-one relationship between concepts in ontologies generated in Java and OWL . OWL is still very new and subject to changes; however, we share the belief that it will continue to play an important role in the Semantic Web with an increasing support from agent communities.

In addition to ontology generation, ontology management is important in WS2JADE. WS2JADE organises generated ontologies in an efficient way. For data types that can be shared among different Web services, the corresponding generated ontology concepts are shared and form a common ontology base. This means that every time a new Web service is presented as an agent service, part of the existing ontology base and domain knowledge can be reused for this new service. Also, this allows the ontologies to be structured in a manageable way.

10.3.2 Interaction Mapping

The interaction mapping management component handles the conversion from Web Service communication into agent communication. Specifically, it converts Web service transport messages into ACL envelopes and Web service interaction patterns into agent protocols.

It corresponds to two sub-functionalities: language translation and interaction conversion.

Language Translation: To translate Web Service transport messages (commonly SOAP)

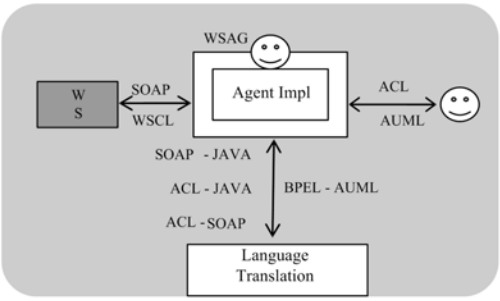


Figure 10.5: Interaction Translator and Management.

into agent ACL messages, the SOAP envelope is first projected into Java languages and then into ACL. We do not translate SOAP into ACL directly for two reasons. Firstly, we want to reuse our generated ontologies and the existing Java implementations of SOAP. Secondly, we make an assumption of the agent's intelligent capabilities to understand and process the messages according to its own logic, in addition to language translation, before forwarding the messages. This is best done by translating SOAP and ACL into Java - the native language for JADE. Figure 10.5 indicates that when a WSAG receives a SOAP message, it uses the Language Translation component to convert this message directly to ACL and send it to the client agent. It can also perform some reasoning and modification on the message by converting the message to Java classes before any translation into ACL. In the Language Translation part of the interaction management component, Axis' JAX-RPC (Java API for XML based Remote Procedure Call) implementation and JADE support for content languages and ontologies are used for translations between XML and Java, and between ACL and Java. On one hand, JAX-RPC, led by Sun, is a specification of a Web Service Invocation framework in Java. In JAX-RPC specification, at the client side, Java to XML translation in remote method call is done through a mapping from Java client stubs to the SOAP message representation. On the other hand, in JADE, information represented in Jade ontology-supported classes (Java objects) can be converted to different ACL content languages, including SL and LEAP. As can be seen from Fig-

ure 10.5, language translation is leveraged by the reuse of existing technologies instead of reinventing the wheel. SOAP-ACL translation is done by piping SOAP-JAVA and ACL-JAVA translation together. The main task of the language translation component is to map Axis stubs to Jade ontologies. However, we found that due to the restrictions of Jade ontology and JAX-RPC classes it is not easy to convert data between them. In particular, an automation of the conversion process for any data types is difficult. We use special classes which represent the ontology facets to preserve accuracy in the conversion process. There has been a similar discussion in [VH03] for Sztaki WSDL2JADE. Complex data mapping in WS2JADE (for example mapping of Axis Holder and Enumeration types to Jade ontology concepts and classes) is done recursively through simple data type.

Interaction Pattern Translation: In the Interaction Pattern Translation component, we focus on choreography. By "choreography" we mean the required patterns of interactions among parties. This is in contrast to "orchestration" that describes how a composite Web service is constructed from other atomic services. For a composite Web Service, choreography is obtained by looking from an outsider's perspective. It tells the Web service clients the different steps of how to use a composite service.

Our approach is based on mapping simple interactions implicitly described in WSDL documents into standard FIPA interaction protocols. Web service (WSDL version 1.2 [W3C05b]) provides four types of operation: one-way, request-response, solicit-response, and notification. In the one-way operation, a Web service client sends a request without receiving any response from the Web service. In the request-response, the client sends a request and receives a response synchronously. In the solicit-response, the Web service sends a solicit request to the client and receives a response. In the last type, notification, the Web service notifies the client without receiving any response. These four types of Web service operations lead to three common interaction patterns in practice: request-response, solicit-response, and subscribe-notification. The request-response and solicit-response interaction patterns correspond to those of Web service operation types. The subscribe-notification interaction pattern describes the conversation style in which a client registers to the Web service in order to receive notifications when some event occurs.

Table 10.1 summarises the mapping between these interaction styles into agent protocols. More information on FIPA Request Interaction protocol, Request Interaction protocol, and Subscribe Interaction protocol can be found in [FIP06].

WS Interaction patterns	**Agent protocols**
Request-response	FIPA Request Interaction Protocol
Solicit-response	FIPA Request Interaction Protocol
Subscribe-Notification	FIPA Subscribe Interaction Protocol

Table 10.1: Interaction pattern Mapping

10.3.3 Service Assignment Management and Service Discovery

The Service Assignment Management component is responsible for cardinal mapping and service deployment management. The cardinal mapping management manages M:N relationships between Web services and WSAG. In WS2JADE, a number of Web services can be offered as services of different WSAGs. In other words, a WSAG can offer more than one Web Service and a Web Service can be offered by more than one WSAG. This cardinal relationship is managed through a registry that keeps records as triples of a Web service, a WSAG that offers the Web service, and a new name for the Web Service in the agent platform. The service assignment management also provides a tool for deploying and destroying WSAGs, and assigns new Web services to a WSAG. It informs the WSAG which ontologies should be used for a particular newly assigned Web Service. If an assigned Web service is reported to be no longer available, the service deployment management removes the service from the list of the offered services of WSAGs and from the DF.

The Service Discovery component is designed to discover Web services. It is essentially a piece of software that can use Web service discovery protocols and translate the received information into agent service description for DF. As mentioned above, we prefer an active discovery model rather than waiting for services to be registered. At the time of writing, Web service discovery protocol is complex and subject to change with the latest revised

version of WS-Discovery specification which uses multicast protocols. The traditional Web service discovery mechanism of UDDI shares a common model with agent DF in the sense of accessing the directory. However, UDDI has evolved away from the concept of a "Universal Business Directory" that represented a master directory of publicly available services, as DF still is. Most P2P based and multicast discovery protocols prove that requesting service providers to register the services is not always the case.

10.4 Remarks

As discussed in [Age05], the main difficulties in integration of Web services and FIPA compliant agent platforms are the mismatches in communication and descriptions. These are summarized in Tables 10.2 and 10.3, respectively. For translation from a Web service to an agent service, WS2JADE handles these mismatches through its different components. Although the current version of WS2JADE is operational and offers many advantages over other tools, it can still be improved in a number of areas.

For example, to increase the ontology reuse and avoid redundancy, the semantic mapping management component can be extended to detect semantic equivalence of two syntactically different generated concepts and keeps one of them only. In [KS03], the authors focus on this topic and outline some approaches. This version of WS2JADE has not yet implemented that specific feature. Another area is the interaction pattern translations. Web

FIPA agent communication	**W3 Web Service communication**
ACL/IIOP+HTTP	SOAP/HTTP
Asynchronous	Synchronous/Asynchornous
Stateful	Stateless

Table 10.2: Communication Mismatch

Service Choreography Description Language (WS-CDL [W3C06]) has been under development for same time. WS-CDL is considered as a layer above WSDL in the Web Service technology layer hierarchy. It describes a set of rules to explain how different partners may act in a conversation. In the WS2JADE approach, we plan to convert BPEL4WS and

WS-CDL (however, not at this stage of WS-CDL development) into Agent Unified Modeling Language (AUML [BMO01]) for the overall protocol representation in UML templates. AUML is an extension of UML language for agents and has been used as a standard language to describe FIPA interaction protocols. The interaction translator and management will keep generated AUML documents in its protocol specification repository which can be looked up by client agents (or the client agents' designers) before using the service. In this version of WS2JADE we have not implemented the translation of WSCL to AUML. One reason is because of the instability and immaturity in Web Service Choreography at this time evidenced by the suppression of WSCI (Web Service Choreography Interface)/WSCL (Web Service Choreography Language) [W3C05a] by WS-CDL which is still in the first draft version.

FIPA agent service	**W3 Web Service**
Name - Name of the service	Names of services, port types, operations, etc.
Type - Type of the service	Type - Container of data type
Protocols - List of supported protocols	Choreography interfaces
Ontologies - List of supported ontologies and languages	XML Schemas - different namespaces imported
Ownership - The owner of the service	
	Binding - Protocol and data format specs for a port type
	Port - Single endpoint as combination of a binding and a port type

Table 10.3: Description Mismatches in FIPA agents and W3C Web services.

10.5 WS2JADE in Action

WS2JADE version 1.0 implements the core components shown in Figure 10.4 with the gray color. From a user perspective, WS2JADE provides two distinct tools: a tool for ontology generation and management, and a tool for deploying and controlling Web Services as agent

services. The ontology generation and management tool is offered through the combination of the ontology translator and the interaction pattern management components. The Web Service deploying and controlling tool is provided through combination of the service assignment mapping and the interaction pattern management components.

The ontology generation and management tool can be used alone from a command line with an input of a WSDL address and several user options. The options include whether the user wants to generate ontologies for the immediate imported XML types which are defined in other documents, and where to store Web Service stubs and agent ontologies. Another option is to specify the name for the agent service. The tool has been developed under the *ciamas.wsjade.wsdl2jade* package with Java main class *ciamas.wsjade.wsdl2jade.WSDL2Jade*.

In WS2JADE, the basic functionalities of a WSAG have been implemented in *ciamas.wsjade.wsdl2jade.utils.WSAgent*. The users can reuse the code or extend this base class to implement new functionalities, agent reasoning for example; since the base class supports message translation only. If the agent is designed with some complex behaviors in which the single threaded mode is preferred, the base class can then be modified so that the agent is single threaded without any difficulty. WS2JADE's Web Service deploying and

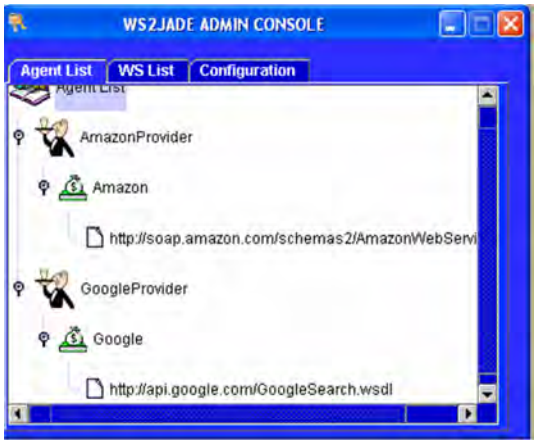


Figure 10.6: Service Deployment Interface of WS2JADE.

controlling tool allows Jade agents to deploy Web Services on the fly. The tool has been

developed in the *ciamas.wsjade.management* package with the main class for graphical administration interface at *ciamas.wsjade.management.utils.Admin*. WS2JADE operates as follows. First, important parameters in the configuration panel need to be set. The gateway agent container is then started and new agents are created from the "Agent List" tab. After deploying the new agents, new WSAG agents appear on the Jade platform ready to be assigned with Web Services. Once the Web Services are assigned they have to be activated in order to be used by client agents. The deployment process can be done simply as shown in Figure 10.6, in which "Amazon" and "Google" services are deployed with a few clicks and inputs of the real Amazon and Google Web Service addresses. The WSAG which provides these services are "AmazonProvider" and "GoogleProvider". Ontology packages are generated and compiled on the fly, and the Web Services are now available as services of these WSAG agents. The generated ontology package is stored under the "Jade output folder" and can be sent to a Web Server for downloading and using by the client agents. Also, since each WSAG is multithreaded, each of them can

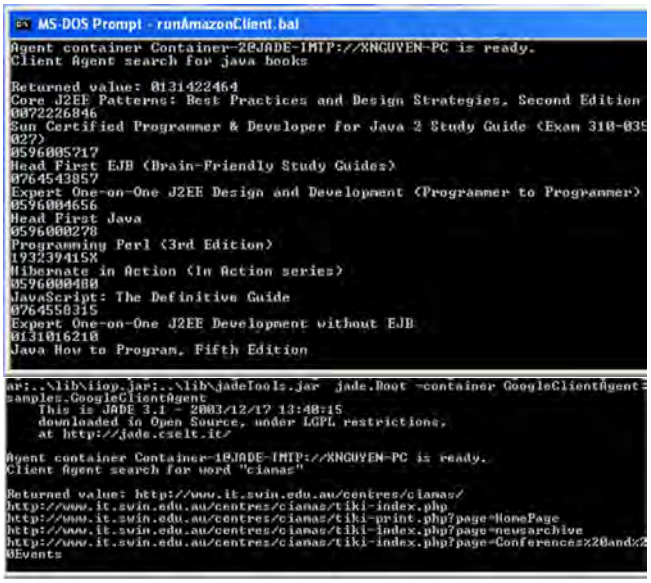


Figure 10.7: Amazon client – demonstration of WS2JADE.

serve many Web Services at the same time. In the deployment process, these services must be assigned different agent-service names. These names must be different for one WSAG

and may be the same as the real Web Service name. The "WS List" tab shows a snapshot of all deployed Web Services and generated ontologies. To test the Web Service, the user may create a client agent to use the distributed ontologies to invoke the Web Service through WS2JADE. For the Amazon and Google services in this example, a simple client sample code can be found from the WS2JADE distribution. Figure 10.7 shows the output of running sample Amazon client agent in the top window and Google client agent in the bottom window, in which, the Amazon client sends a request to search for all products with the key word "Java" under "book" category from Amazon Web Service; the Google client agent finds five most popular links for a search on keyword "ciamas". Note that in order to compile and use these sample client agents, developer keys are required to access the Amazon and Google Web Services. These keys can be obtained for free after registration from Amazon and Google Web Service websites [Goo05; Ama05].

The WS2JADE version 1.0 is freely available for download from *http://www.it.swin.edu.au/centres/ciamas*. It can be used as a supporting component in the proposed framework. Specifically, agents in WS2JADE can communicate with the managed Web services. In addition, existing implementations of different algorithms (e.g. DynABT, FABT, ADOPT-*)[1] proposed in this thesis, can be used with WS2JADE agents. This enables WS2JADE agents to collaborate and manage the Web services together.

10.6 Summary

This chapter has proposed an approach to integrate FIPA compliant agents and Web services so that the agents can communicate with the Web services. A detailed description of WS2JADE – the integration toolkit followed in the approach has also been presented, together with a demonstration of its capabilities. WS2JADE is a useful software for the management framework proposed in Chapter 4 of this thesis. Any implementation of the framework can reuse WS2JADE for interconnecting agents at the management layer to the Web services at the infrastructure layer. Such an implementation brings certain advantages in reusing existing work from the MAS field. The current version of WS2JADE focuses

[1]The implementation of these algorithms in the SIM-MAS toolkit are available from the same Web site

on 1-way communications from the agents to the Web services. In the future, this will be extended to bi-directional communications between them.

Chapter 11

Conclusion and future work

The quality of service (QoS) guarantee is a critical step in the application and management of Web service compositions. The QoS guarantee is a difficult task due to the distributed and dynamic nature of Web service environments. QoS management systems for Web services should be able to distributively manage the Web services as well as to handle dynamic changes and uncertainties arising in the Web service environments. Adding to these complexities, different composite Web services may have inter-dependencies in their underlying resources and hence need to be managed together. This thesis has contributed to the development of a management framework to automate the QoS management process using Distributed Constraint Satisfaction (DisCSP) algorithms. Existing DisCSP algorithms have been adapted and new ones are proposed for the framework. The algorithms can improve the effectiveness and efficiency of the QoS management. In this chapter, the key contributions of the thesis are summarised in Section 11.1, followed by directions for future research in Section 11.2.

11.1 Key contributions

This thesis makes the following key contributions to research into QoS management for Web services:

- A new *distributed framework* for QoS management of multiple related service com-

positions is proposed. In this framework, agents are distributed management entities from different organizations. These agents view the management tasks as instances of DisCSPs/DynDisCSPs. The agents communicate and collaborate to collectively manage the QoS of Web services using some DisCSP/DynDisCSP algorithms. Importantly, advanced AI techniques and agent technologies can be integrated into the framework. Implementations of the framework can be supported by WS2JADE – a toolkit for FIPA compliant agents and Web services integration.

- *Extensions and improvements to existing distributed constraint satisfaction algorithms* are proposed. This enables existing DisCSP algorithms such as ABT and AAS to be used practically in the framework. In the adaptations and improvements, existing DisCSP algorithms are reviewed. Their possible limitations when being used in the Web service environments are analysed and addressed adequately. The limitations include the assumption that message sending and receiving are FIFO, and no message is lost in transit for DisCSP applications. Moreover, a heuristic that exploits the monotonic characteristic of agents' preferences for QoS parameters is proposed to improve the performance of ABT and AAS. Experiments showed that this heuristic significantly cuts down the completion time by 40% for ABT and AAS for DisCSPs with medium tightness.

- *New distributed constraint satisfaction algorithms*, most importantly the Dynamic Asynchronous Backtracking Search (DynABT) and Asynchronous Backtracking Search for Fuzzy Constraints (FABT), are proposed for the QoS management. DynABT and FABT are complete algorithms. Using DynABT, Web services can be dynamically added into or removed from the QoS management scope. Using FABT, the quality of a management solution can be maximized according to agents' preferences. In addition, the existing DCOP algorithm – ADOPT, is also investigated and adapted into ADOPT-* for the same purpose of maximizing the agents' preferences. Experiments showed that FABT has better performance than ADOPT in terms of completion time for DisFCSPs with five levels of preferences.

- *A novel verification mechanism* is proposed to check the conformance of agents to the specification of a DisCSP algorithm (ABT or AAS) at runtime. This mechanism allows the algorithm to be used practically in an untrusted environment such as that of Web services. In the Web service environment, agents represent different organizations and have different interests. An agent may execute the DisCSP algorithm incorrectly if this brings the agent some own benefits.

11.2 Future directions

This thesis has proposed the DisCSP techniques for QoS management and made significant inroads into understanding and developing solutions in this direction. Despite the initial success and effort spent on solving the multi-composition management problem, there are challenges ahead. In what follows we discuss important directions for future work.

- *Improving constraint formation/representation:* Constraint formation/representation in Chapter 5 is the process of collecting/forming QoS related constraints inside different ASs in the proposed framework. The current work in Chapter 5 focuses only on constraints formed by the Web service composition structure. Further investigations are needed for other types of constraints. Moreover, a more comprehensive investigation into "vertical or "bottom-up" mapping formulas which translate values of QoS parameters at the Web service layers onto lower resource layers is required.

- *Verification system for dynamic algorithms:* One shortcoming of this thesis is the limited usage of the proposed verification mechanism in dynamic environments. The verification is proposed only for ABT and AAS – static DisCSP algorithms. Further work is required to extend the verification for DynDisCSP algorithms, i.e. algorithms which operate in dynamic environments. As discussed in Chapter 11, one possible direction in extending the verification is to classify changes into verifiable changes and non-verifiable changes. A change is verifiable if it can be verified externally by a third party. Otherwise, it is non-verifiable. Agents can only notify and propagate verifiable changes and should be responsible for non-verifiable changes. If the monitoring

system proposed in Chapter 11 has the authority to verify all verifiable changes, then it can potentially be used to verify the conformance of agents in executing a DynDisCSP algorithm.

- *Improvement in performance for DisCSP algorithms:* Improving the performance of distributed algorithms, especially algorithms which can handle multi-variables per agent, is crucial for the management framework. This is also important for the field of distributed constraint satisfaction in general. To achieve this, more efficient consistency propagation techniques can be incorporated into the search to speed up its completion time and lower the storage complexity. Moreover, agents can trade some of their privacy for a shorter completion time. In particular, the agents can reveal some of their less important constraints to others, and hence speed up the solving process. It would be an interesting direction to take in examining how these techniques can be transferred to solving our QoS management problem in a practical way.

Appendix A

ABT Algorithm

Algorithm 17: ABT Protocol

```
1  procedure abt(mesg) do
2      if mesg is ok? message then
3          processOKMsg(mesg);
4      else if mesg is nogood message then
5          processNogoodMsg(mesg);
6      else if mesg is addlink message then
7          processAddlinkMsg(mesg);
8  procedure processOKMsg(mesg) do
9      get the assignment (x_j, d_j) from mesg;
10     add (x_j, d_j) to agent_view;
11 procedure processNogoodMsg(mesg) do
12     get the nogood from mesg;
13     foreach (x_k, d_k) where x_k is not connected and is contained in nogood do
14         send Addlink to x_k to add a link from x_k to x_i;
15         add (x_k, d_k) to agent_view;
16     current_value = old_value;
17     check_agent_view;
18     if old_value == current_value then
19         send(ok?, (x_j, current_value)) to x_j;
20 procedure processAddlinkMsg(mesg) do
21     send ok? of my assignment to the sender of mesg;
```

Algorithm 18: ABT Protocol

```
1  procedure check_agent_view do
2      if agent_view and current_value are not consistent then
3          if no value in D_i is consistent with agent_view then
4              back_track;
5          else
6              find a new value d so d and agent_view are consistent;
7              current_value=d;
8              send (ok?,(x_i, d)) to outgoing links;
9  procedure backtrack do
10     nogoods = {V: V is an inconsistent subset of agent_view};
11     if ∅ ∈ nogoods then
12         broadcast overconstraint to all agents;
13         terminate the algorithm ;
14     foreach V ∈ nogoods do
15         select (x_j, d_j) where x_j has the lowest priority in V;
16         send(nogood, x_i, V) to x_j;
17     check_agent_view;
```

Appendix B

ABT With Termination Detection Algorithm

Algorithm 19: ABT with Termination Detection Protocol

```
1  procedure abt_with_ter_detect(mesg) do
2      if mesg is known to ABT then
3          abt(mesg);
4      else
5          if mesg is a good message then
6              processGoodMsg(mesg);
7  procedure processNoGoodMsg(mesg) do
8      if nogood assiNoment is my current solution then
9          clear partial_sol;
10         nogood_reported = true;
```

Algorithm 20: ABT with Termination Detection Protocol

```
1  procedure processGoodMsg(mesg) do
2      if nogood_reported is true then
3          return;
4      else
5          if good assignment is my current solution then
6              update and aggregate this good assignment into the partial_sol;
7              if the subtree is solved then
8                  send partial_sol in good messages up to higher prio neighbors;
9                  if this is the highest priority agent then
10                     announce a solution found;

11 procedure check_agent_view do
12     if the view is consistent and subtree is solved then
13         send up a good message to every parent;
14     else
15         if a new assignment found then
16             nogood_reported = false;
```

Appendix C

ABT Restart Algorithm

Algorithm 21: DynABTRestart Protocol

```
1  procedure abt_restart(mesg) do
2      if mesg type ∈ {ChangeAck, ChangeUpdate, Restart} then
3          process mesg using processChangeAckMessage,
           processChangeUpdateMessage, or processRestartMessage appropriately;
4      else if The set of changes embeded in mesg is the same as of this agent then
5          abt(mesg);
6  procedure processChangeAckMessage(mesg) do
7      get the changeID contained in mesg;
8      increase the number of agents responding for this changeID;
9      if all recorded changeID are responded by all agents then
10         broadcast a RestartMessage;
11 procedure processRestartMessage(mesg) do
12     clear partial solution;
13     clear nogood storage;
14     find a valid assignment asmgt for controlled variables;
15     foreach lower priority agent AID do
16         send(ok(asgmt), AID);
17 procedure processChangeUpdateMessage(mesg) do
18     if This is the coordinator and the AckedRequest field in mesg is off then
19         broadcast mesg with AckedRequest field on ;
20     get the change in mesg;
21     if changeID of change is new for this agent then
22         add the changeID of mesg into the list of known change;
23         send( ChangeAck , coordinator);
```

Algorithm 22: DynABTRestart Protocol

```
1  procedure handleChange(change) do
2      update the local CSP according to the change;
3      create a new message mesg with RequestAck field off;
4      send(mesg, coordinator);
5  procedure send(mesg, AID) do
6      embed the list of known changes into mesg;
7      send out mesg to AID;
8  procedure broadcast(mesg) do
9      embed the list of known changes into mesg;
10     broadcast mesg ;
```

Appendix D

Dynamic ABT Algorithm

Algorithm 23: DynABT Protocol

```
1  procedure dyn_abt(mesg) do
2      get the set of changes changes embeded in mesg;
3      if processChange(changes) then
4          if mesg is an ABT message then
5              abt(mesg);
6  procedure processChange(changes) do
7      changes_for_me = changes \ my_changes;
8      changes_for_neighbor = my_changes \ changes;
9      if changes_for_me != ∅ and changes_for_neighbor != ∅ then
10         update the neighbor about changes_for_neighbor;
11         update myself on changes_for_neighbor;
12         return false;
13     else if changes_for_me != ∅ and changes_for_neighbor == ∅ then
14         update myself on changes_for_neighbor;
15         return true;
16     else if changes_for_me == ∅ and changes_for_neighbor != ∅ then
17         update the neighbor about changes_for_neighbor;
18         return false;
```

Algorithm 24: DynABT Protocol

```
1  procedure updateNeighbor(AID) do
2      if AID has higher priority than this agent then
3          create a ChangeUpdate message mesg ;            /* notify upstream */
4          send(mesg, AID);
5  procedure updateMyself(changes) do
6      processChange(changes);
7      create a ChangeUpdate message mesg;
8      send(mesg, lowestPriHigherNeighbor);
9      if local view is consistent then
10         foreach lower priority neighbor nb do
11             send(ok?, nb) ;                          /* notify downstream */
12     else
13         check_agent_view ; /* skip check local consistency inside this */
14 procedure handleChange(changes) do
15     updateMyself(changes);
16 procedure processChange(changes) do
17     add changes to awared_changes;
18     update the local CSP and neighbor list according to changes;
19     foreach change in changes do
20         if change is VarAddition then
21             get the set of controlled variables vars in change;
22             foreach v in vars do
23                 current_assignmt = current_assignmt ⋀(v=min(D_v));
24         else if change is VarRemoval then
25             get the set of variables vars in change;
26             removed_constraints = {c in constraints: var(c)∩vars != ∅};
27             processChange(ConstraintRemoval(removed_constraints));
28             delete all assignments of vars in current_assignment;
29             delete all nogoods containning any of vars in the nogood storage;
30         else if change is ConstraintRemoval then
31             get the set of constraints removed_constraints in change;
32             foreach nogood in the nogood storage do
33                 get the set of constraint sets CS in the explanation of nogood;
34                 if removed_constraints is a hit set of CS then
35                     remove nogood;
36         if change is not ConstraintRemoval then
37             empty the content of change;
```

Bibliography

[Aal03] W. M. P. van der Aalst. Don't go with the flow: web services composition standards exposed. *IEEE Intelligent Systems*, 18(1):72–76, 2003.

[ADK+05] V. Agarwal, K. Dasgupta, N. Karnik, A. Kumar, A. Kundu, S. Mittal, and B. Srivastava. A service creation environment based on end to end composition of Web services. In *WWW 05: Proceedings of the 14th international conference on World Wide Web*, pages 128–137, New York, NY, USA, 2005. ACM Press.

[Age05] Agentcities Web Services Working Group. *Integrating Web services into agentCities*. http://www.agentcities.org/rec/00006/, 2005.

[Ama05] Amazon. *Amazon Web services*. aws.amazon.com, 2005.

[AtHKB03] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[BBMMar] C. Bessiere, I. Brito, A. Maestre, and P. Meseguer. Asynchronous backtracking without adding links: A new member in the abt family. *Artificial Intelligence*, (to appear).

[BCS94] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview. Technical Report 1633, Internet Engineering Task Force, 1994.

[BMO01] B. Bauer, J. P. Muller, and J. Odell. Agent uml: a formalism for specifying multiagent software systems. In *First international workshop, AOSE 2000 on*

Agent-oriented software engineering, pages 91–103, Secaucus, NJ, USA, 2001. Springer-Verlag New York, Inc.

[BPR99] F. Bellifemine, A. Poggi, and G. Rimassa. JADE - a FIPA-compliant agent framework. In *Proceedings of the Practical Applications of Intelligent Agents*, 1999.

[BTPT06] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS 06: Proceedings of the IEEE International Conference on Web Services*, pages 63–71, Washington, DC, USA, 2006. IEEE Computer Society.

[CH05] Y. L. Chong and Y. Hamadi. Reducing redundant messages in the asynchronous backtracking algorithm. In *6th International Workshop on DCR*, 2005.

[Chu05] A. Chun. *NSolver home page*. www.cs.cityu.edu.hk/ hwchun/nsolver/, 2005.

[CK00] Fangzhe Chang and Vijay Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *HPDC*, pages 11–20, 2000.

[Com95] Douglas Comer. *Internetworking with TCP/IP, Volume 1: Principles, Protocols, and Architecture*. Prentice Hall, 1995.

[CPE+06] G. Canfora, M. D. Penta, R. Esposito, F. Perfetto, and M. L. Villani. Service composition (re)binding driven by application-specific qos. In *ICSOC*, pages 141–152, 2006.

[D'A06] A. D'Ambrogio. A model-driven wsdl extension for describing the qos ofweb services. In *ICWS 06: Proceedings of the IEEE International Conference on Web Services (ICWS 06)*, pages 789–796, Washington, DC, USA, 2006. IEEE Computer Society.

[DBC04] H. A. Duran-Limon, G. S. Blair, and G. Coulson. Adaptive resource management in middleware: A survey. *IEEE Distributed Systems Online*, 5(7):1, 2004.

[DBL04] Proceedings of the 16th eureopean conference on artificial intelligence, ecai'2004, including prestigious applicants of intelligent systems, pais 2004, valencia, spain, august 22-27, 2004. In Ramon López de Mántaras and Lorenza Saitta, editors, *ECAI*. IOS Press, 2004.

[DD88] Rina Dechter and Avi Dechter. Belief maintenance in dynamic constraint networks. In *Proceedings of the Seventh Annual Conference of the American Association of Artificial Intelligence*, pages 37–42, 1988.

[Dis04] Distributed Management Task Force (DMTF). *WBEM: Web Services for Management*. http://www.dmtf.org/standards/wbem/wsman, 2004.

[Dis07] Distributed Management Task Force. Web-based enterprise management (wbem), 2007.

[FF99] Christian Frei and Boi Faltings. Resource allocation and constraint satisfaction techniques. In *Principles and Practice of Constraint Programming*, pages 204–218, 1999.

[FIP06] FIPA. *The foundation of Intelligent Physical Agents*. http://www.fipa.org/, 2006.

[Fre78] Eugene C. Freuder. Synthesizing constraint expressions. volume 21, pages 958–966, New York, NY, USA, 1978. ACM Press.

[GC04a] D. Greenwood and M. Calisti. An automatic, bi-directional service integration gateway. In *International workshop on Web Services Agent-Based Engineering (WSABE 04)* , 2004.

[GC04b] D. Greenwood and M. Calisti. Engineering Web service - Agent integration. In *IEEE Systems, Cybernetics and Man Conference*, pages 1918–1925, 2004.

[GG05] Ying Guan and Aditya K. Ghose. Dealing with web service qos factors using constraint hierarchy. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering*, pages 578–583, 2005.

[GHM+07] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. *SOAP Version 1.2 Part 1: Messaging Framework*. World Wide Web Consortium (W3C), 2nd edition, April 2007. URL: http://www.w3.org/TR/soap12-part1/.

[GNCW03] X. Gu, K. Nahrstedt, R. Chang, and C. Ward. Qos-assured service composition in managed service overlay networks. In *23rd International Conference on Distributed Computing Systems*. IEEE CS Press, 2003.

[Goo05] Google. *Google SOAP Search API (Beta)*. http://code.google.com/apis/soapsearch/reference.html, 2005.

[HE80] R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[Hew04a] Hewlett-Packard. *Web Services Distributed Management: Management of Web Services*. docs.oasis-open.org/wsdm/2004/12/mows/cd-wsdm-mows-1.0.pdf, 2004.

[Hew04b] Hewlett-Packard. *Web Services Distributed Management: Management Using Web Services*. docs.oasis-open.org/wsdm/2004/12/muws/cd-wsdm-muws-part1-1.0.pdf, 2004.

[Hew04c] Hewlett-Packard. *Web Services Management Framework*. devresource.hp.com/drc/specifications/wsmf/index.jsp, 2004.

[HSB+05] Michael N. Huhns, Munindar P. Singh, Mark Burstein, Keith Decker, Ed Durfee, Tim Finin, Les Gasser, Hrishikesh Goradia, Nick Jennings, Kiran Lakkaraju, Hideyuki Nakashima, Van Parunak, Jeffrey S. Rosenschein, Alicia Ruvinsky, Gita Sukthankar, Samarth Swarup, Katia Sycara, Milind Tambe, Tom Wagner, and Laura Zavala. Research directions for service-oriented multiagent systems. *IEEE Internet Computing*, 9(6):65–70, 2005.

[HY00] K. Hirayama and M. Yokoo. An approach to overconstrained distributed constraint satisfaction problems: Distributed hierarchical constraint satisfaction. In *In Proceedings of ICMAS-2000*, 2000.

[IAM05] IAMAS Group. *Web services to agents: WS2JADE*. http://www.it.swin.edu.au/centres/ciamas/tiki-index.php?page=ws2jade-proj, 2005.

[IAM06] IAMAS Group. *Web services to agents: WS2JADE*. http://ciamas.it.swin.edu.au/software/simulator-doc/, 2006.

[IBM04] IBM. *alphaWorks*. http://www.alphaworks.ibm.com/, 2004.

[ILO05] ILOG. *ILOG Constraint Programming*. http://www.ilog.com/products/cp/, 2005.

[Int89] Internet Engineering Task Force. Rfc 1122: Requirements for internet hosts – communication layers, 1989.

[JCZ03] J. Jang, Y. Choi, and J. L. Zhao. Adaptive workflow management with open kernel framework based on web services. In *ICWS*, pages 124–130, 2003.

[JRG04] M. C. Jaeger, G. Rojec-Goldmann, and G.Muhl. Qos aggregation for service composition using workflow patterns. In *Proceedings of the 8th International Enterprise Distributed Object Computing Conference (EDOC 2004)*, pages 149–159, Monterey, California, USA, 2004. IEEE CS Press.

[KS03] Y. Kalfoglou and Y. Schorlemmer. Ontology mapping: the state of the art. *The Knowledge Engineering Review*, 18:1–31, 2003.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[LJS+03] X. Luo, N. Jennings, N. Shadbolt, H. Leung, and J. Lee. A fuzzy constraint based model for bilateral multi-issue negotiations in semi-competitive environments. *Artificial Intelligence Journal*, 148(1-2):53–102, 2003.

[LLY05] M. W. Lin, K. Robert Lai, and T. J. Yu. Fuzzy constraint-based agent negotiation. *J. Comput. Sci. Technol.*, 20(3):319–330, 2005.

[LMD06] Shangmin Luan, Lorenzo Magnani, and Guozhong Dai. Algorithms for computing minimal conflicts. *Logic Journal of the IGPL*, 14(2):391–406, 2006.

[LNZ04] Y. Liu, A. H. Ngu, and L. Z. Zen. QoS computation and policing in dynamic web service selection. In *WWW Alt. 04: Proceedings of the 13th international World Wide Web conference on Alternate track papers and posters*, pages 66–73, New York, NY, USA, 2004. ACM Press.

[Mai05] R. Mailler. Comparing two approaches to dynamic, distributed constraint satisfaction. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 1049–1056, New York, NY, USA, 2005. ACM Press.

[M.C03] M.C.Silaghi. Solving a distributed csp with cryptographic multiparty computations, without revealing constraints and without involving trusted servers. In *Workshop on Distributed Constraint Reasoning DCR-03 IJCAI*, 2003.

[Mei91] Itay Meiri. Combining qualitative and quantitative constraints in temporal reasoning. In Thomas Dean and Kathleen McKeown, editors, *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 260–267, Menlo Park, California, 1991. AAAI Press.

[Men02] D. A. Menasc. Qos issues in web services. *IEEE Internet Computing*, 6(6):72–75, 2002.

[Men04] D. A. Menasce. Composing web services: A qos view. *IEEE Internet Computing*, 8(6):88–90, 2004.

[MH86] Roger Mohr and Thomas C. Henderson. Arc and path consistency revisited. volume 28, pages 225–233, 1986.

[MJT+01] P. J. Modi, H. Jung, M. Tambe, W. M. Shen, and S. Kulkarni. Dynamic distributed resource allocation: A distributed constraint satisfaction approach. In *Pre-proceedings of the Eighth International Workshop on Agent Theories, Architectures, and Languages*, pages 181–193, 2001.

[ML97] P. Meseguer and J. Larrosa. Solving fuzzy constraint satisfaction problems. In *Proceedings of the Sixth IEEE International Conference on Fuzzy Systems*, pages 1233 – 1238, 1997.

[ML04] R. Mailler and V. Lesser. Solving distributed constraint optimization problems using cooperative mediation. In *Proceedings of the Third International Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 438–445. ACM, 2004.

[ML06] Roger Mailler and Victor Lesser. Asynchronous Partial Overlay: A New Algorithm for Solving Distributed Constraint Satisfaction Problems. *Journal of Artificial Intelligence Research*, 25:529–576, April 2006.

[MM88] Roger Mohr and G. Masini. Good old discrete relaxation. In *European Conference on Artificial Intelligence*, pages 651–656, 1988.

[MS05] K. Mahbub and G. Spanoudakis. Run-time monitoring of requirements for systems composed of web-services: Initial implementation and evaluation experience. *icws*, 0:257–265, 2005.

[MSTY05] P. J. Modi, W. M. Shen, M. Tambe, and M. Yokoo. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 16(1–2):149–180, 2005.

[NK05a] X. T. Nguyen and R. Kowalczyk. Enabling agent based management of web services with ws2jade. In *Proceedings of the 5th International Conference on Quality Software*, pages 407–412, 2005.

[NK05b] X. T. Nguyen and R. Kowalczyk. Ws2jade: Integrating web service with jade agents. In *Proceedings of the Workshop on Service-Oriented Computing and Agent-Based Engineering (SOCABE2005) in conjunction with 4th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS2005)*, pages 15–22, 2005.

[NK06] X. T. Nguyen and R. Kowalczyk. An agent based qos conflict mediation framework for web services compositions. In *Proceedings of IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 522–528, 2006.

[NK07] X. T. Nguyen and R. Kowalczyk. On solving distributed fuzzy constraint satisfaction problems with agents. In *Proceedings of IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, 2007.

[NKH06] X. T. Nguyen, R. Kowalczyk, and J. Han. Using dynamic asynchronous aggregate search for quality guarantees of multiple web services compositions. In *Proceedings of the 4th International Conference on Service Oriented Computing*, pages 129–140, 2006.

[NKP06a] X. T. Nguyen, R. Kowalczyk, and K. A. Phan. Dynamic asynchronous aggregate search for solving qos compositions of web services. In *SERCOMP06 workshop in conjunction with IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 142–145, 2006.

[NKP06b] X. T. Nguyen, R. Kowalczyk, and K. A. Phan. Modelling and solving qos composition problem using discsp. In *In Proceedings of the 3rd International Conference on Grid Service Engineering and Management*, pages 179–195, 2006.

[NKP06c] X. T. Nguyen, R. Kowalczyk, and M. T. Phan. Modelling and solving qos composition problem using fuzzy discsp. In *Proceedings of the IEEE International Conference on Web Services*, pages 55–62, 2006.

[OAS06a] OASIS. *OASIS Web Services Notification (WSN)*. www.oasis-open.org/committees/wsn/, 2006.

[OAS06b] OASIS. *Web service Resource Framework: WSRF*. www.oasis-open.org/committees/wsrf/, 2006.

[OAS06c] OASIS. *Web Services Business Process Execution Language Version 2.0*. http://www.oasis-open.org/committees/download.php/18714/wsbpel-specification-draft-May17.htm, 2006.

[OAS07] OASIS. http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm, 2007.

[Ope03] Open Grid Forum (OGF). *WS-Agreement Specification*. http://www.gridforum.org/Meetings/GGF11/Documents/draft-ggf-graap-agreement.pdf, 2003.

[PEH+00] T. Plagemann, F. Eliassen, B. Hafskjold, T. Kristensen, R. H. Macdonald, and H. O. Rafaelsen. Flexible and extensible qos management for adaptable middleware'. In *Proceedings of International Workshop on Protocols for Multimedia Systems (PROMS)*, Cracow, Poland, 2000.

[PEWS03] G. Piccinelli, W. Emmerich, S. L. Williams, and M. Stearns. A model-driven architecture for electronic service management systems. In *ICSOC*, pages 241–255, 2003.

[PF07] A. Petcu and B. Faltings. Mb-dpop: A new memory-bounded algorithm for distributed optimization. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI-07*, Hyderabad, India, Jan 2007.

[POSV04] A. A. Patil, S. A. Oundhakar, A. P. Sheth, and K. Verma. Meteor-s web service annotation framework. In *WWW 04: Proceedings of the 13th international conference on World Wide Web*, pages 553–562, New York, NY, USA, 2004. ACM Press.

[Pra03] M. J. Pragnesh. *ADOPT*. http://www.cs.cmu.edu/ pmodi/adopt/, 2003.

[PS03] A. K. Patankar and A. Segev. Inter-organizational business process management using web services - a case study and research issues. In *ICWS*, pages 138–144, 2003.

[PSL03] C. Patel, K. Supekar, and Y. Lee. A QoS Oriented Framework for Adaptive Management of Web Service Based Workflows. In *Lecture Notes in Computer Science: Database and Expert Systems Applications*, pages 826–835. Springer Berlin / Heidelberg, 2003.

[RAC+02] B. Raman, S. Agarwal, Y. Chen, M. Caesar, W. Cui, P. Johansson, K. Lai, T. Lavian, S. Machiraju, Z. M. Mao, G. Porter, T. Roscoe, and R. Mukund. The sahara model for service composition across multiple providers. In *Proc. of International Conference on Pervasive Computing, Zurich, Switzerland*, Aug 2002.

[RD05] K. Ranganathan and A. Dan. Proactive management of service instance pools for meeting service level agreements. In *ICSOC*, pages 296–309, 2005.

[RHF+05] G. Rochart, C. Hadrien, L. Francois, N. Jussien, and T. Benoist. *Choco: A java library for constraint satisfaction problems.* http://choco-solver.net, 2005.

[Rin01] Georg Ringwelski. Distributed constraint satisfaction with cooperating asynchronous solvers. In *CP*, page 777, 2001.

[RLLS97] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for qos management. In *IEEE Real-Time Systems Symposium*, pages 298–307, Dec 1997.

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Pearson Education, 2003.

[SC00a] F. Siqueira and V. Cahill. Quartz: A qos architecture for open systems. In *International Conference on Distributed Computing Systems*, pages 197–204, 2000.

[SC00b] Frank Siqueira and Vinny Cahill. Quartz: A qos architecture for open systems. In *International Conference on Distributed Computing Systems*, pages 197–204, 2000.

[Sei81] Raimund Seidel. A new method for solving constraint satisfaction problems. In *IJCAI*, pages 338–342, 1981.

[SF05] Marius C. Silaghi and Boi Faltings. Asynchronous aggregation and consistency in distributed constraint satisfaction. In *Artificial Intelligence Journal Vol.161*, pages 25–53, New York, NY, USA, 2005. ACM Press.

[SF02] M. C. Silaghi and B. Faltings. Openness in asynchronous constraint satisfaction algorithms. In *3rd DCR-02 Workshop*, pages 156–166, 202.

[SH04] T. Szigeti and C. Hattingh. *End-To-End QoS Network Design: Quality of Service in LANs, WANs, and VPNs*. Cisco Press, 2004.

[SS77] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. volume 9, pages 135–196, 1977.

[SSF99] M. C. Silaghi, D. Sam-Haroud, and B. Faltings. Intelligent domain splitting for csps with ordered domains. In *Principles and Practice of Constraint Programming*, pages 488–489, 1999.

[SSF00] M. Calin Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *AAAI/IAAI*, pages 917–922, 2000.

[SSF01] M. C. Silaghi, D. Sam-Haroud, and B. Faltings. Polynomial space and complete multiply asynchronous search with abstractions. In *IJCAI'01 workshop on Distributed Constraint Reasoning*, 2001.

[Sta98] W. Stallings. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. Addison Wesley, 1998.

[Sta02] Michael Stal. Web services: beyond component-based computing. *Commun. ACM*, 45(10):71–76, 2002.

[Sun07] Sun Microsystems, Inc. Java management extensions (jmx) technology, 2007.

[SZ99] I. Stoica and H. Zhang. Providing guaranteed services without per flow management. In *SIGCOMM*, pages 81–94, 1999.

[TRPA06] D. T. Tsesmetzis, I. G. Roussaki, I. V. Papaioannou, and M. E. Anagnostou. Qos awareness support in web-service semantics. In *AICT-ICIW 06: Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services*, page 128, Washington, DC, USA, 2006. IEEE Computer Society.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction*, chapter Chapter 2 CSP solving An overview, pages 31–52. Academic Press, 1993.

[Ver06] Verisign. *Verisign Homepage*. http://www.verisign.com/, 2006.

[VH03] L. Zs. Varga and A. Hajnal. Engineering web service invocations from agent systems. In *International Central and Eastern European Conference on Multi-agent Systems, CEEMAS 2003*, pages 626–635, 2003.

[VJ05] Gérard Verfaillie and Narendra Jussien. Constraint solving in uncertain and dynamic environments: A survey. *Constraints*, 10(3):253–281, 2005.

[VS94a] G. Verfaillie and T. Schiex. Dynamic backtracking for dynamic constraint satisfaction problems. In *Proceedings of the ECAI'94 Workshop on Constraint Satisfaction Issues Raised by Practical Applications, Amsterdam, The Netherlands*, pages 1–8, 1994.

[VS94b] G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *National Conference on Artificial Intelligence*, pages 307–312, 1994.

[W3C] W3C. *Web Services Architecture*. http://www.w3c.org/TR/ws-arch.

[W3C03] W3C. *QoS for Web Services: Requirements and Possible Approaches*. http://www.w3c.or.kr/kr-office/TR/2003/ws-qos/, 2003.

[W3C05a] W3C. *Web Service Choreography Interface (WSCI) 1.0*. http://www.w3.org/TR/wsci/, 2005.

[W3C05b] W3C. *Web service Description Language*. http://www.w3.org/TR/wsdl, 2005.

[W3C06] W3C. *Web Services Choreography Description Language Version 1.0*. http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/, 2006.

[W3C07a] W3C. *Extensible Markup Language (XML)*. http://www.w3.org/XML/, 2007.

[W3C07b] W3C. *Web service use case: Travel reservation*. http://www.w3.org/2002/06/ws-example.xml, 2007.

[Whi05] Whitestein Information Technology Group. *Web services agent Integration Project*. http://wsai.sourceforge.net/index.html, 2005.

[wsl04] The wsla framework: Specifying and monitoring service level agreements for web services. *Journal of Network and Systems Management*, 11:57–81, 2004.

[WVKT06] Xia Wang, Tomas Vitvar, Mick Kerrigan, and Ioan Toma. A qos-aware selection model for semantic web services. In *ICSOC*, pages 390–401, 2006.

[xcs07] *XML Representation of Constraint Networks Version 2.0*. http://www.cril.univ-artois.fr/ lecoutre/research/tools/format2.pdf, 2007.

[YDIK92] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *International Conference on Distributed Computing Systems*, pages 614–621, 1992.

[YH00] Makoto Yokoo and Katsutoshi Hirayama. Algorithms for distributed constraint satisfaction: A review. *Autonomous Agents and Multi-Agent Systems*, 3(2):185–207, 2000.

[YL04] Tao Yu and Kwei-Jay Lin. Service selection algorithms for web services with end-to-end qos constraints. In *CEC '04: Proceedings of the IEEE International Conference on E-Commerce Technology (CEC'04)*, pages 129–136, Washington, DC, USA, 2004. IEEE Computer Society.

[YN04] W. Yuan and K. Nahrstedt. Process group management in cross-layer adaptation. In *In Proc. of SPIE/ACM Multimedia Computing and Networking Conference*, 2004.

[Yok95] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proc. 1st Intrnat. Conf. on Const. Progr.*, pages 88–102, Cassis, France, 1995.

[Yok01] Makoto Yokoo. *Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-agent Systems.* Springer Berlin / Heidelberg, 2001.

[YSH05] M. Yokoo, K. Suzuki, and K. Hirayama. Secure distributed constraint satisfaction: reaching agreement without revealing private information. *Artif. Intell.*, 161(1-2):229–245, 2005.

[Zad75] L. A. Zadeh. The concept of a linguistic variable and its applications to approximate reasoning - i. *Information Sciences*, 8:199–249, 1975.

[ZM06] Roie Zivan and Amnon Meisels. Dynamic ordering for asynchronous backtracking on discsps. *Constraints*, 11(2-3):179–197, 2006.