Using Dynamic Asynchronous Aggregate Search for Quality Guarantees of Multiple Web Services Compositions

Xuan Thang Nguyen, Ryszard Kowalczyk, and Jun Han

Swinburne University of Technology, Faculty of Information and Communication Technologies,Melbourne VIC 3122, Australia
{xnguyen, rkowalczyk, jhan}@ict.swin.edu.au

Abstract. With the increasing impact and popularity of Web service technologies in today's World Wide Web, composition of Web services has received much interest to support enterprise-to-enterprise application integrations. As for service providers and their partners, the Quality of service (QoS) offered by a composite Web service is important. The QoS guarantee for composite services has been investigated in a number of works. However, those works consider only an individual composition or take the viewpoint of a single provider. In this paper, we focus on the problem of QoS guarantees for multiple inter-related compositions and consider the global viewpoints of all providers engaged in the compositions. The contributions of this paper are two folds. We first formalize the problem of QoS guarantees for multi-compositions and show that it can be modelled as a Distributed Constraint Satisfaction Problem (DisCSP). We also take into account the dynamic nature of the Web service environment of which compositions may be formed or dissolved any time. Secondly, we present a dynamic DisCSP algorithm to solve the problem and discuss our initial experiment to show the feasibility of our approach for multiple Web service compositions with QoS guarantees.

1 Introduction

During the past few years, in an effort to improve the collaborations between organizations, the Web service framework has been emerging as a de-facto choice for integrating distributed and heterogeneous applications across organizational boundaries. Consequently, much research has been carried out in various areas including Web service discovery, composition, and management. Web service composition in general focuses on building a new value-added *composite* Web service from a number of existing *component* Web services. A Web service composition can be considered as a *choreography* or an *orchestration* of Web services from different viewpoints. A *choreography* describes a composition from a 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. While Web service orchestration has enjoyed its popularity with an

A. Dan and W. Lamersdorf (Eds.): ICSOC 2006, LNCS 4294, pp. 129–140, 2006.

increasing number of support tools and implementations [13], Web service choreography standards emerge rather late with the replacement of WSCI/WSCL [17] by WS-CDL [18]. Without choreography, Web services compositions' examples [8,6] are often restricted to a model which we call the *single provider* composition model. For a *single provider* composition, a provider searches for available Web services, combines them together to form a new composite Web service that it can offer. The major characteristic of a *single provider* composition is that it is planned and composed solely by a single provider or QoS broker. The composition may even not be noticed by component service providers. With the increasing popularity of Web service choreography as a mechanism for *multi-party* contracts [18], Web services choreography opens up the possiblities for *multi-provider* compositions in which every participant has some vested interest in the composite service and actively engages in composing the service. QoS guarantees for an individual *single provider* composition has been well investigated in a number of works [3,6,8]. However, to our knowledge, there has not been any works on the same problem for multiple related *multi-provider* compositions in which multiple providers collaborate to guarantee the QoS levels of composite services.

In parallel to the advancement of Web services, the MAS and AI communities have shown an increasing interest in the Distributed Constraint Satisfaction Problem (DisCSP) in the past few years. DisCSP has been widely viewed as a powerful paradigm for solving combinatorial problems arising in distributed, multi-agent environments. A DisCSP is a problem with 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 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. In this paper, we investigate the application of DisCSP techniques to the QoS guarantee problem and propose a new DisCSP based algorithm for multiple multi-provider Web service compositions. The rest of the paper is organized as follows. In the next section we present some important related work. We discuss how the QoS guarantee for Web service composition problems can be modelled as Dynamic DisCSP problems in Section 3. We also present formal descriptions of the QoS guarantee for multi-provider compositions, DisCSP and Dynamic DisCSP frameworks in that section. We review the AAS (Asynchronous Aggregate Search) algorithm for its application in the problem of QoS guarantees and describe our proposed DynAAS (Dynamic Asynchronous Aggregate Search) algorithm in Section 4. Section 5 present our experiment on the algorithm's performance. Finally, conclusions and future work are discussed in Section 6.

2 Related Work and Open Issues

There have been several studies on QoS of Web service compositions. QoS guarantees for compositions are discussed in [8,7,6]. In [7], the authors discuss an approach for QoS aggregation based on Web service composition patterns [14].

More related work on QoS planning can be found in [8], in which a method for selecting optimal sub-providers from a list of service providers is proposed. In [3], the authors model the QoS requirements as an optimization problem and employ a special centralized CSP technique to solve it. However, we argue that there are three major issues that have not been addressed in those works:

- QoS guarantees for multi-provider compositions: As we explained before, choreography offers a new service model in which a number of component service providers may share common goals and hence *collaborate* together to offer a composite service. Most of the current QoS composition research focus on *single provider* composition. Therefore they look at a composition from a local view of a single provider, as for orchestration. The QoS composition for multi-provider compositions, as for choreography, requires a global view.
- Multiple inter-related compositions: Some services or service providers may engage in many compositions and hence there is a relationship between these compositions through the shared services and providers. This relationship needs to be taken into account in the composition planning.
- Discovery of supported QoS levels: Most of current QoS composition research assume that service providers *publicly* advertise precisely their supported QoS levels. This can be done by categorizing different *classes of service* and embedding the supported QoS values directly into WSDL interfaces or UDDI registries [11]. However, such a *public* advertisement requires disclosure of private information and is not the only way for QoS discovery. Works on negotiation [5] suggest that QoS discovery can be achieved by direct negotiation between a client and a service provider. By doing this, supported QoS levels can be kept private.

In addition, we argue that public advertisement is more suitable for *atomic* services (i.e services which do not use third party component services). For a composite service which uses a third party service, supported QoS can be better negotiated because the composite service provider might replace the third pary service with a better one at runtime. Of course here we assume that self-reconfiguration can be done within the composite service and this is the subject of research in [1]. The dynamic runtime change in the structure of a composite service suggests that a set of pre-defined QoS levels for that service may not be desirable. Align to this argument, there are works on "services on demand" [4,1] platforms which attempt to satisfy any QoS requirements from clients. In the remaining part of this paper, we introduce a framework to handle the above three issues. Our framework focuses on a global view as opposed to work in [1,8,11,6].

3 Formalization of the QoS Guarantee Problem for Web Service Compositions

We present a motivation example in Figure 1 which shows four composite services: Mel(burne)-Tourist, Aus(tralia)-Tourist, Syd(ney)-Tourist, and Aus-Attraction which make up the set $\mathbf{S}_{composite}$. The composite services are composed from six

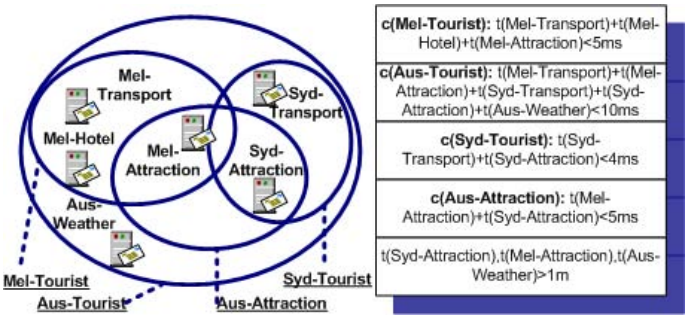

Fig. 1. An example of multiple compositions for tourist related services

individual services: Mel-Transport, Mel-Hotel, Mel-Attraction, Syd-Transport, Syd-Attraction, and Aus-Weather which make up the set $\mathbf{S}_{component}$.

Since a service provider must allocate necessary resources to live up to the QoS guarantees, if its service engages in a number of compositions, there will be a dependency between the levels of QoS that service can contribute to these compositions. Consequently, there is a mutual relationship between the compositions. Here we assume that this relationship can be formally expressed as constraints. For the sake of clarity, we assume the response time is our only interested QoS parameter. The t(S) variable in Figure 1 represents the response time of a Web service S, S$\in\mathbf{S}_{composite}\cup\mathbf{S}_{component}$. We also assume that every composition in $\mathbf{S}_{composite}$ is a sequential combination of its component services and hence its E2E (end-to-end) response time can be computed as a sum of the component services' response time. For other QoS parameters and composition patterns, the E2E QoS can be computed with different aggregation operators [7]. The QoS requirements on the values of these sums form the set of constraints $\{c(S):S\in\mathbf{S}_{composite}\}$. We note that these constraints c(S) are only *shared* among service providers who engage in the composition S(i.e. not all providers). In addition to these shared constraints, each provider has its private constraints as shown in the last row of the table in Figure 1. These constraints might be shaped by the provider's resource limitations, business rules, organizational policies or even conditions in contracts with a third party. The providers have a choice to reveal them or not by making the constraints *shared* (i.e. known to a number of or all other providers) or *private* respectively. Here we focus on a general problem in which multiple providers engage in multi-Web service compositions. The final goal of the QoS guarantee for multiple Web service compositions is to satisfy the E2E QoS requirements of all compositions. Formally, the problem can be stated as:

Definition 1. *Given m service providers participate in n compositions and a set of pre-defined E2E QoS requirements for those compositions. The problem of QoS guarantee for multi-Web service compositions is to assign QoS values to each component service so that these values can be supported by the service's provider and all the compositions meet their QoS requirements.*

Some main characteristics of the QoS guarantee problem for multi-Web service compositions that makes it more complex and difficult than the single-provider QoS composition problem are:

- **Prop$_1$:** Many providers engage in the composition process. They may publish their supported QoS levels or require direct negotiations.
- **Prop$_2$:** Many compositions need to be considered concurrently. QoS planning in one composition may affect another composition.

In a real world Web services environment, there are two main sources of dynamism regarding the compositions and the constraints. They are also important characteristics of the QoS guarantee problem for multi-Web service compositions:

- **Prop$_3$:** Compositions can be formed and disbanded any time, e.g. compositions in $\mathbf{S}_{composite}$ do not appear and disappear at the same time. They might be formed or dropped one after one.
- **Prop$_4$:** Service providers might have their own constraints changed during their service lifetime. QoS requirements for a composition might also change (e.g. changes in user's requests).

The characteristic **Prop$_3$** reflects many possibilities. One of them is that some providers may realize that the final goal to satisfy the QoS requirements of all compositions may not be achieved. They then drop less important compositions (according to their own ratings) and hence the original problem of QoS guarantee for Web service compositions is transformed into a new easier one to solve.

4 Modelling the QoS Guarantee Problem for Web Service Compositions as an Instance of DisCSP

Based on the above discussion, it is proposed that DisCSP techniques can be well suited for modelling and solving the QoS guarantee problem for Web service composition. More specifically:

- The distributed nature of the Web environment and the engagement of many participants in multi-provider services suggest that a distributed approach is best suited.
- Constraints in the QoS guarantee for multi-Web service composition problem can be either private or shared. Distributed constraints with different visibility levels have been one of the main focuses of DisCSP techniques.

To apply DisCSP techniques for solving the problem of QoS guarantee for multi-Web services compositions, each service provider can be considered as an agent (an autonomously processing entity) in a constraint network. Each QoS parameter is mapped into a variable in the constraint network; the set of providers' constraints is mapped into the network's constraint set. For the rest of this paper, we will use the terms *service providers* and *agents* interchangeably. More formally, the problem of QoS guarantee for Web service compositions can be considered as an instance of DisCSP problems of which the general definition is:

Definition 2. *A static distributed constraint satisfaction problem P is a tuple* $\langle V,D,C,A\rangle$ *where* $V=\{x_1,...,x_n\}$ *is a set of variables,* $D=\{D_1,...,D_n\}$ *is a set of discrete finite domains for each of the variables , and* $C=\{C_1,...,C_m\}$ *is a set of constraints on possible values of variables. These variables and constraints are distributed among a set of agents* $A=\{A_1,...,A_k\}$*. If an agent* A_l *knows a constraint* C_q*, it also must know all variables contained in* C_q*. A solution is an assignment of values in the domains to all variables such that every constraint is satisfied.*

To take into account the dynamic nature of Web services environment, we consider the implications of the properties $\mathbf{Prop}_3$, and $\mathbf{Prop}_4$ discussed in the previous section. In the DisCSP framework, the appearance of a new composition indicates that new constraints and possiblly new variables and agents are added into the constraint network. Dissolving of a composition means that some existing constraints are removed and possibly some existing variables or agents are also removed. In general, there are introductions or reductions of new variables, constraints, and agents.

Traditionally, Dynamic CSP (DynCSP) is a branch of CSP. Its goal is to effectively handle CSP problems with dynamic changes instead of restarting a static search every time a change is detected. DynCSP has been modestly extended into distributed environments [9].A formal description of dynamic DisCSP followed by a dynamic CSP definition in [16] can be given as:

Definition 3. *A dynamic distributed constraint satisfaction problem P is a sequence* $P^0...,P^i...$ *of static DisCSPs, where each one resulting from a change in the preceding one. This change may be a restriction or a relaxation.*

A restriction can be caused by more agents, variables, or new constraints. A relaxation results from removing agents, variables, or constraints. Note that a change rate is important to measure and specify how fast a DynDisCSP changes over time. This rate, defined as Δ can be measured as the total of added or removed constraints between any two P^i and P^j over the time distance between them.

5 DisCSP Algorithms to Solve the QoS Guarantee Problem for Web Service Compositions

There have recently been many publications on DisCSP algorithms. Traditionally these algorithms are developed and demonstrated in the context of the Meeting Scheduling and Sensor Network [2]. However, there are some characteristics that make the QoS guarantee for Web service composition problem different from those problems: Firstly each agent holds a set (often more than one) of variables to represent QoS parameters; secondly local constraints in QoS problem can be very complex; and thirdly service providers are heterogeneous and hence flexibility in algorithm implementations is desireable. In searching for a suitable DisCSP algorithm, these characteristics are the most important criteria for us.

Whilst most DisCSP algorithms can be extended so that one agent can hold more than one variable, substantial effort is required for that and for handling complex private constraints. The originial DisCSP model [20] and most of the solving algorithms focus on shared constraints instead of private constraints and hence is more suitable for distributed control but not negotiation. A notable exception is Asynchronous Aggregate Search (AAS) [12] that allows one agent to maintain a set of variables and these variables can be shared and hence is suitable for negotiation. Also all constraints are private in AAS (shared constraints can be modeled as duplicated private constraints). However, in the current version of AAS, private constraints at each agent are assumed to be simple and hence there is no attention in solving these local constraints. Also, AAS is designed only for static environments. In this section, we introduce AAS and suggest to use a centralized CSP solver inside each agent to handle complex local constraints. We also propose an extension of AAS called DynAAS to handle the dynamic nature of Web services environment.

5.1 Asynchronous Aggregate Search and Local CSP Solvers

Here we briefly introduce AAS in the Web services context. A complete explanation of AAS can be found in [12] where its termination, correctness and completeness are proven. Asynchronous Aggregate Search (AAS) is a DisCSP search technique based on the classical Asynchronous Backtrack (ABT) algorithm [20]. In AAS, each agent (service provider) maintains a set of variables (relevant QoS variables in our Web services QoS guarantee problem) which can be shared with others and a set of private constraints on the values of these variables. AAS differs from most of existing methods in that it exchanges aggregated consistent values (in contrast to a single value in ABT) of partial solutions during the solving process. The aggregated consistent values are the Cartesian products of domains which represent a set of possible valuations. This aggregate significantly reduces the number of backtracks. At the beginning, AAS agents are (randomly) assigned with priorities and generate random assignments (i.e. proposals). Two agents are neighbors if they share some variables. During search, each agent A sends assignments in *ok?* messages to A^+ or rejections in *nogood* messages to A^-. Here we denote A^+ the set of neighboring agents whose priorities are higher, and A^- the set of neighboring agents whose priorities are lower than A's priority. V^+ is the set of variables the agent share with A^+, and V^- is with A^-. An agent can also send *addneighbor* to ask another agent to become its neighbor. Each agent keeps a view (current assignments of its variables and variables in V^+) and a list of nogoods (assignments rejected by A^-).

In AAS, an agent implements three main procedures *process-ok*, *process-nogood*, and *process-addneighbor* to handle incoming *ok?*, *nogood*, and *addneighbor* messages. These procedures check whether the information of a partial solution in the messages is still compatible with the agent's own assignment of its variables. The procedures may invoke a *check-agent-view* procedure to find out a new compatible assignment for the agent's local variables. In particular, the procedure *process-ok* updates the agent-view and nogood list from the remaining valid assignments

before possibly invoking the *check-agent-view* procedure. The procedure *process-nogood* updates its view according to new assignments found in the nogood content. If the nogood invalidates the current instantiation and contains new variables then the agent will try to establish new links with agents in $A^+{}_k$ which hold these variables. The procedure *check-agent-view* is used to find a new instantiation and sends updated values in this instantiation to appropriate agents in $A^-{}_k$. To effectively handle the complexity of local constraints, we introduce a local CSP solver into each agent. Instead of carrying out a simple local search as in the originial version of AAS, our *check-agent-view* employs a local CSP solver to find an aggregate V over the Cartesian product of domains of the agent's variables so that the current agent-view and V are consistent and satisfy the agent's local constraints. In general, the CSP solver of an agent A takes assignments from A^- and generates solutions for V^+. If a solution cannot be found for an assignment from an agent in A^+, a *nogood* message is backtracked to this agent. Otherwise, new assignments generated by A and sent to A^-.

5.2 Dynamic Asynchronous Aggregate Search

Our new extension of AAS for dynamic environment is based on an indexing technique called *eliminating explanation* which had been proposed in centralized DynCSP [15]. Note that (nogood based) CSP algorithms in general generate and test solutions, and record nogoods (invalid solutions). The main idea of the *eliminating explanation* technique is simple enough: to index every nogood against the minimal set of constraints that create the nogood, and remove the nogood if a constraint in the constraint set is removed. In DynAAS, an agent creates and stores an *eliminating explanation* before it sends a nogood. The sent nogood is tagged with an identity number and kept by both the sender and the receiver so that the sender, due to some changes later, can ask the receiver to remove this nogood. It does this by sending the receiver a *remove-nogood* message that contains the nogood's identity.

Algorithm 1 shows a procedure *add-constraints* used by an agent to handle a newly added constraint set C_{new}. In the algorithm, the agent first tries a local repair of the partial assignments of variables contained in both those constraints and V^+ (line 2). If it fails, the agent then attempts to repair the assignments of the whole V^+ (line 4). New assignments if exist are used to update the view and sent to A^+, otherwise a backtrack occurs.

Algorithm 2 explains a procedure *remove-constraints* which handles the removal of a constraint set $C_{removed}$. The agent bases on its eliminating explanation set (E) to detect which nogood it sent to a parent in the past is no longer a nogood (line 2). It then sends a *remove-nogood* message to ask the parent for the removal of this nogood. The nogood is a constraint from the parent's perspective. Therefore, the parent handles the message by invoking Algorithm 2. Adding and removing constraints are the same as calling *add-constraints* and *remove-constraints* sequentially.

Algorithm 1. Add-Constraints(C_{new})

```
1: update neighbor list, variable list, and constraint list
2: assgmts =re-assign(v(C_new) ∩ V+)
3: if assgmts = ∅ then
4:    assigmts =re-assign(V+)
5: end if
6: if assgmts = ∅ then
7:    send nogoods to A−
8: else
9:    update view and send ok to A+
10: end if
```

Algorithm 2. Remove-Constraints($C_{removed}$)

```
1: update neighbor list, variable list, and constraint list
2: for all e ∈ E and c(e) ∩ C_removed ≠ ∅ do
3:    send remove-nogood message to parent to remove c
4:    delete e from E
5: end for
```

It is important to note that adding or removing of variables or domain values can be modeled as constraints [16], therefore can be handled by the two above algorithms. If a new agent is added to the network, it is given the lowest priority. For every type of changes, affected agent must update its lists of neighbors, variables, and constraints (e.g. line 1 of Algorithm 1 and 2) first. As we have seen so far, the key idea of DynAAS is to reuse partial solutions to achieve stability. If we model the whole environment as a discrete-event system where events are constraint additions or removals, then during the interval between any two consecutive events the system can be viewed as a static DisCSP using AAS. This is because DynAAS reacts to maintain consistent views and nogood storages whenever constraints are added or removed.

6 Experiments

We have built a prototype for experiment, in which we uses Axis 2 for SOAP engine running on Windows platforms. We develop a DisCSP module with 2 supported protocols: AAS and DynAAS. The module is implemented as a Web service that has an one-way operation to receive messages sent from other DisCSP modules. The endpoint reference implementation of a DisCSP module supports four different WS-Addressing actions with local names: *ok*, *nogood*, *add-neighbor*, and *remove-nogood*. These actions are used to identify a message type sent between two Web services. We use XPATH to present the constraints. We use NSolver [10] as the Local Solver module. We developed an adaptor to invoke NSolver engine (.NET process) from the Web services. The adapter transforms XPath expression into NSolver native constraints and can be downloaded

from [19]. For our experiment, 10 Web services are used. A list of prefetched constraints on QoS parameters, and neighbors (i.e addresses of other DisCSP Web services) are stored in a mySQL database. These data are modified by a simulator with the varying rate σ of adding/removing constraints. In particular, compositions are randomly formed and removed among any 3 agents. For the sake of clarity, each of our compositions consists of exactly three component services and introduces maximum two constraints on the E2E QoS parameters of the composition. The QoS parameters that we use are response time and cost which both have simple aggregation formulas [7]. Each parameter has a domain of 10 discrete values. The average of all constraint tightness (the probability that there is not a valid assignment within a constraint) is 30% . 20 instances are run for each test.

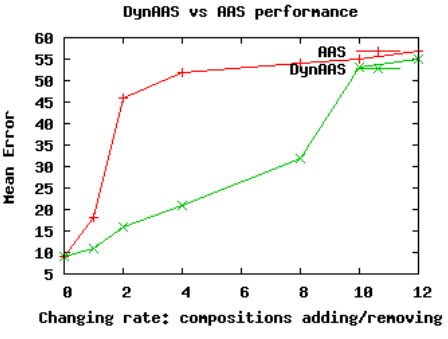

Fig. 2. Mean Error of DynAAS versus Static AAS for 10 providers with dynamic number of compositions

Before an explaining experiments' result, we first introduce some important metrics. Most of current DisCSP algorithms use *processing cycles* as a measurement of time due to its good approximation and the asynchronously distributed nature of the search (i.e. there is no global clock). A processing cycle of an agent consists of receiving a message, processing it and sending out new messages. Also for DynDisCSP, it is important to measure the rate of environmental changes. We adopt the rate function σ defined in [9]. It is defined as the first deriviative of the change rate Δ that measures how reactive an algorithm is to changes. The time unit to calculate Δ and σ is one *processing cycle*. For example, at a rate σ=4, four constraints are added or removed at each *processing cycle*. Note that because sometimes an algorithm might not keep up with the change rate, completion time is not an appropriate indicator for performance of DynDisCSP algorithms. Instead, accuracy in approximating a valid solution is used. The metric is *instantaneous error* [9] which is calculated as the distance from a current solution found by the algorithm and the valid solution bounds. Note that these valid solution bounds are computed by our simulator everytime it adds or removes a constraint.

In the experiments, we have used both static AAS and our new DynAAS algorithm. Static AAS restarts the search from scratch every time a change is detected. Figure 4 shows the performance of DynAAS and static AAS in terms of mean values of normalized *instantaneous errors* vesus rate of change. The graph shows that the error rate of DynAAS is significant lower than AAS for low changed rates and increased for larger values of σ. This can be explained as the the change rate is greater than the adaptive rate at which DynAAS can handle. However it shows that DynAAS offers significant reduction in solution errors if the change rate is reasonable. This reduction is greater than 50% for $\sigma \leq 8$. In other words, for the current setup, as long as there are no more than 8 constraints added or removed at each processing cycle then DynAAS gives twice of the level of solution accuracy over static AAS.

7 Conclusions

We have discussed in this paper the limitations of current approaches in solving general QoS composition problems and outlined a new approach for modelling and solving the QoS guarantees for multi-provider compositions as a DisCSP problem. We also describe a new extension of AAS called DynAAS for dynamic environment where unexpected events and changes can happen. Experiments show that the QoS guarantee problem for multiple Web service can efficiently be solved with DisCSP. Our future work focuses on QoS guarantees for Web service compositions and optimization of a joint interest function among all providers, such as the joint satisfaction levels of the DisCSP solution quality.

References

1. 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.
2. R. Bejar, B. Krishnamachari, C. Gomes, and B. Selman. Distributed constraint satisfaction in a wireless sensor tracking system. In *Workshop on Distributed Constraints, IJCAI*, 2001.
3. B. Benatallah, F. Casati, and P. Traverso, editors. *Service-Oriented Computing - ICSOC 2005, Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005, Proceedings*, volume 3826 of *Lecture Notes in Computer Science*. Springer, 2005.
4. A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler, H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef. Web services on demand: Wsla-driven automated management. *IBM Syst. J.*, 43(1):136–158, 2004.
5. A. Elfatatry and P. Layzell. Negotiating in service-oriented environments. *Commun. ACM*, 47(8):103–108, 2004.
6. X. Gu, K. Nahrstedt, R. Chang, and C. Ward. Qos-assured service composition in managed service overlay networks, 2003.

7. M. C. Jaeger, G. Rojec-Goldmann, and Mühl. 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.
8. Y. Liu, A. H. Ngu, and L. Z. Zeng. 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 & posters*, pages 66–73, New York, NY, USA, 2004. ACM Press.
9. 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.
10. *NSolver home page.* http://www.cs.cityu.edu.hk/ hwchun/nsolver/, 2005.
11. S. Ran. A model for web services discovery with qos. *SIGecom Exch.*, 4(1):1–10, 2003.
12. M. C. Silaghi and B. 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.
13. 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.
14. 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.
15. 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.
16. G. Verfaillie and T. Schiex. Solution reuse in dynamic constraint satisfaction problems. In *National Conference on Artificial Intelligence*, pages 307–312, 1994.
17. *Web Service Choreography Interface (WSCI) 1.0.* http://www.w3.org/TR/wsci/, 2005.
18. *Web Services Choreography Description Language Version 1.0.* http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/, 2006.
19. *XPath Adapter for NSolver.* http://www.it.swin.edu.au/centres/ciamas/tiki-index.php?page=xpath2nsolver, 2005.
20. 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.