Modelling and Solving QoS Composition Problem Using DisCSP

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

Khoi Anh Phan
Faculty of Applied Science
RMIT University
thikhoi@cs.rmit.edu.au

Abstract:
Web services have emerged as a popular technology for integrating distributed applications. A Web service composition is a description of how different Web services can interoperate in order to perform more complex tasks. QoS for composite Web services has attracted interests from both research and industrial communities. In this paper, we propose an agent-based solution for the QoS composition problem using Distributed Constraint Satisfaction Problem (DisCSP) techniques. We show that by using the composition structures, local constraints can be constructed and used with DisCSP. We also present an enhancement of the Asynchronous Aggregate Search (AAS) algorithm to solve the problem and discuss our initial experiment in building a multi-agent system to prove the feasibility of our approach.

1 Introduction

Web service technology is prevailing for business-to-business integration due to its well defined infrastructure aiming at enabling interoperability among heterogeneous applications. Many of the recent research projects have been carried out in areas including of Web service discovery, composition, and management. One of the challenges introduced by Web service composition is represented by QoS. In this paper, we argue that since a composite service is built from different component services and a component service may engage in different compositions, there are relationships between the QoS planning of these compositions. Consequently, understanding the relationships between QoS compositions of multiple related composite services is important. This has not been addressed in most of the current work (e.g. [LNZ04, ADK+05, JRGM04, GNCW]) on QoS compositions as those work separately plan the end to end QoS for an individual composition.

In parallel to the advancement of Web services, the MAS and AI community has shown an increasing interest in the Distributed Constraint Satisfaction Problem (DisCSP) in the last

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 propose a DisCSP-based technique to solve the QoS composition problem. In our proposed technique, we focus on addressing composite services with nested composition structures, where component services can be compositions themselves. The rest of the paper is organized as follows. A literature review on Web service compositions and DisCSP are presented in the next section. In Section 3, we model the QoS composition problem as a DisCSP. Section 4 proposes an algorithm to construct constraints from a Web service composition structure. An enhancement of the Asynchronous Aggregate Search (AAS) algorithm, which can handle multiple variables in each agent with a new heuristic developed to exploit QoS parameters' characteristics, is presented in Section 4. Section 5 discusses our initial experiment in building a multi-agent system as a proof of concept for our proposal. Finally, conclusions and future work are discussed in Section 6.

2 Related Work

Several studies have been carried out on QoS planning of Web service compositions, e.g. [LNZ04, ADK$^+$05, JRGM04, GNCW]. In [Men04], numeric calculations of execution time and cost of a composite service are presented. In [JRGM04], the authors discuss a method for QoS aggregation based on Web service composition patterns [vdAtHKB03]. In [LNZ04] an approach is proposed for selecting optimal sub-providers from a list of service providers. All of these research focus on a composition problem at each provider independently without considering the globally nested composition structures of a service. Consequently, these works do not make use of possible collaborations between service providers at different nested levels in a composition structure. To overcome this limitation, we propose to use an agent-based approach with DisCSP techniques in which service providers from different nested levels can collaborate to solve the QoS composition problem together. To our best knowledge, there has been no application of DisCSP in this area so far. DisCSP is a major technique for coordination and conflict resolutions in a distributed and collaborative environment. A DisCSP is a constraint satisfaction problem in which the variables and constraints on these variables are distributed among independent but communicating agents. One important motivation behind the DisCSP paradigm is that it allows agents to keep their constraints privately while permits the solution to be solved globally. Most current DisCSP algorithms [YDIK92, BKGS01, Yok95] focus on propositional satisfiability. Formally, distributed constraint satisfaction can be defined as following:

A *distributed constraint satisfaction* P=$\langle$V,D,C,A$\rangle$ is a problem defined on a set of variables V=$\{x_1,...,x_n\}$, a set of discrete finite domains for each of the variables D=$\{D_1,...,D_n\}$, and a set of constraints C=$\{C_1,...,C_m\}$ 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 holds a constraint C_q, it also must hold 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.

Many studies have been done in proposing algorithms for solving DisCSPs, such as Asynchronous Backtracking (ABT) [YDIK92], Asynchronous Weak-Commitment (AWC) search [Yok95], and Asynchronous Forward Checking (AFC) [MZ] to name a few. ABT is normally considered as the base algorithm for others. Two popular problems in the DisCSP application domain are Meeting Scheduling and SensorCSP [BKGS01]. In the Meeting Scheduling problem, multiple agents negotiate to find a meeting time that can satisfy all agents' personal constraints. In the SensorCSP there are a number of sensors modeled as agents and targets where a target is tracked if k sensors are tracking it at the same time. The sensors have to cooperate to track all the targets with their own constraints that a sensor can only track one target at a time. In this paper, we introduce another problem that can be modeled and solved with DisCSP - the problem of QoS composition. This problem is of a great interest to the agent community.

3 Modeling QoS Composition Problem as DisCSP

In the QoS composition problem, multiple service providers participate in the service composition process. Each provider has its own constraints on QoS levels of many parameters such as resource limitations, business rules, organizational policies and service composition structures. The QoS levels may be continuous but in most cases the service providers and consumers are normally interested in the discrete values of the QoS levels; for example, discrete values of cost, disk space and response time. In the current Web service model the local constraints of each service provider are normally revealed through different offered *classes of service*. However, we believe that in many situations, service providers will most likely keep their local constraints private until some negotiations are made between the two parties. Thus, in our model, the local constraints of each provider are private within that particular provider only.

When a QoS service composition problem is transformed to a DisCSP, it is reasonable that each service provider is mapped into an agent in a constraint network. Similarly, each QoS parameter is mapped into a variable in V as defined in the DisCSP definition above; and the set of providers' constraints is mapped into the constraint set C. Figure 3 shows an example of five agents A_1, A_2, A_3, A_4, A_5 together forming an Attraction Service composite service. Each agent has its own constraints on QoS.

In Figure 2, c(S) and t(S) define the total cost and response time of a composite service S while $c_{i:own}$ and $t_{i:own}$ define the cost and response time introduced by the agent A_i itself. The c variable can take a price between \$1 and \$100 (USD) and the t variable can take an integer value between 1 and 100 (ms) for all services. We allow shared variables in our model, QoS variables of a sub-service provider are shared with the composite service provider. For example, c(FindAttraction) and t(FindAttraction) are shared between A_1 and A_2. It is important to note that the above scenario is a very simple description of how

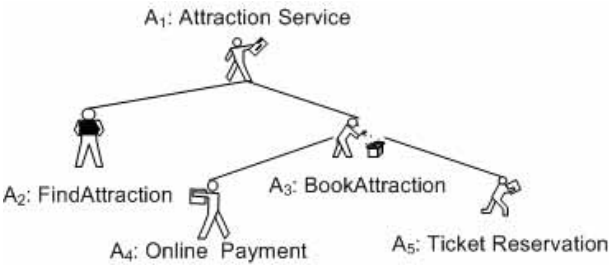

Figure 1: A scenario of a composite Attraction Service which has a nested composition structure with the Book Attraction service offered by A_3.

a QoS composition problem may look like. In a more complex example, to negotiate on the values of cost and response time, A_3 and A_4 may need to negotiate on other variables as well. In other words, the set of variables at each agent may not represent the same set of QoS parameters. Also, the example in Figure 1 gives an impression of a tree structure. However, this is not always true as many consumers can use the same Web services, especially stateful Web services like Grid services.

To illustrate the differences when applying DisCSP techniques, we first consider the two most commonly used approaches for QoS composition problems [LNZ04, ADK+05, JRGM04, GNCW]:

- In the first approach, QoS composition is solved incrementally. Firstly, A_1 negotiates with A_2 and A_3 for attraction finding and booking. A_2 agrees to provide a *FindAttraction* service with, for example, $\{$1ms, \$10$\}$ for $\{$response time, cost$\}$. A_3 agrees to provide its *BookAttraction* service with, for example, $\{$4 ms, \$10$\}$ because it is confident to find sub-providers to support these values. After that, A_3 contacts and negotiates with A_4 and A_5. If A_4 agrees on $\{$1ms, \$5$\}$ and A_5 agrees on $\{$3ms, \$5$\}$ for their services respectively, a solution is found. Otherwise, A_3 have to find a substitute of A_4 and A_5.

- In the second approach, synchronous backtracking is used to solve the QoS composition problem. A1 proposes some cost and response time values to A_2 and A_3. For example, it proposes $\{$1ms, \$15$\}$ to A_2 and $\{$4ms, \$5$\}$ to A_3. A_2 accepts the proposal while A_3 subsequently negotiates with A_4 and A_5 before responding to A_1. In the negotiation, if A_4 and A_5 cannot satisfy with any proposals from A_3; A_3 backtracks to A_1 with a refusal on $\{$4ms, 5 USD$\}$ values. A1 then has to negotiate for another value with A_2 and A_3. The above steps are repeated until A_4 and A_5 can agree on some proposals from A_3.

It can be seen that both approaches are synchronous, i.e. each agent has to wait for responses from other agents before it can proceed. Also, each agent is only aware of its local composition problem which is formed whenever a request from a client arrives. For example, A_3 is only aware of a composition problem when it receives a proposal from A_1.

A_1: Attraction Service
c(AttractionService) =
c(FindAttraction) + c(BookAttraction)+ $c_{1:own}$
t(AttractionService) =
t(FindAttraction) + t(BookAttraction)+$t_{1:own}$
c(AttractionService) $<$ \$20
t(AttractionService) $<$ 5ms
A_2: Find Attraction
$c_{2:own} >$ \$5
$t_{2:own} >$ 1ms
$c_{2:own}$ = \$5 + (4ms -$t_{2:own}$) x \$1/ms
A_3: Book Attraction
c(BookAttraction)=
c(OnlinePayment) +c(TicketReservation) +$c_{3:own}$
t(BookAttraction)=
t(OnlinePayment) +t(TicketReservation) +$t_{3:own}$
A_4: Online Payment
$c_{4:own} >$ \$2
$t_{4:own} >$ 1ms
A_5: Ticket Reservation
$c_{5:own} >$ \$3
$t_{5:own} >$ 1 ms

Figure 2: Illustration of different constraints held by each agent

To address this limitation, our approach allows every agent to be aware of the global QoS requirement, and encourages them to take part in the solving process asynchronously as soon as possible. In order to do this, we propose that functionality composition is carried out before QoS composition. In this way, a functional composite Web service is formed conceptually before any negotiations on QoS levels are made and hence a set of service providers involved in the composition is determined. This model is especially suitable for composing services with different classes of QoS. In the functionality composition phase, when an agent receives a problem *id* from its consumers, it generates a new problem *id* if necessary, and then forwards this to its providers to inform them that there is a global QoS problem which needs to be solved. In particular, a problem *id* received in combination with the sender address creates a unique context to identify a particular global QoS problem.

4 Composite Service Structure as Constraints

As stated in the previous part, constraints can be formed from different items such as business rules, organizational policies, or composition structures. Translations from business rules and organizational policies into QoS constraints may vary from organizations to another since these rules and policies can be represented and interpreted differently. In this

No.	Workflow Pattern (BPEL pattern)	QoS Composition Pattern
Basic Control Flow Patterns		
1	Sequence (Sequence)	Sequence
2	Parallel Split (Flow/Link)	AND Split
3	Synchronization (Flow/Link)	AND Join
4	Exclusive Choice (Switch/Link)	XOR Split
5	Simple Merge (Switch/Link)	XOR Join
Advanced Branching and Synchronization Patterns		
6	Multi-choice (Link)	OR Split
Structural Patterns		
7	Implicit Termination (By Default)	
Patterns Involving Multiple Instances		
8	M.I. Without Synchronization(Flow)	AND Split and AND Join
State Based Patterns		
9	Deferred Choice(Pick)	XOR Split
10	Interleaved Parallel (Serializable Scope)	Sequence
11	Cancel Activity (Terminate)	

Table 1: Workflow patterns to QoS composition patterns

	Sequence	AND Split/Join	OR Split/Join
Cost	Sum	Sum	Max
Response Time	Sum	Max	Min

Table 2: Formulas of constraints for simple composition patterns of Sequence, AND and OR

section, we show how constraints can be formulated from the composite structures of services offered at each agent. For examples, how the first two constraints in A_1 and A_3 in Figure 2 were constructed. These constraints represent the relationships among QoS parameters of a composite service and of its component services. In this paper, we use the

```
CONSTRUCT-CONSTRAINT(qos-name, activity )
as XPath
1 activity-type ←get type of the activity
2 if activity-type ∈
{“assign”, “throw”, “wait”,
“empty”, “scope”, “compensate”, “terminate”}
3 return null
4 if activity type ∈{ “invoke”, “receive”}
5 service-name ← get name of the service
invoked by activity
6 return Xpath(service-name)
7 sub-activities←get component activities from activity
8 qos-pattern-name←get the QoS
composition pattern equivalent to activity
9 Xpath-array ← {CONSTRUCT-CONSTRAINT(
sub-activities[i])}
10 return F(qos-name, qos-pattern-name, Xpath-array)
```

Figure 3: Constraint formation algorithm

term “QoS constraint formulation” to describe the process of finding these constraints/relationships from a composition input. The composition is presented in

BPEL4WS (the Business Process Execution Language for Web services) format. BPEL4WS is selected because it is a popular Web service composition language supported by different Web service vendors. Our approach bases on important work in [vdAtHKB03, vdA03, JRGM04]. In [vdAtHKB03] and [vdA03], the authors conduct a survey and collect a set of workflow patterns which have been used in workflow languages today, including Web service composition languages. In [JRGM04] the authors map these workflow patterns into QoS composition patterns. We combine these results to form a mapping from BPEL4WS into QoS composition patterns as shown in Table 1. It is worth noting that Table 1 only presents composition patterns supported by BPEL4WS. Our QoS constraint formulation process consists of the following three main steps:

1. Construct constraint formulas for QoS composition patterns: Sequence, AND (Split/Join), XOR (Split/Join) and OR (Split/Join).
2. Iteratively decompose the composite structure through BPEL workflow patterns into smaller sub-compositions until this cannot be done any further.
3. Iteratively re-construct the constraints in a composition from constraints of its sub-compositions. The reconstruction uses formulas available in step 1 above.

In reference to other efforts on QoS aggregation [JRGM04, LNZ04, GNCW, Men04], our constraints can be considered as means to compute the QoS aggregations. The difference is that we construct the formulas to aggregate QoS instead of compute a value. Formulas

```
<flow name= "Shipping">
<sequence>
<invoke name= "ShipmentAir"/>
</sequence>
<sequence>
<invoke name= "ShipmentWater"/>
</sequence>
<sequence>
<invoke name= "ShipmentLand"/>
</sequence>
</flow>
```

Figure 4: An example of BPEL activity

for simple composition patterns in step 1 with different QoS attributes can be found in [JRGM04]. Table 2 presents a partial list of these.

We propose a dynamic programming algorithm to construct constraints in XPATH format from a BPEL4WS input file. In our algorithm, the BPEL4WS composition is represented as a *ProcessDef* object which has a base *Activity* object. The *Activity* class represents all possible activities available in BPEL4WS processes such as "receive", "reply", "invoke", "sequence" and "switch". Hence, an *Activity* object is composed of other *Activity* objects. XPath class represents an XPATH expression. The pseudo-code for our constraint formulation process is listed in Figure 3. The main idea behinds the *Construct-Constraint* algorithm in Figure 3 is to iteratively decompose a Bpel activity and apply equivalent QoS composition formulas (Table 2) at each iteration. New variables are added whenever "invoke" and "receive" activities are encountered. In the line 10, F is a constraint function (constructed from Table 2) for QoS composition patterns. For an example, a composite service with a structure listed in Figure 4 after passing through the *Construct-Constraint* function will produce t(Shipping)= max(t(ShipmentAir), t(ShipmentWater), t(ShipmentLand)) for the input of *qos-name* as *response time*.

5 AAS4QoS Algorithm

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 problems as discussed at the beginning of this paper. Often the techniques used in these algorithms combine tree-search with backtracking, look-ahead, and backjumping. However, there are some characteristics that make the QoS composition problem different from the Meeting Scheduling and Sensor Network problems:

- Each agent holds more than one variable.

- Each agent holds a set of interested QoS parameters and the variables that represent these parameters.
- Local constraints in QoS problem can be very complex.
- Provider agents are not willing to reveal their consumer agents' addresses to others.
- Agents may want to hide private information from others as much as possible.

In searching for a suitable DisCSP algorithm, those above characteristics are the most important criteria for us. Whilst most algorithms such as ADOPT and IDIBT can be extended so that one agent can hold more than one variable, substantial effort is required for this and for handling complex private constraints. In addition, the back-jumping technique in some algorithms [MZ] require undesirable revelations of agent consumers' addresses to others. Asynchronous Aggregate Search (AAS) [SF05, SSHF00] is a good candidate since it allows one agent to maintain a set of variables and these variables can be shared. 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. This reduces the number of backtracks. However, private information is revealed more in AAS than in ABT because of the aggregation,. Using trusted servers where critical information of organizations is hosted to prevent privacy loss is not very practical for Web services. In this work, we rate privacy as the most important issue. Also, our goal is to develope a simple but effective algorithm in which agents can exploit special characteristics of the QoS composition problem. As such, we propose an extension of AAS with three following enhancements and modifications:

- Incorporating local CSP solvers into agents to solve complex private constraints.
- Making use of common characteristics of QoS parameters to speed up the solving process.
- Reducing the aggregation in AAS into one value per variable to reduce privacy loss.

Algorithm 1: AAS4QoS message processing

```
1  when received ok(<x_j,s_j,h_j>) do
2      if history(x_j) invalidates h_j then
3          return;
4      add <x_j,s_j,h_j> to agent-view;
5      update nogood list store;
6      check-agent-view;
7  when received nogood(A_j,¬N) do
8      update agent-view for any new assignments in ¬N;
9      send setup channel request to A_j not connected agents in A^- which hold unknown
       variables in ¬N;
10     if ¬N is invalid then
11         return;
12     insert ¬N into the nogood list;
13     update nogood list store;
14     check-agent-view;
15     send ok? to the nogood sender if the new assignment for this agent is unchanged;
16 when received channel(<A_j, x_j,pa_j>) do
17     setup relay-channel for A_j;
```

Similar to AAS, in our algorithm, agents are assigned with priorities so that they can be arranged in order. Every agent proceeds with a similar process. Information about the outside world learnt by each agent is stored in an agent view and a nogood list. An agent view is a set of values that the agent believes to be assigned to the variables belonging to the higher priority agents. Agents exchange assignments and nogoods. An assignment has the form $\{\bigwedge_{m}^{i=1}(x_i{=}a_i),h_j,pa_k\}$ which indicates that the variable x_i is assigned a value a_i. h_j is the message history [SF05], pa_k is the pseudo-agent address and will be explained later. The assignment is an AAS aggregation of single value. A nogood list holds the assignments of values to the variables during the solving process which cause inconsistency. '*ok?*', or '*nogood*' messages are used as in AAS. The '*ok?*' message is used to inform the lower priority agents of new value assignments. A nogood is used to backtrack the assignment that causes inconsistency between constraints. In Algorithm 1, we use A^+ to denote the set of agents that are linked to A and have priorities higher than the priority of A. Similarly, A^- is denoted as the set of agents that are linked to A and have priorities are lower than the priority of A. V^+ is the set of variables the agent share with A^+, and V^- is with A^-. As illustrated in Algorithm 1, when an agent receives an '*ok?*' message, it validates the message and adds the assignment in the message into its local view. It then checks the local view consistency. To handle the complexity of local constraints, we use a local CSP solver inside each agent. In the *check-local-view* procedure, first the agent updates any new assignment. It detects if there is any unknown variable in the assignment. If there is such a variable then the agent requests to add link to the agents who hold this variable so it can communicate directly with this agent. In the *check-local-view*, if the agent view is inconsistent then the agent searchs for

new value assignments for its variables. If a solution is found, the agent sends new 'ok?' messages with the assignments to the lower priority agents; otherwise, it sends a *nogood* message back to the *'ok?'* message sender. It also creates the explanation for this nogood and sends along with the nogood message. The detailed description of the extended AAS algorithm and performance analysis is a subject of another paper (under review).

Algorithm 2: AAS4QoS check-agent-view

```
1  if agent-view and current-aggregate are inconsistent then
2      V =localCSP.solve(agent-view,nogood-list, local-constraints);
3      if V=∅ then
4          backtrack
5      else
6          reset current-aggregate;
7          forall a∈ V do
8              if a is new for A+_k then
9                  append new history and send ok message to A+_k;
10                 current-aggregate = current-aggregate∩a;
11             else
12                 if a is needed then
13                     current-aggregate=current-aggregate∩a;
```

In the context of Web service composition, QoS parameters have particular characteristics that normal constraint variables have not. Agents (service providers) often have had their constraint preference as a monotonic function over a QoS value. In other words, there is a preference operator $\preceq$ defined between any two values $a^{(1)}$ and $a^{(2)}$ in the domain of variable x. If $a^{(1)} \preceq a^{(2)}$ then we say that the agent prefers $a^{(2)}$ to $a^{(1)}$ for x. In addition, this operator can also be defined over two set of assignments as following: $\bigwedge_m^{i=1}(x_i=a^1{}_i) \preceq \bigwedge_m^{i=1}(x_i=a^2{}_i)$ if $\forall$i=1..m $a^1{}_i \preceq a^2{}_i$. The preference for A2 in Figure 2, for example, can be that
t(FindAttraction)=2$\bigwedge c$(FindAttraction)=5$\preceq$
t(FindAttraction)=1$\bigwedge$ c(FindAttraction)=4
While the AAS4QoS algorithm follows the principles of AAS, the local CSP solver uses a new heuristic for selecting new values. When a new assignment is generated a heuristic criterion could be to choose the value with not less preference (compared to the previous rejected assignments) to the lower priority agent.

6 Implementation

In this part, we describe our initial agent-based implementation of a framework for DisCSP with our toolkit J4WSM (Jade for Web Services Management) to support our proposal. J4WSM is a re-factor of our previous toolkit called WS2JADE that integrates Jade agents with Web services [Ngu05], towards WS management. J4WSM, which consists of a Jade

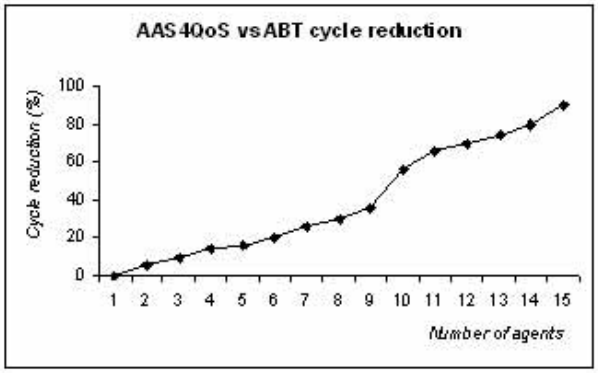

Figure 5: AAS4QoS versus ABT

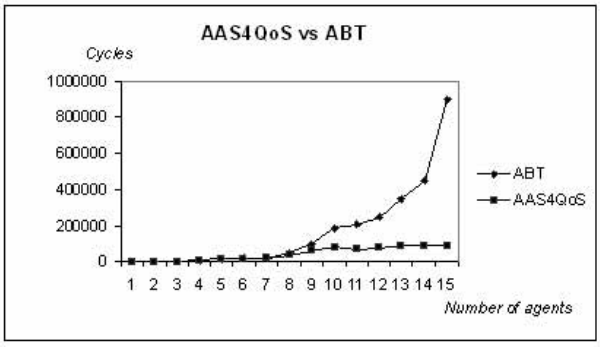

Figure 6: AAS4QoS versus ABT

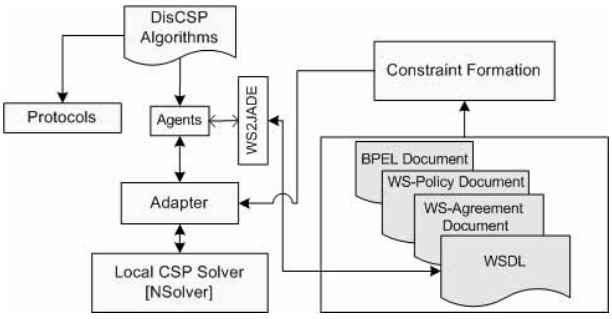

Figure 7: Different J4WSM system components

agent platform and other utilities including WS4JADE, runs as a service inside a J2EE container. In this initial version, J4WSM runs under JBoss. J4WSM is similar to Blue-Jade [CB02] but targets Web Services Management particularly. J4WSM is designed to use services instead of making API calls.

The main components in J4WSM consists of three main modules: constraint formations, local CSP solver and DisCSP protocols. The constraint formation module is to construct and keep the constraints available at each provider. Constraints in J4WSM are presented in XPath expressions. At this version, constraints are created from composite service structures (in BPEL format) by using the algorithm specified in Section 3. In J4WSM, we do not specify any particular CSP solver engines. Different CSP solver engines can be used as long as there is an adapter which can translate XPath expressions into the CSP solver languages. For our experiment, we use NSolver [NSo05] for the local CSP solver. Whenever a provider needs to solve the local constraints, it invokes NSolver to find a solution. We have developed an adapter to translate XPath expression into NSolver on the fly. This adapter can be downloaded from [XPa05]. In this version, J4WSM only supports one DisCSP protocol which corresponds to the AAS4QoS specified in the previous part. This protocol is developed as an interaction protocol in Jade. Each provider in the DisCSP is represented by a Jade agent. These agents must understand a protocol and agree to use it before the solving process can start. Different components in J4WSM are presented in figure 6. Figure 5 shows performance of synchronous backtracking versus AAS4QoS for samples of QoS composition problems. The graph shows the average of 10 trials for each value in the horizonal axis. The structure of the composition in this experiment is built incrementally by adding one agent each time from the first agent A1 until a balanced binary tree with depth =4 (i.e there are 15 agents in total) is formed. The experiment is carried out for two parameters: cost and response time. Each agent A_k has a business rule: $c(A_k) \geq \frac{max\{T_0 - t(A_k), 0\}}{T_0} C_T$ in addition to the constraints deduced from the structure of its own composite service: $t(A_k) = t_{own}(A_k) + \sum_{\forall A_q \in A^+{}_k} t(A_q)$, $c(A_k) = c_{own}(A_k) + \sum_{\forall A_q \in A^+{}_k} c(A_q)$ in which $t(A_k)$ and $c(A_k)$ are response time and cost of the composite service at A_k, $t_{own}(A_k)$ and $c_{own}(A_k)$ are response time and cost introduced by A_k's own service respectively. In the experiment, the domain of the cost variable for an agent with depth=i is $[1..2^{5-i}]$ and the domain of this agent's time response variable is also $[1..2^{5-i}]$. T_0=4ms and C_T=1\$. The purpose of this experiment setup is to mimic a real scenario of Web service composition and demonstrate the applicability of DisCSP technique into the QoS composition problem. It can be observed from Figure 5 and 8 that AAS4QoS outperforms ABT algorithm which in turn has a better performance than the synchronous backtracking algorithm [YDIK92]. Figure 5 and 6 show AAS4QoS improvements in term of processing cycles. For each agent, one cycle consists of reading all incoming messages, invoking its CSP solver to find a solution and sending messages [Yok95]. It can be seen that the reduction of cycle increases with the number of agents and reaches around 90% for 15 agents.

7 Conclusions

This paper proposes a new approach of DisCSP application into the QoS composition problem. We describe how the problem can be modeled in DisCSP by suggesting that functionality compositions can be carried out before QoS compositions. An algorithm to construct constraints from the composition topology is proposed. We also develop an enhanced version of AAS for multiple variables and a new heuristic for distributed agents to exploit QoS parameters' characteristics. Our future work will concentrate on a framework which allows agents to exchange not only assignments but also possible constraints during the solving process, as a part of J4WSM toolkit. We believe that this is beneficial for the QoS composition problem and useful in developing new DisCSP algorithms for the solving process.

References

[ADK+05] Vikas Agarwal, Koustuv Dasgupta, Neeran Karnik, Arun Kumar, Ashish Kundu, Sumit Mittal, and Biplav 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.

[BKGS01] Ramon Bejar, Bhaskar Krishnamachari, Carla Gomes, and Bart Selman. Distributed Constraint Satisfaction in a Wireless Sensor Tracking System. In *Workshop on Distributed Constraints, IJCAI*, 2001.

[CB02] Griss M. Cowan, D. and Burg B. BlueJade-A service for managing software agents. Hp technical report, HP, Murray Hill, New Jersey, 2002.

[GNCW03] X. Gu, K. Nahrstedt, R. Chang, and C. Ward. QoS-Assured Service Composition in Managed Service Overlay Networks. In *Proceedings of the IEEE 23rd International Conference on Distributed Computing Systems*, 2003.

[JRGM04] Michael C. Jaeger, Gregor 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.

[LNZ04] Yutu Liu, Anne H. Ngu, and Liang 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.

[Men04] Daniel A. Menasce. Composing Web Services: A QoS View. *IEEE Internet Computing*, 8(6):88–90, 2004.

[MZ03] A. Meisels and R. Zivan. Asynchronous forward-checking on DisCSPs. In *Proceedings of the Distributed Constraint Reasoning Workshop, IJCAI*, Acapulco, Mexico, 2003.

[Ngu05] Xuan Thang Nguyen. Demonstration of WS2JADE. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 135–136, New York, NY, USA, 2005. ACM Press.

[NSo05] *NSolver home page*. www.cs.cityu.edu.hk/ hwchun/nsolver/, 2005.

[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.

[SSHF00] Marius-Calin Silaghi, Djamila Sam-Haroud, and Boi Faltings. Asynchronous Search with Aggregations. In *AAAI/IAAI*, pages 917–922, 2000.

[vdA03] 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.

[vdAtHKB03] 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.

[XPa05] *XPath Adapter for NSolver*. www.it.swin.edu.au/centres/ciamas/tiki-index.php?page=xpath2nsolver, 2005.

[YDIK92] Makoto Yokoo, Edmund H. Durfee, Toru Ishida, and Kazuhiro Kuwabara. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. In *International Conference on Distributed Computing Systems*, pages 614–621, 1992.

[Yok95] Makoto 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.