

An agent based QoS conflict mediation framework for Web services compositions

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

Abstract—Web services technology is prevailing for business-to-business integration due to its well defined infrastructure enabling interoperability among heterogeneous applications. However, this interoperability promise also poses a difficulty in building a Web service management framework which can work across organizational boundaries. In this paper, we argue that existing Web service management systems are inflexible in the way they handle QoS violations of a composite service. We suggest to use an intermediate step, called QoS conflict mediation, to make existing Web service management systems more flexible. Our mediation approach is based on a combination of two new ideas. Firstly, we propose to use techniques from the AI field of Distributed Constraint Satisfaction to intelligently mediate any QoS conflicts between services in compositions involving multiple service providers. Secondly, we propose a novel monitoring system, based on cryptography, to verify the conformance of service providers to the specification of a selected DisCSP algorithm - the Asynchronous Aggregate Search (AAS). This enables our DisCSP based QoS mediation to be used in the real Web services environment where full collaboration between providers may not always be guaranteed.

I. INTRODUCTION

Web service technology has emerged as a popular interoperable tool for distributed applications. It exposes the resources and applications in an existing infrastructure via a standard interface and hence makes the infrastructure more accessible, reusable, and composable. Different Web services can be combined to form a new value-added Web service which is referred as a composition or a composite Web service. Composite Web service management is the process of ensuring the satisfaction of functional and non-functional (i.e. QoS) requirements of a composite Web service during its execution. Composite Web service management process in general includes of component service selection, execution and monitoring, and replacement of contract violated services. Management of a composite Web service is difficult because its managerial system must work across organizational boundaries. Here we refer to the Web services QoS as non-functional indicators of a Web service's performance. These parameters can be quantitatively measured, such as availability, security level, and response time.

In this paper, we argue that current Web service management framework, e.g. [1], [6], [3] are too rigid in the way that they handle QoS violations. In particular, they always

replace a violated or underperformance service with a new one. As a contribution, we propose an intermediate step, called *QoS conflict mediation*, to gracefully handle the violations before any replacement may take place. In this step, the service providers mediate their QoS conflicts using Distributed Constraint Satisfaction algorithms. We select the AAS (Asynchronous Aggregate Search) algorithm for the mediation since it is suitable for modelling collaborative negotiation. Our second contribution is the introduction of a novel verification mechanism, based on cryptography, for AAS. This explicit verification mechanism eliminates the impractical assumption of fully collaborative agents (i.e. providers) in DisCSP and hence enable our DisCSP based QoS mediation to be practically used in the real Web service environment where full collaboration between providers may not be always guaranteed.

The rest of the paper is organized as follows. In Section III we discuss and present a formal description of the QoS conflict mediation. We review the AAS (Asynchronous Aggregate Search) algorithm for its application in the problem of QoS conflict mediation in Section IV. We describe our novel monitoring system in Section V. Finally, conclusions and future work are discussed in Section VII.

II. RELATED WORK AND PROBLEM DEFINITION

There have been a number of works focused particularly on QoS management for Web services. In [4], [7] the author propose a QoS management framework which performs the refinement of existing Web service services through continuous monitoring of service execution. A QoS adaptation mechanism, based on Service Level Agreement, is presented in [1]. In these works an under-performance Web service, if detected, is always replaced by better ones as they assume that there are many available Web services which can offer similar functionalities. QoS conflict mediation is closely related to QoS composition of which current work can be found in [3] and [6]. In [6] 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.

In general, the management process in these approaches consists of three major steps: *composition planning*, *service discovery and selection*, and *execution and monitoring*. The

relationships between these steps are depicted in Figure 1. As can be seen, these steps can be executed iteratively. In particular, if a contract violation is detected, replacements of the violated services are taken place. The replacements lead to a re-discovery and re-selection of component services.

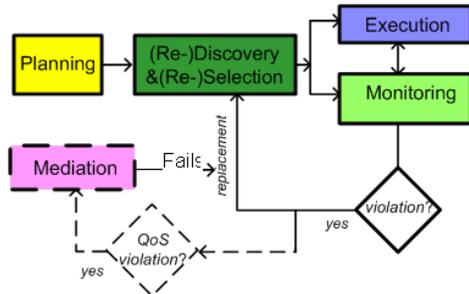


Fig. 1. Different steps for a QoS management framework. We introduce a mediation step before any replacements

We argue that the above approaches are too rigid in the way violations are handled. In particular, they immediately replace a violated component service provider if the QoS contract of this provider is violated. In this paper, we address this inflexibility by introducing an intermediate step, called *QoS conflict mediation*. In particular, if the violation is QoS related, this step is invoked before any replacement (see Figure 1). In this mediation step, every component provider collaborates to solve the QoS conflicts, and update their contracts if necessary. Only when the conflicts cannot be resolved, the violated providers are then replaced. If successful, this *mediation* mechanism is advantageous due to the following reasons:

- It may be costly and difficult to replace a violated service, especially when there are very few available services which can offer similar functionalities to the violated one and when certain levels of reputation and trust (which can only be learnt from a long term collaboration) of the new services are required. QoS mediation offers a new possibility to avoid this.
- A QoS mediation causes less disruption to the execution of a composite service as compared to service replacement. This is due to the inflexibility of most current Web services composition languages such as BPEL and WSCDL in changing the structure of a composition. These languages require that the end point addresses of all component Web services must be specified and statically bounded to a composition at the design time.

It is important to note that our *QoS conflict mediation* process is carried out not only by component service providers engaged in the composition but also providers of other compositions which are related to this composition. This will become clearer in the next section.

III. THE QoS MEDIATION PROCESS

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 levels contributed to these compositions by the component services. The overall idea of our QoS mediation process is that if a component service violates the requirements of its QoS levels, compensations for this violation are sought from other related services.

For a motivation example to illustrate how the *QoS conflict mediation* can solve QoS conflicts, we refer to Fig.1. This figure shows a scenario of five component Web services for travel information: *Mel-Transport*, *Mel-Attraction*, *Syd-Transport*, *Syd-Attraction*, and *Aus-Weather* which make up three composite online booking services with their e2e QoS shown in the right hand side. For the sake of clarity, we assume that response time and cost are our only considered QoS parameters. Also every composition is a sequential combination of its component services and hence its e2e response time can be computed as a sum of the component services' response time. To summarize, QoS violations of a composition can

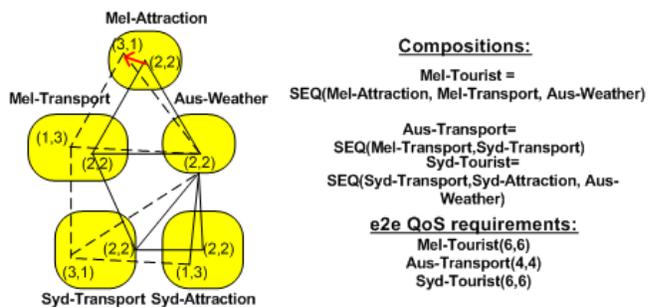


Fig. 2. An example of QoS management of multiple compositions. Initially each component service initially has the response time of 2(ms) and the cost of 2(\$), denoted as (2,2) in the Figure. Suppose that a violation happens to the *Mel-Tourist* service and this violation is caused by a change in the response time of the *Mel-Attraction*, which is now 3(ms). Due to this degradation, i.e from 2(ms) to 3(ms) delay, *Mel-Attraction* drops its cost (e.g. pays the penalty and hence the cost is reduced) to 1(\$). This violation is illustrated as a move from (2,2) into (3,1) by the *Mel-Attraction*. It can be handled as shown by the shift from the continuous lines to the dotted lines.

possibly be fixed by providers in related compositions. It is worthy to note that the changes of response time made by *Mel-Transport* cannot always fix the violation of *Mel-Attraction*. This is due to different factors. Temporal ordering of service executions is an example. If the *Mel-transport* is executed before *Mel-Attraction* then *Mel-transport* cannot repair *Mel-Attraction's* violation since it has been completed before the *Mel-Attraction* starts. By completing execution, *Mel-Transport* introduces a new constraint on the value of its response time which can no longer be changed after that. In relation to a service execution, we call a constraint is *fixable* at an instance of time if this constraint can be changed, e.g. contract constraints of not-yet executed services. Otherwise, it is called *unfixable*.

In order to formalize the relationships between the QoS levels of component services, we note that the QoS requirements of the composite services can be considered as constraints. For details of how these constraints can be formulated (as a

restriction on QoS values of component services) we refer the QoS aggregation work in [5]. These constraints are *shared* among service providers which engage in the composition. In addition to these *shared* constraints, each provider has its own service constraints. 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 or all other providers) or *private* respectively.

Constraints can also be categorized by having different scopes: Constraints known only by a single provider are *own* constraints. *Contract* constraints are terms in a contract to specify the level (i.e. value) of a QoS parameter a provider must satisfy. Constraints with a composition scope are the *e2e* QoS requirements of a composite service. It can be seen that satisfactions of all *contract* constraints leads to satisfactions of all *composition* constraints but not visa versa.

In general, from the service providers' perspectives, there are three types of QoS constraints: *composition*, *contract* and *own* constraints. They can be either *private* or *shared*, and *fixable* or *unfixable*. Apparently, *contract* constraints of completed services are *unfixable*.

We present an algorithm, based on these different constraint types, for the QoS mediation procedure in Algorithm 1. This algorithm is invoked every time a violation is detected. The algorithm returns either *true* or *false*. If *true* is returned, the mediation is successful and no replacement is required. Otherwise, violated services are replaced normally, e.g. as in [1], [6], [3].

Algorithm 1 QoS Mediation

- 1: collect all *contract unfixable*, *composition* and *own* constraints into the set C
 - 2: communicate with other providers and search for new QoS values which satisfy all constraints in C , constraints' visibilities are kept in the search.
 - 3: **if** a solution is found **then**
 - 4: form contracts with new satisfied QoS values
 - 5: **else**
 - 6: return *true*
 - 7: **end if**
 - 8: return *false*
-

Algorithm 1 can be explained as follows. In principles, whenever a QoS violated is detected, except *contract fixable* ones, all constraints are collected and the provider may collaborate with other providers to find new QoS values which satisfy all these constraints (line 2). These values can be used to form new contracts to replace existing contracts and hence update the *contract* constraints. If these values can be found then the algorithm returns *true*. Otherwise, it returns *false* to trigger a possible service replacement. The search in line 2 is the main part of the QoS mediation algorithm. We call this search the *mediation solving process* which is detailed in the next section.

IV. DISTRIBUTED ASYNCHRONOUS SEARCH FOR QoS MEDIATION

The *mediation solving process* is a major part of our QoS conflict mediation algorithm (Algorithm 1). Here we argue that DisCSP techniques can be used for effective *mediation solving* due to the following reasons:

- Distributed nature of the Web service environment and the engagement of many participants in compositions suggest that a distributed approach is best suited.
- Constraints in the QoS *conflict mediation* process can be both *private* and *shared*. Distributed constraints with different visibility levels have been a main focus of DisCSP techniques.

To apply DisCSP techniques in the QoS mediation solving process, each service provider in the set of related compositions 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; and the set of providers' constraints is mapped into the network's constraint set. From now on, we will use the terms *service providers* and *agents* interchangeably. The searching problem for the QoS *conflict mediation* can be considered as an instance of DisCSP problems.

In the rest of this paper, we focus on two aspects: finding a suitable DisCSP algorithm for the QoS conflict mediation and addressing the DisCSP assumption in which providers are totally collaborative. Many works on DisCSP algorithms have been published recently. 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 conflict mediation 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 desirable. In looking 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 original DisCSP model [9] and most of the solving algorithms focus on shared constraints instead of private constraints. A notable exception is Asynchronous Aggregate Search (AAS) [8] that allows one agent to maintain a set of variables and these variables can be shared. Also all constraints are private in AAS (shared constraints can be modelled as duplicated private constraints). AAS is suitable for negotiation and hence is selected in our approach. We describe AAS next.

Asynchronous Aggregate Search

Here we briefly introduce AAS in the Web services context. A complete explanation of AAS can be found in [8] 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 [9]. 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 the 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 and thus improves the performance. At the beginning, AAS agents are (randomly) assigned with priorities and generate random assignments (i.e. proposals). Two agents are neighboring if they share some variables. During search, each agent A_i sends a proposal, which has an aggregate (see Definition 2), in *ok* messages to lower priority neighbors or rejections in *nogood* messages to higher priority neighbors.

Algorithm 2 AAS message_processing(m_{in})

```

1: if  $m_{in}$  is an ok message then
2:   update agent_view
3:   check_agent_view
4: else if  $m_{in}$  is a nogood message then
5:   update agent_view
6:   if consequence of the nogood in  $m_{in}$  is not covered by
       other nogood then
7:     send addlink messages to owners of variables which
       are not connected with this agent
8:     add the nogood into the nogood list
9:   end if
10:  check_agent_view
11:  resend ok messages to the nogood's sender if the
       assignment for this sender is unchanged.
12: end if

```

Algorithm 3 AAS check_agent_view

```

1: if current_view is inconsistent with local constrains 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
       constrains and nogoods
6:     update agent_view with the aggregate
7:     send ok messages to lower priority agents
8:   end if
9: end if

```

Definition 1 (Assignment and Aggregate): An AAS assignment is a tuple $\langle x_i, \mathbf{v}_i, h_i \rangle$ in which x_i is a variable, \mathbf{v}_i is a set of values for x_i and h_i is a history of the pair (x_i, h_i) . An AAS aggregate is a list of assignments.

Definition 2 (Nogood): An AAS nogood is a rejection of a previous proposal. A nogood has the form $\neg\Gamma$ where Γ is an aggregate.

In AAS, each agent implements the *message_processing* procedure outlined in Algorithm 2 (a general form of AAS in [8], full nogood recording is assumed) to handle an incoming message m_{in} and generates a set of outgoing messages sent to its neighbors. An execution of the *message_processing* procedure is called a *processing cycle*. The procedure checks whether the information of a partial solution in m_{in} is still compatible with the agent's view. It may invoke a *check-agent-view* procedure to find out a new compatible assignment for the agent's local variables. In particular, if m_{in} is an *ok* message, the procedure *message_processing* updates the agent-view (line 2) before possibly invoking the *check-agent-view* procedure to find new assignments. If m_{in} is a *nogood* message (see Definition 3), the procedure updates its view according to assignments of unknown variables found in the *nogood* content. The agent also try to establish new links with higher priority agents which hold these unknown variables (line 7). The procedure *check-agent-view* is used to find a new instantiation and sends updated values in this instantiation to lower priority neighbors. Inside *check-agent-view*, a local solving process takes place (line 2) to find a new aggregate. In general, the local solving process of an agent A_i takes assignments from its higher priority neighbors and generates aggregates for lower priority neighbors. If the solving process fails, a *nogood* message is sent back to one of the higher priority agent. Otherwise, new assignments are generated by A_i and sent to lower priority neighbors.

V. AAS VERIFICATION

A. Weak Conformance

For a DisCSP algorithm, an agent does not conform to this 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 execution correctness 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' point 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 some 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 3 (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 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.

B. Monitoring Overview and Initial Setup

As presented in the previous sections, DisCSP techniques are good candidates for QoS conflict mediation. However, these techniques assume that every agent fully collaborates with each other by strictly conforming to the DisCSP protocols. This condition may not be realized in an environment like Web services where providers come from different organizations and have different goals. An agent may act differently from the protocol specifications for some purposes. Instead of making the impractical assumption of fully collaborative agents, we propose 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. It can analyze and read different fields of a message but not the values of variables in the messages.

The overall idea of our monitoring system is to capture every incoming message of an agent A_i and simulate A_i 's execution (by following AAS specification) for this input message. The simulation output of messages are used to compare 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 verify the correctness of such a solution reported by A_i . The verification of a CSP solution in general is simpler and requires less resource as compared to a solving process.

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

Definition 4 (Local solution): A local solution of an agent A_i is an aggregate 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.

It can be seen that the aggregate in line 5 of the *check_agent_view* procedure is a local solution. A local solution is temporary since the nogood set is continuously updated after each cycle.

Definition 5 (Initial 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.

On the contrary 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 our monitoring network is as follows. For a DisCSP network of n solving agents $A_i, i = 1..n$, our 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 . AAS verification and solving processes are executed in parallel. Initially, agents are assumed 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* $S(A_i)$ (see Definition 6) using the above secret key and then agent's public key p_{A_i} . These encrypted values are sent to the monitoring agent M_i and are used by M_i as its initial solution set $S(M_i)$.

To better explain the above relationship between $S(A_i)$ and $S(M_i)$, we assume that the set $S(A_i)$ is represented as:

$$S(A_i) = \{ \langle x_{i_1} = S_{i_1}^p, \dots, x_{i_k} = S_{i_k}^p \rangle : p = 1..m \}$$

in which x_{i_1}, \dots, x_{i_k} are variables of A_i and $S_{i_j}^p$ is a set of values for x_{i_j} .

The set $S(M_i)$, which is also the M_i 's valid solution set, can be obtained as:

$$S(M_i) = \{ \langle x_{i_1} = f(S_{i_1}^p), \dots, x_{i_k} = f(S_{i_k}^p) \rangle : p = 1..m \} \quad (1)$$

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

$$f(S_{i_j}^p) = \{ f(s) : \forall s \in S_{i_j}^p \}, j = 1..k \quad (2)$$

$$f(s) = p_{A_i}(k(s)) \quad (3)$$

For examples, 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 leak 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 is 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 they do 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.

C. Monitoring Algorithms

As discussed before, each monitoring agent M_i executes a similar process as A_i does. In particular, the *ver-*

ify_message_processing procedure in Algorithm 4 for M_i is similar to the *message_processing* procedure in Algorithm 2 for A_i . From line 2 to line 13, 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 $m_{out}^{encrypted}$ at line 1. It later compares those messages against its generated messages of m_{out}^{self} at line 14. 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 becomes more clear in the *check_agent_view* (Algorithms 3) and *verify_check_agent_view* (Algorithms 5) procedures. Algorithms 5 show that instead of computing a new local solution at M_i (as in line 5 of *check_agent_view*-Algorithm 3), we require that A_i encrypts and reports its satisfied aggregate to M_i . This encrypted aggregate, 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 due to 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 incurs less processing resources as compared to a solving process.
- If A_i reports no solution, it 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 aggregates found in line 5 of the *check_agent_view*. If A_i does not report its aggregate to M_i , there is no guarantee that there is a correspondence between their aggregates.

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:

$$\forall v_a \in V(A_i), v_a = \langle x_{i_1} = S_{i_1}^p, \dots, x_{i_k} = S_{i_k}^p \rangle \text{ then}$$

$$\exists v_m \in V(M_i), v_m = \langle x_{i_1} = f(S_{i_1}^p), \dots, x_{i_k} = f(S_{i_k}^p) \rangle$$

And vice versa:

$$\forall v_m \in V(M_i), v_m = \langle x_{i_1} = S_{i_1}^p, \dots, x_{i_k} = S_{i_k}^p \rangle \text{ then}$$

$$\exists v_a \in V(A_i), v_a = \langle x_{i_1} = f^{-1}(S_{i_1}^p), \dots, x_{i_k} = f^{-1}(S_{i_k}^p) \rangle$$

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 1: 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 aggregate is a local solution in the same way that is based solely on their agent views, nogood sets and valid solution sets. We say an aggregate Γ_1 *fully contains* another aggregate Γ_2 if for every variable, the value set of this variable in Γ_2

Algorithm 4 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 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:     generate addlink messages for owners of variables
       which are not connected with this agent and add these
       messages to  $m_{out}^{self}$ 
9:     add the nogood into the nogood list
10:  end if
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: end if
14: if  $m_{out}^{self}$  is inconsistent with  $m_{out}^{encrypted}$  then
15:   report AAS violation
16: end if
17: if  $m_{out}^{self}$  is inconsistent with  $A_i$ 's reported solution then
18:   report AAS violation
19: end if

```

Algorithm 5 AAS *verify_check_agent_view*

```

1: if  $A_i$  reports no solution then
2:   if find an aggregate which is consistent with local
   constrains and nogoods then
3:     report AAS violation
4:   end if
5:   backtrack
6: else if  $A_i$  reports a solution then
7:   encrypt the reported solution with  $A_i$ 's public key
8:   if the solution is inconsistent with local constrains and
   nogoods then
9:     report AAS violation
10:  end if
11:  update the agent_view with the solution
12:  generate ok messages for lower priority agents and add
   them to  $m_{out}^{self}$ 
13: else
14:   report AAS violation
15: end if

```

is a subset of that in Γ_1 . We denote this relationship as $\Gamma_1 \supseteq \Gamma_2$. An aggregate Γ , which has the same set of variables as A_i does, is a local solution of A_i iff:

$$V(A_i) \supseteq \Gamma \quad (4)$$

$$\exists s \in S(A_i) : s \supseteq \Gamma \quad (5)$$

$$\exists ! n \in N(A_i) : n \supseteq \Gamma \quad (6)$$

The condition (4) says that Γ must be consistent with the agent view, (5) states that Γ must be consistent with A_i 's local constraints, and (6) confirms that no nogoods' aggregate can be induced from this aggregate. Because these conditions can be checked by matching of the values, they are preserved after f transformation. Due to this and Property 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 2: If A_i does not *weakly conform* to AAS algorithm (specified by Algorithms 2 and 3) then M_i is able to detect this by using Algorithms 4 and 5.

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 5 (line 1 to 10), 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 4 (in line 4) can detect if the input/output of A_i does not *strictly conform* to 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 with 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' is strictly conform 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 Proposition 2 is proved. Also, line 17 of Algorithm 4 disallows A_i from incorrectly reports 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.

VI. CONCLUSIONS

We have discussed in this paper the limitations of current approaches in QoS management of composite Web services and outlined a new approach for more flexible handling of QoS violation in an intermediate step of management called QoS conflict mediation. We have also described the application of AAS algorithm in QoS conflict mediations and proposed a monitoring framework which can verify the conformance of providers to the AAS algorithm specification. This verification enables AAS to be used in the real Web services environment. Our future work focuses on better modelling of providers' behaviour during the solving process, such as different satisfaction and preference levels of a DisCSP solution.

REFERENCES

- [1] R. Al-Ali, A. Hafid, O. Rana, and D. Walker. An approach for quality of service adaptation in service-oriented grids: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(5):401–412, 2004.
- [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] Y. L. Chintan Patel, Kaustubh Supekar. A qos oriented framework for adaptive management of web service based workflows. *Lecture Notes in Computer Science*, 2736:826 – 835, Sep 2003.
- [5] 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.
- [6] 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.
- [7] C. Patel, K. Supekar, and Y. Lee. Provisioning resilient, adaptive web services-based workflow: A semantic modeling approach. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, page 480, Washington, DC, USA, 2004. IEEE Computer Society.
- [8] 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.
- [9] 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.