Modeling and Transforming Abstract Constraints for Automatic Service Composition

Incheon Paik*, Haruhiko Takada*
*School of Computer Science and Engineering, University of Aizu
Aizu-Wakamatsu City, Fukushima, Japan
E-mail: paikic@u-aizu.ac.jp, takada@ebiz.u-aizu.ac.jp

***Abstract*— Automatic service composition typically consists of logical composition to produce an abstract workflow, followed by physical composition to transform it into a concrete workflow satisfying composition properties, such as Quality of service (QoS), preferences, and logic constraints. The composition properties have not been considered together up to now and, because they are abstract during the logical composition phase, they cannot be understood by the physical composer that reads concrete composition properties. This is an impediment to automatic service composition. Therefore, abstract properties in the logical composer have to be transformed to concrete properties for automated physical composition. This research investigates a stack of composition properties for QoS, preferences, and logic constraints considered together and architecture for automatic service composition, along with semi-automatic transformation of intermediate constraints in the architecture. The architecture together with an ontology for transforming constraints, and evaluation of our prototype are described.**

Keywords- Automatic Web Services Composition, Composition Property, Constraint Transformation, Semantic Web

I. INTRODUCTION

Web services are already providing useful APIs on the Internet and, thanks to the Semantic Web, are evolving into the rudiments of an automatic development environment for agents. To further this environment, the goal of automatic Web service composition is to create new value-added services from existing Web services, and the result is more capable and novel services for users.

Automatic service composition usually follows a two-stage procedure: logical composition followed by physical composition [1-3,6-7]. During logical composition, an abstract workflow is created that composes available Web services to fulfill the desired purposes of a new service. The logical composer (LC) is called a process generator. During physical composition, an execution flow of concrete Web services fitted to the user's objective is generated from the abstract workflow created at the logical composition stage [1]. The physical composer (PC) is called a selector[6] or evaluator. The composition procedures [4,5] consider a variety of composition properties to fulfill the goal as completely as possible. The composition properties include the QoS of the resultant composed service [1,11], preference constraints [1,2], interactivity requirements [4,5,10], matchmaking [2], filtering, verification [13], and exception handling [13]. Service composition attempts to find a concrete workflow satisfying the composition properties (mainly constraints, user preferences, and QoS factors) from the abstract workflow.

Research conducted until now deals with composition properties individually and has not considered them holistically. And as composition properties, such as constraints or preferences are specified abstractly via natural language (NL) or an abstract ontology, the properties must be understood during physical composition. In other words, the abstract constraints in the LC have to be transformed to the concrete constraints that can be understood by the PC automatically. In this paper, we formalize the composition properties based on formulas of first-order logic (FOL) and linear temporal logic (LTL). Also, the degree of abstraction of a constraint is formalized for defining the range of transformation. A framework for the semiautomatic transformation of intermediate composition properties in the LC to concrete version is introduced. A schematic of the framework is shown in Figure 1.

II. AUTOMATIC SERVICE COMPOSITION PROCEDURE

As described above, the creation of a new service via Web service composition is a process involving logical composition, concrete candidate service extraction, and physical composition that considers the composition properties to produce a concrete workflow (Figure 1). As the composition properties are abstract in the LC, they have to be transformed into concrete composition properties. The LC creates a plan (or workflow) to reach the goal state using an AI planner or a finite-state machine (FSM) [4]. The planner or FSM generates the abstract workflow automatically with domain knowledge, according to the user's requests. The plan is a vertical flow of abstract services that can achieve the goal.

The PC must find a concrete sequence of Web services from the abstract services of the logical composition, based on non-functional (e.g. quality of service) requirements. It uses linear planning or CSP to select the concrete services that satisfy constraints optimally. The services are then bound together for deploying the newly created composite service.

978-0-7695-3836-5/09 $26.00 © 2009 IEEE
DOI 10.1109/CIT.2009.96

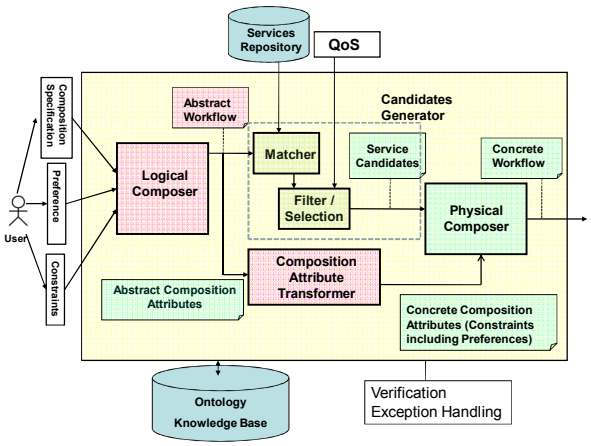


Figure 1. A Framework for Automatic Service Composition

III. COMPOSITION PROPERTY STACK

The composition properties come from users or domain-specific knowledge, and there are four levels of the properties to consider. Level 0 can be handled by the LC and PC, so it is not included in the composition properties that we are discussing. Level 1 describes QoS-related information. These are used to generate candidate services for physical composition. Level 2 deals with domain-specific constraints or preferences. Level 3 deals with constraints and preferences beyond specific domains.

A. Process

The processes resulting from the logical composition consist of atomic processes. OWL-S describes the details of atomic processes very well. An atomic process consists of a process ID and IOPEs (inputs, outputs, preconditions, and effects). The process ID can belong to a specific domain in a process ontology. The ontology for the process can give useful information to extract composition information by inference on the domain ontology.

B. QoS Information

The QoS specifications of a Web service characterize its performance and other qualitative or quantitative aspects. So that the suppliers of services can understand each other's QoS terms, a common understanding must be reached on the meaning of the terms. An ontology can be used to represent and explicate the semantics of these parameters. Generic QoS metrics based on time, cost, availability, and reliability have been described in [2].

C. User Preferences

Preferences are described generally as an aggregation of a user's basic desire formulas. A basic desire formula might be expressed in first-order logic (FOL) and describe properties of (partial) situations (i.e., composition). A detailed definition can be found in.

D. Constraints from Userss

User's constraints for some facts are described by axioms in OWL. As we see in the user preferences, all user constraints can be described by basic formulas and rules. OWL and the Semantic Web Rule Language (SWRL) are suitable for the formulas and rules, respectively. We can describe all the user constraints in the form of FOL.

E. Constraints from Existing Knowledge

There are constraints from general knowledge or that can be inferred from users' experiences. Some constraints can be related to a specific domain, others cannot. Such constraints are usually described in the same formula as constraints from users. The formula consists of inference rules that can extract new facts or constraints from existing knowledge. The constraints relate not only to individual processes, but also to multiple processes. Therefore, constraints will be relationships of more than one process. Sources of constraints are unlimited and change dynamically. LTL is a useful tool to describe constraints in complex general situations. Here, we extend the existing rules using LTL to enable a more flexible environment for constraint generation. LTL can check temporal logical violations of constraints according to the workflow of Web services invocation by time. Details of LTL definitions are explained next.

IV. FORMALISM FOR CONSTRAINT DESCRIPTIONS

Constraints including preferences are described in FOL. Here we give definitions for the FOL fundamentals and formulas for constraints, defined in the order term, formula, FOL language, and constraint formula.

Definition 1 (Basic Constraint Formula)
A basic constraint formula is a sentence drawn from the smallest set B where:

- $R \subset B$
  where R is a set of rules for constraints.
- If ϕ_1 and $\phi_2 \in R$, then so are $\neg\phi_1$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $(\exists x)\phi_1$, $(\forall x)\phi_1$, next(ϕ_1), always(ϕ_1), eventually(ϕ_1), release(ϕ_1, ϕ_2), and until(ϕ_1, ϕ_2)
- The next(ϕ_1), always(ϕ_1), eventually(ϕ_1), release(ϕ_1, ϕ_2), and until(ϕ_1, ϕ_2) are basic LTL constructs
- Additional LTL constructs:
  - These are EX (possibly-next), AY (previous), EU (possibly-until), AS (since), possibly-eventually, inevitably-next, inevitably-always, past, and inevitably-past.

Operators and Constraints in LTL

We generalize the rules in the LTL as follows:

Let the sequence of tasks in an abstract workflow be a vector T, ontology for the tasks as O_T, and N = |T| the cardinality of T. In detail, each element of T is an ontology class of the abstract task, which constructs the abstract workflow.

Next, we define an operator K for the ontology

classification operation.

When there is a class X that has an anonymous class to have the property objectProperty(ServiceDomain)) as the necessary and sufficient condition, we can find the T_D, that is, the tasks in the corresponding composition domain to apply K_X to O_T as

$$T_D = K_X \cdot O_T \quad (1)$$

Then, we can find all sequences of subclasses under T_D, and set them to Θ as

$$\Theta = \text{getAllSequenceinOrder}(T_D)$$

Now, we describe the LTL example of Section 4.1.5 as follows:

$$(\forall i)G(\text{ LessThan}(\Theta(i) + \text{TransitionTime}, \Theta(i+1)))$$

or

$$AG(\text{ LessThan}(\theta + \text{TransitionTime}, X(\theta))), \forall \theta \in \Theta$$

V. Transformation of Abstract Constraints

A. Abstract, Intermediate, and Concrete Constraints

As we discussed in the previous section, constraints can be described in FOL. If a constraint is described in natural language, it needs to be translated into FOL. This is done by using natural language processing to extract keywords corresponding to the terms and predicates of FOL sentences. Some words may have a complex meaning that combines term and context information. An abstract constraint consists of an abstract relation, terms, and context information. The abstract constraint transformer maps the abstract constraints to concrete constraints using the terms and context information.

Level 3: Abstract Constraints

Constraints at this level have humans' natural concepts. All terms are abstract and may not already be formalized into FOL. They can be in NL, or contain several complex meaning in a keyword. For example, "I prefer a non-smoking room", "I want the hotel fee to be less than 30,000", or LessThan(TotalCost, 60000). The first and the second one are like NL, but the third one takes the form of FOL. But the term "TotalCost" of the sentence has compound meaning (we say this is a compound term), that is "the entire fee of the complete travel sequence including hotel fee". Therefore, in the "LessThan(TotalCost, 60000)", the term "TotalCost" is a compound consisting of the message term "Cost" and the context of the term "Total".

Level 2: Intermediate Constraints

Intermediate constraints consist of a relation, terms, and context information. They are generated by extracting abstract relations, terms, and context information from abstract terms (which may include context information) in NL or compound terms at level 3. All the terms are terminal (not compound), and have not been bound to concrete terms yet. For example, "LessThan(TotalCost, 60000)" at the upper level (level 3) can be translated into "LessThan(AllService.Costs, 60000)", where "AllService" is a term for the context information, and "Costs" is a term for the message.

Level 1: Concrete Constraints

Concrete constraints have relations, terms as arguments of Web services, and invocation information for physical composition (in our implementation, CSP solving). All terms are bound to a real message of an operation of a Web service. The difference between a concrete constraint and an abstract constraint is that a term of the concrete constraint is bound to a message of operation of a service (or a process). And, the context information is used to find an operation or message of the corresponding process suitable for the message term.

Ontologies for PropertyDomain, ServiceDomain, and Context Information

The "PropertyDomain" ontology defines properties for characteristics of terms that appear in constraints, such as for cost and time. The "ServiceDomain" ontology defines all the terminologies and relations of a service domain, such as travel, hospital situation, etc. The "ContextInformation" ontology defines terms and operations for indicating a service that is related to the term that has an instance of the ontology. The instances of the ontologies are the terminal terms. A detailed explanation of the ontologies is in the following section.

Definition 2 (Term in Constraint)
A term (T) in a constraint consists of

- A constant. The term for constant is a real value that belongs to a class in the "PropertyDomain" ontology.
- A term for the message of a service
- A term for context information. The context information is an instance of the "ContextInformation" ontology.

Definition 3 (Terminal Terms and Non-Terminal Terms)

A terminal term (T_T) is an element of the set of instances of the "ServiceDomain" ontology or the "PropertyDomain" ontology, including constants. It also belongs to set of instances of ontologies "AbstractConstraint" and "ConcreteConstraint" in definition. It cannot be split any more. A non-terminal term (T_N) is produced by one or more terminal terms or non-terminal terms. The production rule for the non-terminal follows the language grammar.

Definition 4 (Abstract Constraints (C_A) - Level 3)

- The abstract constraints consist fundamentally of a basic constraint formula with a context information term. The form can be one of FOL, Context Free Grammar, Context Sensitive Grammar, or NL.
- Terms that are abstract constraints belong to the set (T_T

$\cup T_N$), the term is to be split to T_T by translation algorithm eventually. We define the abstract constraint in FOL as $C_A = <R_A, (T_T \mid T_N)^+>$, where R_A represents an abstract relation.

Definition 5 (Intermediate Constraints (C_I) - Level 2)

An intermediate constraint has the form of the basic constraint formula fundamentally.

- All the terms in the constraint belong to T_T only.
- The terms of context information also are terminal.
- We define the intermediate constraint as $C_I = <R_A, T_{TA}^+>$, where T_{TA} is T_T in an abstract (in detail, intermediate) constraint.

Definition 6 (Concrete Constraint - Level 1)

A concrete constraint has the form of the basic constraint formula. A term of the formula can be either a term for a message or a term for context operator.

- A term for a message (concrete message term) in the constraint is a message (M) in an operation (O) of a service (S).
- A term for a context operator is used for managing terms for a message. We define the concrete message term as $T_{CM} = <S,O,M>$, and the concrete context operator term as T_{CO}.
- A term of a concrete constraint is a tuple, $T_{CC} = <T_{CM} \mid T_{CO} \mid \text{Constant}>$ where $T_{CM} \in T_T$ and $T_{CO} \in T_T$.
- A concrete constraint is a tuple, $C_C = <R_C, [T_{CC}]^+>$.

Definition 7 (Context Information Term)

This term contributes to find out the process including operation and message to which the term of this constraint points. The term has information for the following:

- To point out a service to which the constraint with this term belongs.
- To manage the services using defined operations, such as calculation and choosing.

The information can have the following contexts:

- Temporal Contexts: Now(N), Always(G), Next(X), Next+n (X_n), Previous(P), Previous–n (P_n)
- Absolute Pointing Contexts: First, End, This
- Set Operation Context: The operation function is the operator K_X of Equation (1) in section 4.
- Managing Contexts: Sum, Average, Max, Min

Translation from abstract constraints to the intermediate constraints requires extraction of relations and terms from constraints described in a high level language. It may require natural language processing and an understanding of context semantics. As it is beyond the scope of this paper, it will not be discussed further.

After the translation, a transformation is needed to convert from intermediate constraints to concrete constraints. Because the terms and relation of an abstract constraint do not bind to terms for real Web services, the transformer must perform the binding.

VI. TRANSFORMATION OF ABSTRACT CONSTRAINTS

A. Ontologies to Support Transformation

In this section, transformation from an intermediate constraint (C_I) to a concrete constraint (C_C) is described. Sound transformation is carried out by a network of semantics and relations among classes in the ontologies for abstract/concrete constraint, abstract task, service candidate, and service/property domain. Our algorithm works on the ontologies to find concrete terms from abstract terms.

The ontology of abstract constraint and concrete constraint follows the definition 6-7. But they have more information about affiliation of terms to domains (service domain and property domain). The information gives an important clue for transformation to a concrete constraint. The service domain and property domain ontology describes classification information of the domain for terms or processes.

B. Transformation Algorithm

An intermediate constraint C_I consists of an abstract relation R_A and abstract (in definition 8, intermediate) terms T_{TA}

$C_I =< R_A, T_{TA} >$

A concrete constraint C_C consists of a concrete relation R_C and concrete terms T_{CC}

$C_C =< R_C, T_{CC} >$

We impose a constraint that each abstract constraint has a sequence of two terms, a message term and a context term. An intermediate constraint C_I is transformed to a concrete constraint to the following tuple in the case of a pair of terms.

$$C_C \leftarrow T_{CC1} \times T_{CC2} \times R_C$$
$$T_{CC} \leftarrow \boldsymbol{f}_{cTransform}(T_{TA})$$

The function $\boldsymbol{f}_{cTransform}(T_{TA})$ is described in the Algorithm 1. It works according to the abstract terms for transformation. When the context term is SetContext, it extracts context information from inference of the ontologies.

And, the P_t^m is a set of output concrete processes that belong to property domain class.

$P_t^m \leftarrow \{p : p \in t.\text{getOutputConcreteProcesses}() \square$

$\text{getPropertyDomainClass}(p) \subseteq \text{getPropertyDomainClass}(m)\}$

For example, an abstract task "TrainFromAtoB" has some service candidates. A service candidate of the candidates has 3 output concrete processes.

Output concrete process	Property domain
TrainService.getDepartureTime("1222M")	**TimeFrom** Property
TrainService.getArrivalTime("1222M")	**TimeTo** Property
TrainService.getCost("LocationA","LocationB")	**Cost** Property

If an abstract term has a message term "price" and an abstract task is "TrainFromAtoB", a concrete process of a concrete term has "TrainService.getCost("LocationA", "LocationB")" because a message term "price" belongs to CostProperty.

$ct \in T_{CC}$: a new concrete term of a concrete constraint
v : a new variable for a process
$tasks_n$: a vector of abstract tasks made by the logical composer
m : a message term of a process
c : a context term of a process
$m \leftarrow T_{TA}$.getMessageTerm()
$c \leftarrow T_{TA}$.getContextTerm()

//an abstract term is constant value
1 : ***case*** *(T_{TA}.getConstant() ≠ nil)*
2 : T_{CC}.add (ct.setConstant (T_{TA}.getConstant ()))
//a context term has relational index
3 : ***case*** *(c ∈ {AbsoluteTemporalContext})*
4 : $t \leftarrow tasks_{c.getIndex()}$
5 : T_{CC}.add (ct.setVariable(v.setConcreteProcesses (P_t^m)))
//a context term has relational index
6 : ***case*** *(c ∈ {RelationalTemporalContext})*
7 : ***for each*** *$t \in \{t: t \in tasks_n \wedge T_{TA}.getAbstractConstraint() \in t.getAbstractConstraints()\}$*
8 : $t \leftarrow tasks_{c.getIndex()+t.getIndex()}$
9 : TCC.add (ct.setVariable (v.setConcreteProcesses (P_t^m)))
10: ***end for***
// a context term belongs to some service domain classes
11: ***case*** *(c ∈ {SetContext})*
12: ***for each*** *$t \in \{t: t \in tasks_n \wedge getServiceDomainClass(t) \subseteq getServiceDomainClass(T_{TA}.getContextTerm()) \wedge getServiceDomainClass(t)$ has $getPropertyDomainClass(m)\}$*
13: T_{CC}.add (ct.setVariable(v.setConcreteProcesses(P_t^m)))
14: ***end for***
// some variables are operated statistically by CSP solver
15: ***case*** *(c ∈ {TermOperator})*
16: ***for each*** *$t \in \{t: t \in tasks_n\}$*
17: ct.addVariable (v.setConcreteProcess(P_t^m))
18: ***end for***
19: T_{CC}.add(ct)
20: T_{CC}.setTermOperator(c)

Algorithm 1. Transformation Algorithm - cTransform

VII. IMPLEMENTATION AND EVALUATION

A prototype system to show an implementation of the proposed architecture and algorithm was developed and illustrated in the trip planning domain. As in the complete architecture for automatic Web service composition shown in Figure 1, there are three main blocks: the LC, the PC, and the transformer of composition properties.

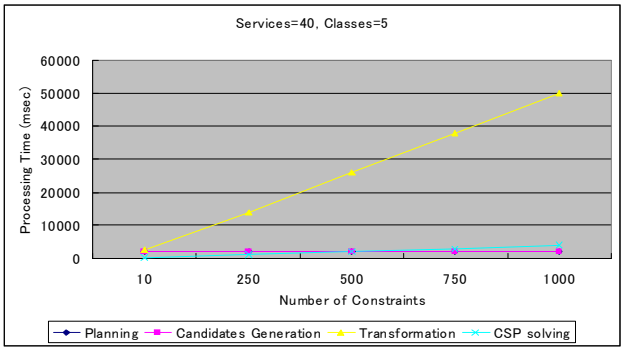
The aim of the analysis is twofold. First, we show the scalability of our architecture by computing the composition time for number of constraints and candidate services. Second, we assess the impact of the number of classes in the ontology for finding concrete services and terms on the transformation time.

Although the test set is not large, it provides explanatory data for the expected scalability and extensibility of the composition system. In the first set of experiments, we evaluate the time for planning (in our architecture, JSHOP for HTN planning was used), candidate service generation, transformation, and CSP solving by number of constraints and number of service candidates. In the travel scenario, we vary the number of constraints from 10 to 1000 with the number of services of 40. Figure 2(a) depicts the composition time by number of constraints. Figure 2(b) is the case of varying the number of services from 20 to 75 with 50 constraints. We separated the transformation time into two: the time for input and parsing, and the time for the transformation as a function of the number of classes in the ontology for finding concrete services and terms. Figure 2(a) shows that the transformation time increases linearly according to the number of constraints when the numbers of services and ontology classes are fixed, and the other processing times are constant and relatively low. Figure 2(b) shows the number of services affects to the time to process candidate generation and CSP solving. The time for CSP solving increases rapidly when the number of services is large.

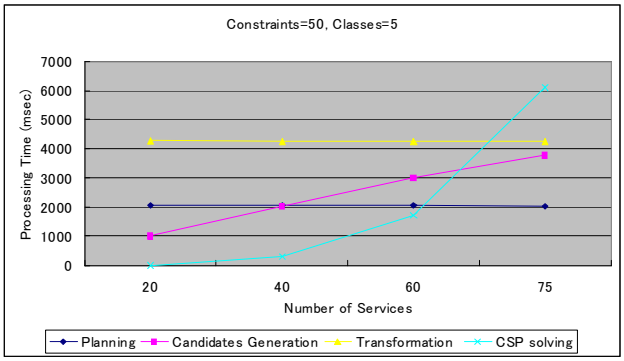
VIII. CONCLUSION AND FUTURE WORK

For automatic composition, there are three important parts: logical composition, physical composition, and composition property handling. This paper explores the handling of the composition properties. We have suggested an architecture for automatic service composition, and a model of service composition properties in the architecture. We have introduced the composition property stack, and a formalism for constraints. We have also modeled abstract/concrete constraints together with an ontology and an algorithm for transformation of the intermediate abstract constraints to concrete ones for automatic Web service composition. Implementation of the composition prototype using the algorithm and experimental analysis to access the performance of our system have also been explained.

Future work will include the following issues. First, translation from highly abstract constraints such as natural language to intermediate abstract constraints will be studied. Second, second subprocedure for LC-to-PC iterations can provide improved service composition functionality, because a new service may be created via several repetitions of an original composition result.



(a)



(b)

Figure 2. Experiments of composition. (a) Processing time at each part of the composer by constraints. (b) Processing time at each part of the composer by services

9. References

[1] A.B. Hassine, S. Matsubara, and T. Ishida, A Constraint-based Approach to Horizontal Web Service Composition, Proceedings of ISWC 2006, pages 130-143, Athens, USA, 2006.

[2] R. Aggarwal, K. Verma, J. Miller, and W. Milnor, Constraint Driven Web Service Composition in METEOR-S. In Proc. IEEE Int. Conf. on Services Computing, pages 23–30, 2004.

[3] V. Agarwal, and 6 others, A Service Creation Environment Based on End to End Composition of Web Services, Proceedings of WWW 2005 Conference, pages 128–137, Chiba, Japan, 2005.

[4] T-C. Au, U. Kuter, and D. Nau, Web Service composition with Volatile Information. Proc. ISWC'05, pages 52–66, 2005.

[5] U. Kuter, E. Sirin, B. Parsia, D. Nau, and J. Hendler, Information Gathering During Planning for Web Service Composition. In Proc. ISWC'04, pages 335-349, 2004.

[6] S. McIlraith, and T.C. Son, Adapting Golog for Composition of Semantic Web Services. KR-2002, France, 2002.

[7] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau, HTN Planning for Web Service Composition Using SHOP2. Journal of Web Semantic vol. 1, pages 377–396, 2004.

[8]E. Sirin, B. Parsia, and J. Hendler, Template-based Composition of Semantic Web Services, AAAI, 2005.

[9] I. Paik, D. Maruyama, and M. Huhns, A Framework for Intelligent Web Services: Combined HTN and CSP Approach, Proc. of IEEE ICWS, pages 959-962, Chicago, 2006.

[10] S. Sohrabi, N. Prokoshyna, and S.A. McIlraith, Web Service Composition via Generic Procedures and Customizing User Preferences, Proceedings of ISWC 2006, pages 597-611,2006.

[11] L. Zeng, B. Benatallah, A.H.H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang, QoS aware Middleware for Web Service Composition. IEEE Trans. Software Engineering, 30(5), 2004.

[12] G. Canfora, M.D. Penta, R. Esposito, and M.L. Villani, An Approach for QoS-aware Service Composition Based on Genetic Algorithms. In Proc. ACM GECCO'05, pp. 25–29, 2005.

[13] X. Fu, T. Bultan, and J. Su, Synchronizability of Conversations among Web Services, IEEE Trans. On Software Engineering, vol. 31, no. 12, pages 1042-1055, Dec. 2005.