Volume 51, Number 2, 2010

ACTA TECHNICA NAPOCENSIS
Electronics and Telecommunications

WORKFLOW BASED SERVICE COMPOSITION USING GENETIC ALGORITHMS

Florin-Claudiu POP   Marcel CREMENE   Mircea-Florin VAIDA
The Technical University of Cluj-Napoca, Faculty of Electronics, Telecommunications and Information Technology
Department of Communications, 26 – 28 Baritiu, 400027 Cluj-Napoca, Romania, Phone: (+40) 264 401810
Email: { Florin.Pop, Cremene, Mircea.Vaida }@com.utcluj.ro

Abstract: One common direction in service oriented computing is to use predefined workflows to compose new services that combine the functionality of existing services. But the problem with this approach is that services can't be composed for which workflows are not available. We propose a genetic algorithm that combines workflows in order to find the best solution that satisfies a set of objectives formulated using imperative natural language requests. The function that measures how close a solution is to what the user expects from the composed service is a semantic similarity function, called the conceptual distance.

Keywords: *web services, service composition, semantic similarity, genetic algorithms.*

I. INTRODUCTION

The World Wide Web is no longer a collection of content-based static HTML pages. Modern Web sites provide complex functionality to their users in the form of Web services (e.g. online document editing, online maps with navigation, online messaging, etc.), thus transforming the Internet into a rich application platform.

Today, Web services are getting more diverse and widespread as Internet-enabled devices become ubiquitous. A *Web service* is defined in [1] as a software component, language and platform independent that provides atomic functionality and is discovered and invoked using standard Internet protocols.

The functionalities provided by two or more services can be combined in order to create a new service that solves a more complex task. This process is called *service composition*. One composed service can use, for example, the *maps* service and the *messaging* service to send an email message with the location of a restaurant in the form of a dinner invitation.

Traditionally, service composition is a manual process, solved by a human expert, because it requires the understanding of service oriented computing concepts. This means that, the creation of a new service requires the human expert intervention, either before, by prediction, or after the user requirement was expressed.

A priori service composition is about predicting all possible user requirements. But such an approach is practically impossible. Even if the human expert will try to predict only the most probable user requirements, this task will be an extremely costly one. On the other hand, service composition (once the user requirements are known) by a human expert requires an important amount of time. Therefore, software applications to automate this process have been proposed.

This paper describes a service composition platform that uses predefined workflows to create new services and proposes a method of combining such workflows in cases when the user requirements cannot be satisfied by the predefined set of workflows. Workflows are combined using a technique inspired by genetic programming, based on the semantic similarity between the user requirements and the semantic annotations of services. We have chosen this approach because we assume that the user expresses his objectives using an imperative natural language sentence, as it is the most natural human form of interaction.

II. SERVICE COMPOSITION

Service composition systems usually resemble an architectural model with three components [11]: an input subsystem to process the user request, a composition subsystem to generate the composed service and an execution subsystem. The problem we approach in this paper is related to the composition subsystem.

Recent research efforts tackle the problem of service composition from two different perspectives [12]:

a) Workflow Based. In this case, the composed service is an executable workflow. A workflow may be, for instance, a standard process that includes some abstract activities, having a set of requirements. Workflows can be static, when they are created a priori by a human expert, or they can be generated dynamically at execution time. In the static approach, like the solutions proposed by [13] and [14], only the selection and binding of concrete services is done by the program. *The drawback of this approach is that they cannot create new services, for which templates are not available a priori.*

An example of dynamically generated templates is the solution called Polymorphic Process Model [7], which generates workflows using predefined subprocesses. Such solutions both create the process model and select concrete services automatically. The main disadvantage of this approach is that there is still a need for the expert user to write subprocesses.

b) *AI Planning Based.* Given a set of actions, their preconditions and positive and negative effects, a complete

Manuscript received April 23, 2008; revised June 15, 2010

description of the initial state and a user goal, AI planners work by finding a sequence of actions to achieve the goal. A composed service is requested by specifying the initial state and the user goal. In service composition, the number of available services is huge creating a shallow and broad search space, in contrast to the narrow and deep search space for which classical planning algorithms are tailored. Also, services might expose complex interfaces with complex message exchange patterns, but without full insight, while classical AI planners require complete knowledge of the world.

Therefore non-classical planners have been proposed for service composition. AI Planning based on situation calculus was proposed by papers [8] and [9]. Software agents are used to discover, execute and compose web services. The user request can be presented by the first-order language of the situation calculus. Web services are conceived as actions. Service composition in this case consists in creating complex actions to solve a user defined goal. Rule-based planning uses rules to define the possible web service's attributes used in service composition. For example, rules can be used to determine if two services are composable. In [10] rules consider the syntactic and semantic properties of web services and are used to determine the compatibility, the purpose and the quality of a composed service.

III. PROPOSED SOLUTION

1. *General architecture*

NLSC (Natural Language Service Composer) [15] was the first release of our natural language service platform. This paper proposes *NLSCgen* (NLSC based on genetic algorithms). The system uses a probabilistic natural language parser, called RASP, in order to transform the user input into a machine readable, *formal request* (Figure 1). The input can be textual or based on voice recognition. The process of obtaining the formal request from English imperative sentences is detailed in papers [15] and [16]. An imperative request can be modeled as a function which assigns a To-Do List (TDL) to the addressee:

$$Requiring\ TDL(A) \cup \{P\},$$

where (A) is the set of actions with parameters $\{P\}$.

First, the sentence from the user is tokenised (i.e. separate punctuation from adjacent words), parts of speech are identified and tagged using a tagset similar to CLAWS C7, and then the sentence is lemmatized using previously assigned tags. Grammatical relations between parts of speech (direct/indirect/prepositional objects, predicate complements or adverbials) are used to identify the actions (A) and their parameters $\{P\}$.

For example, for a user request like "*Print directions from Paris to London*" the verb *print* is the *predicate* and the noun *direction* is its direct object. The *direct object* grammatical relation represents the semantic description of an action the user assigns to the system. Therefore, such a user request produces the following formal request:

$$<A_1, \{P_1\}> = <\textit{print directions}, \{\textit{Paris}, \textit{London}\}>$$

The formal request represents the input for the composition subsystem, called *Service Composer*. The Service Composer is based on a service platform called WComp [6]. This platform is targeted mainly for intelligent environment applications. WComp was designed to support dynamic assembly of services for hardware devices. Web services and UPnP services in general can be used with this platform also.

The *AoA* (Aspects of Assembly) mechanism, which comes with WComp, allows the developer to create composition patterns and use them at runtime in order to modify the service architecture.

We motivate our preference for WComp platform mainly because of the flexibility provided by the AoA pattern support. Additional to other pattern-based approaches, AoA patterns can be superposed. Thus, a large number of valid combinations (services) may be created.

2. *Service Composer*

Service Composer (SC) is a workflow-based, pattern-oriented service composition system, being capable of handling complex interactions between components and providing flexibility in choosing different sets of components.

The AoA architecture consists of an extended model of Aspect Oriented Programming (AOP) for adaptation advices and of a weaving process with logical merging. An AoA pattern is structured as an aspect with a list of components involved in composition (called "*pointcut*") and an adaptation advice (a description of the architectural reconfigurations), which is specified using a domain specific language (DSL).

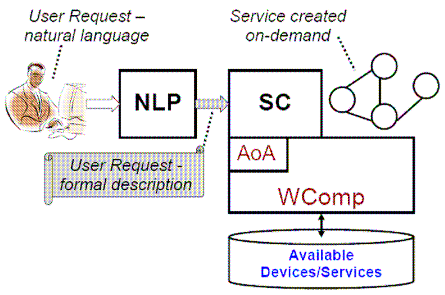
User Request – natural language
Service created on-demand
NLP
SC
User Request - formal description
AoA
WComp
Available Devices/Services

Figure 1. NLSCgen architecture

For example, the pattern that describes the process of a flight ticket reservation can be as follows:

```
Pointcut
in:=/CreditCardService.*/
out:=/FlightBookingService.*/

Advice
BookFlightUsingCreditCard(in, out):
in.^approved ->(out.bookFlight; CALL)
```

The first 3 lines (defining the pointcut as in aspect oriented programming) describe, using filters in the AWK language, the services used when applying this pattern: a credit card service (in) and a flight booking service (out). The filters of type /instanceName.*/ will find services whose names match instanceName.

Line 5 declares a composition schema that uses the previously defined services. Lines 6-7 specify the composition mechanics: call the *bookFlight* method when the *CreditCardService* fires the event *approved*. Since an AoA pattern defines in fact an event driven architecture, the flow of events at runtime can be modeled as a workflow. The workflow for the previous example pattern is modeled in Figure 2.

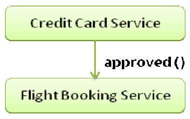

Figure 2. Flight booking service workflow

A pattern is automatically applied by the composition subsystem when satisfying a user request. The result of applying a pattern is projected in terms of pure elementary modifications: add required services, link ports. The resulted architecture behaves according to the pattern's eventflow. Therefore, we can look at a composed service from two perspectives: architectural and behavioral. Since we want to satisfy the user's functional requirements, we approach the problem from the behavioral perspective, focusing on the functionality of the composed service – the workflow.

For example, the previous example workflow will be selected by the *Service Composer* when the user input is "*Book flight from Los Angeles to New York*". In this case, the workflow is selected from the list of available workflows after a process of semantic matching between the user input and the services ontology.

But the main problem with this approach is that we can't create new services, for which workflows are not available a priori. For instance, if the user input is "*If book flight from Los Angeles to New York then rent a car*", the system will select the *flight booking* workflow, but the rent a car objective will not be satisfied.

The AoA architecture allows multiple workflows to be superposed. So if there is a *rent a car* workflow available and a *flight booking* workflow, they can be composed and the user request is fulfilled. Composition in this case is a simple logical merging. And superposition is not always the best solution. Considering that the *rent a car* workflow also includes an insurance buying service the user didn't ask for, the resulted composed service will satisfy more objectives than needed. Therefore, we propose a new method for workflow composition, based on genetic algorithms.

3. *An introduction to Genetic Algorithms*

Genetic algorithms are an evolutionary computing technique used to find exact or approximate solutions to optimization and search problems. The search space (solution domain) is modeled as a population of individuals, where each individual represents a potential solution. This population is the subject of an evolutionary process inspired by the principles of natural selection introduced by C. Darwin: selection, crossover, mutation. On each generation, the best solutions reproduce, while the weaker ones don't survive.

The characteristics of an individual are determined by its *genotype*. This information is encoded in a *chromosome*. The chromosome is composed of *genes*. Each gene represents a feature of an individual. The concrete value of the feature is called an *allele*. The genotype represents a potential solution to a problem, while the *phenotype* is its value.

A typical genetic algorithm executes in 5 steps:

Step 1. *Initialization*. The population is initialized with randomly picked genes.

Step 2. *Evaluation*. Partial solutions encoded in each chromosome are evaluated according to their relevance to the parameters of the accepted solution.

Step 3. Selection. Select the chromosomes that are closest to the final solution.

Step 4. *Evolution*. Create a new generation of chromosomes from the previously selected individuals, using genetic operations: crossover and/or mutation. The new population is composed of the individuals in the new generation and a few individuals (selected probabilistically) from the previous generation.

Step 5. *Termination*. If the solution is not good enough or the time limit was not reached, go to Step 2. Otherwise, stop the execution.

The function used to evaluate the quality of a potential solution is called the *fitness* function. The fitness can be measured in a variety of ways: a distance, an error, a time interval, etc. Evaluation means computing the fitness function for each individual. If the fitness function (*f*) is used to associate a probability (p_i) of selection with each individual chromosome, as in (1), the process is called a *roulette-wheel selection*.

$$p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \qquad (1)$$

Other common types of selection are: stochastic universal sampling, tournament selection or truncation selection.

Genetic operations used to develop the population are:

Reproduction. An individual is probabilistically selected from population, based on its fitness and copied unchanged in the new generation.

Crossover (recombination) allows combining the information from one or more parents in order to generate new offspring. One type of recombination is the *one-point crossover* (Figure 3). In this case, a random crossover point is selected. The first part of the first parents is hooked up with the second part of the second parent to make the first offspring. The second offspring is build from the first part of the second parent and the second part of the first parent.

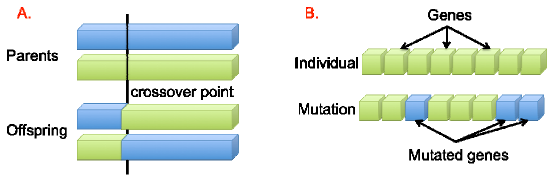

Figure 3. Genetic operations

Mutation means altering the value of a gene. This ensures, through diversity, avoiding the cases when a gene does not appear into the population because it wasn't selected in the initialization phase.

4. Services as genes

The problem we want to solve is finding the best workflow that satisfies a given user request. The search space consists of the initial set of workflows that were created a priori by an expert and all the possible combinations of these. This is a shallow and broad space and the time required by an exhaustive search increases exponentially with the linear growth of the space. Therefore, we developped a heuristic search method to quickly find a good solution to this problem.

We model a workflow as a *chromosome* where each *gene* represents a *service* instance. Unlike classical genetic algorithms, the chromosome does not have a fixed number of genes. This approach is inspired by genetic programming, which evolves computer programs. An example of how a workflow is encoded into a chromosome is depicted in Figure 4.

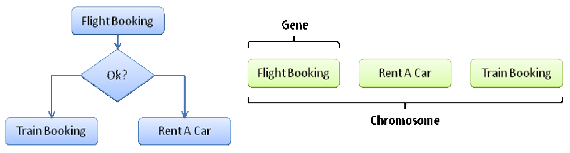

Figure 4. Representing a workflow as a chromosome

We use a variant of the one-point crossover technique for the recombination of individuals, called "*cut and splice*". Pairs of parent chromosomes are probabilistically selected from population. The difference from classical one-point crossover is that the crossover point is not identical for both parents. A random crossover point is chosen for each parent. This results in a change in length of the children strings. An example of the "*cut and splice*" crossover for two workflows is shown in Figure 5.

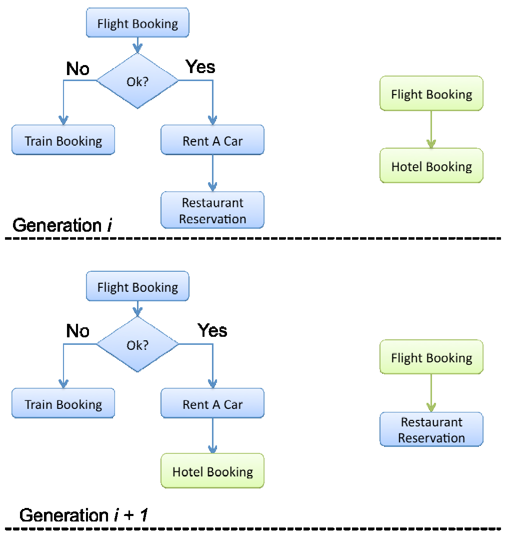

Figure 5. Cut and splice crossover

A mutation means altering the value of a gene. Since a gene stores a service instance, during the mutation, we replace the service instance in a gene with a different service, from the list of available services. For instance, the *flight booking* gene is mutated into the *train booking* gene.

The *fitness* function measures how well the composed workflow satisfies the user input. In order to do this, our fitness function is defined as a measure of the semantic similarity between the workflow and the user input. We call this semantic similarity, the *conceptual distance*.

5. The conceptual distance

Measures of semantic similarity or relatedness are found in linguistic processing literature. According to the study published in [18], the *Jiang and Conrath's* measure proved to be best for practical usability. All compared distances are based on the information content of the lexical terms (the probability of encountering an instance). While this information may be valuable for other applications, it is of less importance for service composition and it adds to the complexity of the implementation, therefore making it slower. Our approach, the conceptual distance, is faster and less complex than Jiang and Conrath's measure, while the last is more accurate.

We used for this purpose a specialized dictionary called WordNet. Word-Net contains nouns, verbs, adjectives and adverbs in sets of synonyms, called synsets. Different senses of a word are in different synsets. Both nouns and verbs are organized into hierarchies, defined by *IS A* relationships. For example, the hierarchy for mobile phone is:

→ cellular telephone, cellular phone, cell phone, cell, mobile phone
→ radiotelephone, radiophone, wireless telephone
→ telephone, phone, telephone set
→ electronic equipment

The words at the same level in hierarchy are synonyms of each other. The algorithm that evaluates the conceptual distance uses the WordNet lexicon to create concept hierarchies. A concept hierarchy is generated in four steps:

Step 1. Find the synset that contains the concept for which the hierarchy is generated.
Step 2. Each word in the synset becomes a root for a tree in the concept hierarchy.
Step 3. For each tree root, find the synsets that are in a relationship with the roots synset. Each word in the related synset becomes a leaf for the tree, on the next level in hierarchy, branching from the root. For each word on the current level in hierarchy, find the synsets related to the words synset and add the words in the found synsets as leafs for the tree on the next level.
Step 4. Repeat Step 3 until the hierarchy is big enough so that the degree of generalization for the notion for which the hierarchy is built, corresponds to an accepted accuracy that produces best results. The bigger the hierarchy the longer it takes to generate it, but the smaller the hierarchy the more confusion can occur among concepts.

In order to evaluate the conceptual distance for two notions, a concept hierarchy is built for each notion. Then, the conceptual distance is calculated as follows:

$$D = \frac{\min(L_{C1}, L_{C2})}{L_{\max}}, \tag{2}$$

where L_{C1} and L_{C2} are numerically equal to the depth of a common node in the 2 hierarchies of concepts, if such a common node exists; $L_{\max}$ is the maximum depth in the hierarchy.

Given the conceptual distance in (2), we can calculate the conceptual similarity with the formula:

$$S = 1 - D \tag{3}$$

Examples:

```
D (Mobile Phone, Cell) = 0
D (Radiotelephone, Radiophone) = 0
D (Mobile Phone, Radiophone) = 0.25
D (Mobile Phone, Telephone) = 0.5
```

5. The fitness function

The *fitness* function calculates the *conceptual distance* between the semantic description of the workflow and the user input. In other words, it evaluates the semantic similarity between the two. As we explained earlier, the formal representation of the user input is a list of actions *(A)* with specific parameters {P}. Each action is described by a group of words (e.g. *print directions*). Also, Web services are semantically annotated using the OWL-S language. Given that a chromosome encodes a workflow and a workflow contains a list of services, then we need to evaluate the similarity between the actions named by the user and the services annotations.

For example, if the user input is "*If book flight from Los Angeles to New York then rent a car*", the set of actions is *A*= {*book flight, rent car*} and if the chromosome is as the one in the subsection 4, the set of services annotations can be *S*={*flight booking, rent a car, train booking*}. The fitness function calculates the similarity between the content of the two sets. It does this using the following formula:

$$F = 1 - \frac{\sum_{i=1}^{N_{\min}} \min_{m,n}(D_i(A_m, S_n))}{N_{\max}} \tag{4}$$

N_{max} / N_{min} is the maximum / minimum of the sizes of sets *A* and *S*. In case *A* and *S* have the same size, $N_{max} = N_{min} = N$. $D_i(A_m, S_n)$ is the conceptual distance between the element *m* in set *A* and element *n* in set *S*. According to (4), the greater the fitness *F* the better the potential solution.

IV. RESULTS

We tested our approach with different user inputs and potential services in order to measure the performance of the genetic algorithm. Results are summarized in table 1. First column represents the total number of available services, while the second column contains the number of services required to satisfy the user. As the order in which the services are invoked is not insignificant, the number of potential solutions is equal to the number of *k*-permutations of available services, where *k* is equal to the number of desired services.

The *population size* column shows the initial number of individuals and the *GA iterations* column represents the number of evolutions before the solution was obtained. As on each test, the evolution of the solutions is unpredictable, we recorded the smallest number of iterations out of 5 test runs. The *GA gain* column contains the computational gain of the genetic algorithm over the exhaustive search and is calculated as:

$$GA\_gain = \frac{Potential\_solutions}{Pop\_size \cdot GA\_iterations} \tag{5}$$

For example, in the case of 48 available services and 4 desired services, if evaluating a workflow takes 2 ms, then an exhaustive search will take 2.6 hours to complete, while the genetic approach will find the solution in 768 ms.

Avail. services	Desired services	Potential solutions	Pop. size	GA iterations	GA gain
5	2	20	4	1	5
9	2	72	4	1	18
15	2	210	4	2	26.25
48	2	2256	16	9	15.66
48	3	103776	16	28	231.64
48	3	103776	32	8	405.37
48	4	4669920	32	12	12161.2
48	4	4669920	64	9	8107.5

Table 1. Experimental results

V. RELATED WORK

1. Genetic Synthesis of Software Architecture

O. Raiha, in his Ph.D. thesis called *Genetic Synthesis of Software Architecture* [3] proposes a method for software architecture synthesis using genetic algorithms. His solution is based on the hypothesis that architectural patterns in programming theory can be interpreted using genetic operations: a mutation means applying or removing an architectural pattern, crossover means combining two architectures. The architectural patterns used by the author are: decomposing a component, using an interface, strategy design pattern, façade design pattern, and message dispatcher architectural style.

Similar to service composition, automated software architecture synthesis starts with a formal representation of the functional and non-functional requirements. Raiha adopted an approach where functional requirements are represented as a graph of functional responsibilities. Nodes in this graph represent tasks (responsibilities) and arcs are used to model dependencies between tasks.

Raiha applied the supergene concept for the genetic encoding of responsibilities. A supergene is a gene that can store more fields with data. Each supergene encodes a responsibility using the following functional information: name, type, parameter size, execution time, and architectural information: classes it belongs to, the interface it implements, the dispatcher it uses, the responsibilities that call it through the dispatcher, and the design pattern it is a part of.

The genetic algorithm starts with an initial population composed of individuals with random responsibilities. Four special cases are inserted: all responsibilities being in the same class, all responsibilities being in different classes, all responsibilities having their own interface, and all responsibilities being as much grouped to same interfaces as the class division allows.

All mutations consist in applying or removing an architectural pattern. Mutations are selected using the roulette wheel method, where each mutation is given a "slice" in proportion to its probability. The crossover is implemented as a one-point crossover and is considered a type of mutation, thus it is also included in the "roulette wheel". The fitness function is based on software metrics from the metrics suite introduced by Chidamber and Kemerer [5].

2. Dynamic Architectural Selection: A Genetic Based Approach

Kim and Park [4] use genetic algorithms to select the components of an architecture based on user preferences and the parameters of the system on which the architecture is deployed. For example, on mobile devices the parameters that have an effect on the functionality of an application can be the ambient noise, signal strength, battery level or the display brightness. Such parameters are called state variables.

To adapt to changes of state variables and user (qualitative) requirements, an application could pack several alternatives: *NormalDisplay / High-ContrastDisplay, RichGUI / NormalGUI / SimpleGUI, SimpleMessaging / Compressed Messaging*. At each particular moment in time, only a single alternative can be used (*NormalDisplay* and *High-ContrastDisplay* cannot be active at the same time). Each alternative has its own relation with qualitative requirements and state variables.

The set of alternatives is represented as an interdependency graph. Selecting an alternative is a combinatorial optimization problem. In order to solve this problem using genetic algorithms, the authors encode an architectural instance using a chromosome. This means that each gene contains an alternative to a component of the architecture. Single-point crossover is used and mutation means replacing a component with a random alternative. The fitness function measures the user satisfaction, based on a metric where each quality requirement has a certain weight factor.

VI. CONCLUSION

We've shown throughout this paper how the problem of service composition can be modeled as a search problem. While the number of services increases lineary, the size of the search space increases exponentialy making it almost impossible to find a good solution in a reasonable amount of time even for a small number of services.

Our approach uses a heuristic search method, based on genetic algorithms. Because it is intended to work with a natural language interface, the function used to test the quality of a potential solution is a semantic similarity function, called the conceptual distance.

Experimental results show that, if properly configured, the genetic algorithm can quickly find the best solution, even when the search space is very broad. The solution is a workflow that can be turned into an executable composed service.

ACKNOWLEDGEMENT

This research was supported by PN2 IDEI projects 1083/2007 and 1062/2007 from the National Authority for Science CNCSIS-UEFISCSU.

REFERENCES

[1] D. Booth, H. Hass, F. Mccabe, E. Newcomer, M. Champion, C. Ferris and D. Orchard "Web services architecture, W3C Working Group Note 11 February 2004", *W3C Technical Reports and Publications*, http://www.w3.org/TR/ws-arch/, 2005.

[2] D. Dumitrescu, *Algoritmi genetici si strategii evolutive*, Editura Albastră, Cluj-Napoca, 2000

[3] O. Raiha, *Genetic Synthesis of Software Architecture*, University of Tampere, Department of Computer Sciences, Lic. Phil. Thesis, September 2008.

[4] D. Kim, S. Park, "Dynamic Architectural Selection: A Genetic Algorithm Based Approach", *Proceedings of the 2009 1st International Symposium on Search Based Software Engineering,* pp.59-68, 2009.

[5] S.R. Chidamber and C.F. Kemerer, "A metrics suite for object oriented design". *IEEE Transactions on Software Engi- neering* 20, 6, 476-492, 1994.

[6] D. Cheung-Foo-Wo, J.-Y. Tigli, S. Lavirotte, M. Riveill, "WComp: a multi-design approach for prototyping applications using heterogeneous resources", *In 17th IEEE Intern. Workshop on Rapid Syst. Prototyping*, pp 119-125, Creta, 2006.

[7] H. Schuster, D. Georgakopoulos, A. Cichocki, D. Baker, "Modeling and composing service-based nd reference process-based multi-enterprise processes". *Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, London, Springer-Verlag 247–263, 2000.

[8] S. Mcilraith, T.C. Son, "Adapting golog for composition of semantic web services". 482–493, 2002.

[9] S. Narayanan, S.A. McIlraith, "Simulation, verification and automated composition of web services". *In WWW '02: Proceedings of the 11th international conference on World Wide Web*, New York, NY, USA, ACM 77–88, 2002.

[10] B. Medjahed, A. Bouguettaya, A.K. Elmagarmid, "Composing web services on the semantic web". *The VLDB Journal* 12(4) 333–351, 2003.

[11] M. Eid, A. Alamri, A. E. Saddik, "A reference model for dynamic web service composition systems". *Int. J. Web Grid Serv.* 4(2) 149–168, 2008.

[12] J. Rao, X. Su, "A survey of automated web service composition methods". *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition*, SWSWPC 2004. 43–54, 2004.

[13] E. Sirin, B. Parsia, J. Hendler, "Template-based composition of semantic web services". *Proceedings of the AAAI Fall Symposium on Agents and the Semantic Web, 2005.*

[14] A. J. Molina, H.-M. Koo, "A template-based mechanism for dynamic service composition based on context prediction in ubicomp applications". *Proceedings of the International Workshop on Intelligent Web Based Tools*, 2007.

[15] M. Cremene, J.-Y. Tigli, S. Lavirotte, F.-C. Pop, M. Riveill, G. Rey, "Service Composition based on Natural Language Requests", in *Proceedings of IEEE SCC 2009*, International Conference on Services Computing, Bangalore, India, 2009.

[16] F.-C. Pop, M. Cremene, M. Riveill, M. Vaida, "On-demand dynamic service composition based on natural language requests", *WONS 2009 The Sixth International Conference on Wireless On-demand Network Systems and Service*, USA, 2009.