2009 Ninth International Conference on Intelligent Systems Design and Applications

A Genetic Programming-based Algorithm for Composing Web Services

Manuel Mucientes, Manuel Lama, Miguel I. Couto
University of Santiago de Compostela, Spain
manuel.mucientes@usc.es, manuel.lama@usc.es, miguelixen.couto@rai.usc.es

***Abstract*—Web Services are interfaces that describe a collection of operations that are network-accessible through standardized web protocols. When a required operation is not found, several services can be compounded to get a composite service that performs the desired task. To find this composite service, a search process over a huge search space must be performed. The algorithm that composes the services must select the adequate atomic processes and, also, must choose the correct way to combine them using the different available control structures. In this paper a genetic programming algorithm for web services composition is presented. The algorithm has a context-free grammar to generate the valid structures of the composite services. Moreover, it includes a method to update the attributes of each node. A full experimental validation with a repository of 1,000 web services has been done, showing a great performance as the algorithm finds a valid solution in all the tests.**

***Keywords*-Web service, composition, genetic programming.**

I. Introduction

Web Services are interfaces that describe a collection of operations that are network-accessible through standardized web protocols, and whose features are described using a standard XML-based language [1], [2]. This includes functional features that indicate the input/output needed to invoke the execution of a web service; non-functional features such as cost, robustness, reliability, etc.; interaction features or choreography that describe how a client dialogs with the service in order to consume its functionality; and structural features or orchestration that model how the internal components of the service are combined to execute it.

In this way, as the characteristics are available through the interfaces, the web services can be discovered and invoked automatically by extern programs (clients). When programs do not find a service with the required functionality (inputs and outputs), it is possible to compose a new service automatically. This composed service combines the functionalities of other services to get the desired outputs. One of the main advantages of web services is that they make their characteristics available to other programs or agents. This combination can consist in a set of services that are executed in a sequence or in a set of workflow-like structures that control the execution of the services (specified through web services composition languages as OWL-S [3] or BPEL4WS [4]). These two problems are very different from the point of view of computational complexity: in the second case the number of candidate solutions can be really high. This makes classical search methods not applicable.

In the last years several papers have dealt with the composition of OWL-S services. Some approaches consider the composition problem as a planning problem of several actions (services) that operate on an initial state (inputs and preconditions) and generate an output state (postconditions) [5], [6], [7], [8], [9]. In these works, the planning techniques are blended with semantic reasoning to combine the outputs of some services with the inputs of others. The main problem is that in these approaches the result of the composition is a sequence of services and, therefore, they do not take into account other control constructions that are part of the OWL-S model. In this way, this particular problem has a computational complexity much lower than those compositions that follow languages like OWL-S or BPEL4WS.

Other papers solve the composition of services with optimization techniques like linear logic [10] or genetic programming [10], [11]. These works have been validated with a low number of services and, consequently, the performance of the proposed algorithms cannot be really tested. The most similar approach to our proposal describes an algorithm for services composition in BPEL4WS [11]. The main difference with our proposal is that (i) it does not show a formal description of the grammar to compound services, and (ii) attributes updating after crossover and mutation is not explicitly managed. Therefore, it is difficult to evaluate the degree of fulfillment of all the interactions among services to get a valid solution.

In this paper we present a genetic programming algorithm that solves the problem of composition of OWL-S services. The algorithm uses a context-free grammar to limit the valid structures and takes into account the attributes updating. A full validation has been done in OWLS-TC [12], a repository with more than 1,000 services, showing a great performance, as in all the cases a correct composition was found.

II. Genetic programming for web services composition

The first step in the design of an algorithm for web services composition requires the definition of the type of composite services that are going to be build. A compact definition of the valid structures of a tree (chromosome) for a web services composition can be described by a context-free grammar.

978-0-7695-3872-3/09 $26.00 © 2009 IEEE
DOI 10.1109/ISDA.2009.155

IEEE computer society

- V = {initialProcess, process, compositeProcess}
- Σ = {anyOrder, atomicProcess, choice, sequence, split, splitJoin}
- S = initialProcess
- Rules:
  - initialProcess $\longrightarrow$ sequence process
  - process $\longrightarrow$ compositeProcess process | atomicProcess process | compositeProcess | atomicProcess
  - compositeProcess $\longrightarrow$ anyOrder process | choice process | sequence process | split process | splitJoin process

Figure 1. Context-free grammar for web services composition.

A. Context-free grammar

A context-free grammar is a quadruple (V, Σ, P, S), where V is a finite set of variables, Σ is a finite set of terminal symbols, P is a finite set of rules or productions, and S is an element of V called the start variable. The grammar that defines the valid structures for web services composition is described in Fig. 1. The first item enumerates the variables, then the terminal symbols, in third place the start variable is defined, and finally the rules for each variable are enumerated. When a variable has more than one rule, rules are separated by symbol |. Variable *initialProcess* is the start variable of the grammar and generates a sequence of processes.

Variable *process* defines either composite processes or atomic processes. Two of the four rules of this variable are recursive and, therefore, a process can be composed of any number of atomic and composite processes. Finally, variable *compositeProcess* represents the combination of a control structure and a process (of any type), i.e. a composite process.

All the nodes of type variable, together with terminal symbol *atomicProcess* constitute the service nodes. They are characterized by the following attributes:

- Control structure: the node of type control structure ({anyOrder, choice, sequence, split, splitJoin}) the service node depends on.
- Available inputs: are those inputs that are available for a service. A subset of them are selected as inputs to the service.
- Necessary inputs: are the inputs that the node needs for running all the atomic processes in the subtree for which the root is the node.
- Obligatory inputs: in some situations, the outputs of several services have to be used as inputs to the current service. This means that at least one of that outputs has to be selected as input to the current service. An example of this situation is the sequence of two services S_a and S_b. Let $O_a = \{o_1^a, \ldots, o_{n_a}^a\}$ be the set of outputs generated by service S_a, and $I_b^n = \{i_1^b, \ldots, i_{n_b}^b\}$ be the set of necessary inputs of S_b. Then, $O_a \cap I_b^n \neq \emptyset$. If this condition is not fulfilled, the composition of services S_a and S_b is not a sequence, and the structure is not valid. Therefore, the inputs of the service must contain a subset of the obligatory inputs. Following the example, the obligatory inputs of service S_b are the outputs of service S_a, i.e., $I_b^o = O_a$.
- Outputs: generated by the service. They can be directly generated by the service (if it is an atomic process) or by the subtree with the service as root node (composite process).

The size of the search space can be calculated taking into account both the grammar and the services repository. In this paper, the repository that has been used for the validation of our proposal is the OWLS-TC collection (a repository of 1,000 services). Thus, the size of the search space is over 10^{60}. Other approaches to services composition only consider sequences of services and, therefore, the size of the search space is much lower (10^{30} for sequences of up to 10 services). Consequently, our proposal solves a much more difficult problem.

B. Attributes updating

The initialization of a tree (web services composition), or a modification of it due to crossover or mutation, requires the updating of all the attributes of each node. The initial step of the algorithm resets all the attributes of all the nodes in the tree, and then initializes the necessary inputs of the root node (*initialProcess*) to the set of inputs of the web services composition to be solved. Then, the tree is traversed in preorder, updating the attributes of each node. To traverse a tree in preorder, the following operations must be performed recursively at each node, starting with the root node: first, visit the root. Then, traverse the subtrees that have as root node the children of the root. Children are traversed in order, starting with the leftmost node and continuing to the right. Updating the attributes of each node is done in a different way depending on the type of attribute:

- Control structure (*cs*): this attribute is propagated in a top-down way. This means that a node inherits the attribute value from its parent. There is an exception to this rule. The node will set its control structure to its leftmost brother when that brother is a control structure.
- Available inputs (I^a): they are propagated in a top-down way. When the control structure of the node is *sequence*, all the outputs of the brothers to the left of the node will also be added as available inputs.
- Necessary inputs (I^n): the propagation is done in a bottom-up way. This means that, if and only if the node is a leaf node, all its ancestor nodes will add as necessary inputs the necessary inputs of the node.
- Obligatory inputs (I^o): they are propagated in a top-down way. There is an exception to this rule: when the control structure of the node is *sequence* and the brother node immediately to the left is a service node. In this case, the following algorithm is applied to get the obligatory inputs:
  1) Traverse in preorder the subtree that has as root node the brother node just to the left of the current node (the one for which the obligatory inputs are being calculated).

2) Get the last node traversed in that process. It will be the rightmost node of the subtree.
3) If both the last and current nodes depend on the same control structure (they have a reference to the same node of type *controlStructure*), then the obligatory inputs of the current node are the outputs of the last node.
4) Else, the obligatory inputs of the current node are the outputs of the brother node immediately to the left.

- Outputs (O): the attribute is propagated in the same way as the necessary inputs.

Fig. 2 shows a services composition. Terminal symbols (leafs of the tree) are represented by rectangles or squares, and variables are shown as flatted circles. Each node includes the values of the different attributes. In this example the initial available inputs are i_a and i_b, while the outputs that are generated by the atomic processes are $o_{2.1}$, $o_{3.1}$, $o_{6.1}$, and $o_{7.1}$.

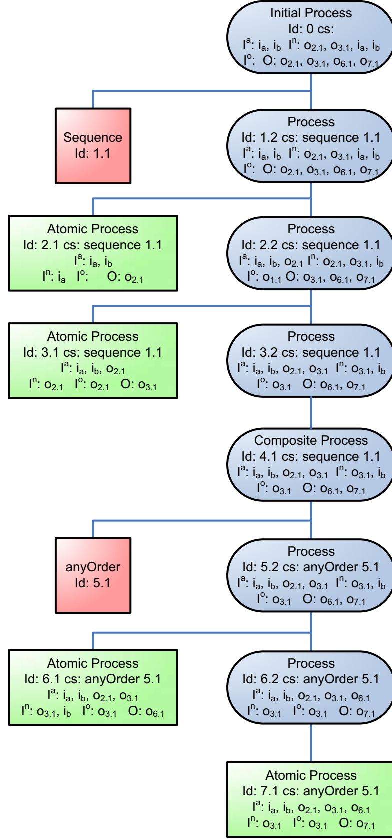

Figure 2. A chromosome representing the composition of several atomic processes.

C. Genetic programming-based algorithm

Fig. 3 describes the genetic programming algorithm that has been used for web services composition. The first three steps of the algorithm correspond to an initialization. t represents the number of iterations, while *timesRun* will be used to detect situations in which the search gets stuck. The iterative part of the algorithm starts at step four. This part will be repeated until the maximum number of iterations is reached or the best possible solution is found. The main stages of the iterative part are the selection of the individuals, the crossover and mutation to generate new individuals, their evaluation, the replacement of the population, the local search, and the checking of stuck situations in the search process. All of them are described in detail in the next sections.

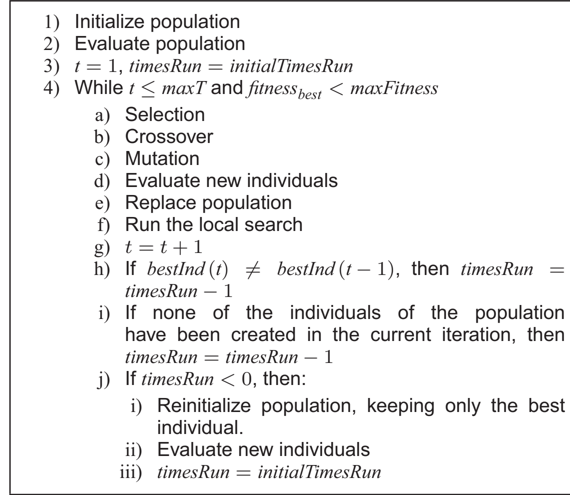
1) Initialize population
2) Evaluate population
3) $t = 1$, $timesRun = initialTimesRun$
4) While $t \leq maxT$ and $fitness_{best} < maxFitness$
   a) Selection
   b) Crossover
   c) Mutation
   d) Evaluate new individuals
   e) Replace population
   f) Run the local search
   g) $t = t + 1$
   h) If $bestInd(t) \neq bestInd(t-1)$, then $timesRun = timesRun - 1$
   i) If none of the individuals of the population have been created in the current iteration, then $timesRun = timesRun - 1$
   j) If $timesRun < 0$, then:
      i) Reinitialize population, keeping only the best individual.
      ii) Evaluate new individuals
      iii) $timesRun = initialTimesRun$

Figure 3. Genetic programming algorithm for web services composition.

1) Initialization: The first step of the algorithm is the generation of the initial population. A new individual is generated applying randomly the rules of the grammar. If the depth of the tree reaches the maximum predefined value, then all the nodes of type service at that depth are transformed to *atomicProcess* nodes. Once the structure of the tree has been defined, the attributes of the nodes must be initialized using the algorithm defined in Sec. II-B.

This attributes updating algorithm is run with one special characteristic. When an *atomicProcess* node is reached during the traversal of the tree, as no specific service has been assigned to it, one has to be selected from the repository. The selection is done randomly from the set of services that fulfill: $I_j^a \supseteq I_k$ and $I_j^o \cap I_k \neq \emptyset$. Thus, a service k can be selected if its inputs are a subset of the available inputs of the *atomicProcess* node j (I_j^a) and if at least one of the inputs of k belongs to the set of obligatory inputs of j (I_j^o).

2) Evaluation: The calculation of the fitness of each individual of the population is done analyzing four criteria: generated outputs, used inputs, depth of the tree and number of nodes of type *atomicProcess*:

$$fitness = \omega_1 \cdot \left(\frac{O_{root}}{O_{obj}} + \frac{I^n_{root}}{I_{obj}} \right) + \omega_2 \cdot \frac{1}{depth} + \omega_3 \cdot \frac{1}{\#atomicProcess} \quad (1)$$

where O_{root} are the outputs of the root node of the tree (this node is the result of the composition of the services), O_{obj}

are the outputs that are required to solve the composition, I^n_{root} are the necessary inputs of the root, I_{obj} are the inputs provided to solve the composition, $\#atomicProcess$ is the number of atomic processes in the tree, and ω_i are values that weight the importance of each criterion.

3) Selection: The selection mechanism that has been used is the binary tournament selection. In a k-tournament selection, k individuals are randomly picked from the population with replacement, and the best of them is selected. In this case, $k = 2$ (binary tournament selection).

4) Crossover: The crossover operator consists in the replacement of a subtree of the individual by a subtree of the other individual. The process is as follows:

- Select randomly a node of type service in the first individual.
- Generate the set of candidate nodes in the second individual. These nodes must have the following characteristics:
  - They must be of type service.
  - $(I^n_2 - O_2) \cap I^o_1 \neq \emptyset$. $I^n_2 - O_2$ represents all the inputs that are used by the subtree of the second individual and that have not been generated inside that subtree. This set of inputs must contain at least one of the obligatory inputs of the subtree that is going to be replaced in the first individual.
  - $I^n_2 - O_2 \subseteq I^a_1$. Also, the set of inputs used in the subtree of the second individual must be a subset of the available inputs for the subtree of the first individual.
- Select randomly a node of the candidate nodes set, and replace the subtree of the first individual with the selected subtree of the second individual.
- Execute the attributes updating algorithm. During the execution of the algorithm, if a leaf node of type *atomicProcess* is reached, two conditions must be checked: $I^a_j \supseteq I_k$ and $I^o_j \cap I_k \neq \emptyset$, i.e., the inputs of the process (I_k) must be a subset of the available inputs of the node and, also, they must contain at least one of the obligatory inputs of the node. If the conditions are not fulfilled, a new atomic process must be selected using the same procedure as in the initialization stage (Sec. II-C1).

5) Mutation: The mutation operator modifies a subtree of the individual. First, a node must be randomly selected. If the node is of type variable, then the subtree that has as root the selected node is eliminated. The new subtree is generated applying the rules of the grammar randomly for that variable in the same way as in the initialization stage (Sec. II-C1). On the other hand, if the node is of type terminal, there are two cases:

- If the node is a control structure, a new one is randomly selected.
- If the node is an atomic process, then a new process is randomly selected from the repository using the same conditions defined in the initialization stage (Sec. II-C1)

In all the cases, the attributes updating algorithm must be run. Also, the validity of the atomic processes must be checked and, if necessary, a new selection of the atomic processes is done.

6) Replacement: The selection mechanism is based on the CHC model [13]. It is a population-based selection approach, i.e., parents and their corresponding offspring are combined (generating a population with a size $2N$, being N the size of the initial population), and the best N individuals are selected for the next population.

7) Local search: The objective of the local search is to improve some of the individuals of the population implementing a search process with a low degree of exploration and a high degree of exploitation, i.e., a very exhaustive search in the neighborhood of the individual. In this paper the local search has been implemented with the steepest ascent hill climbing algorithm.

8) Reinitialization: The last steps of each iteration (Fig. 3) update the value of *timesRun*, decreasing it when the best individual has not improved and also when no individuals of the current iteration have survived the replacement process. If *timesRun* takes a value below 0, the population is reinitialized in the same way as in the initialization stage, but keeping the best individual.

III. RESULTS

The validation of the genetic programming algorithm for web services composition has been done with a set experiments with different degrees of complexity. A repository with 1,000 services has been used for the tests. The service compositions that have been implemented are shown in Fig. 4. For each example, a short description of the task that the composite service solves is given. Also, the available inputs (those provided by the user) and the desired outputs are enumerated. Then, the solution to the requested service is indicated: it is a combination of control structures and atomic processes. In most of the cases there are a few possible best solutions, but only one has been indicated in Fig. 4. After that, each of the atomic processes is described and, finally, the maximum fitness is indicated. This maximum fitness is, in most of the cases, under 1, as the depth of the tree and the number of atomic processes in the composite process are greater than one (Eq. 1).

Table I shows the results for all the test examples described in Fig. 4. Each column in the table represents the results of the evolutionary algorithm for a test example. As evolutionary algorithms are nondeterministic, results of one run over an example are not meaningful. Thus, for each of them 10 runs were executed. The rows represent the time to obtain the best solution found by the algorithm, the quotient between the necessary inputs of the root and the inputs provided to solve the composition, the quotient between the outputs of the root node of the tree and the outputs required to solve the composition, the fitness value, the depth and the number of atomic processes of the tree and, finally, the number of the different control structures in the tree (*sequence, split, splitJoin, anyOrder, choice*). For each of these columns, two values are represented: $\overline{\chi}$ is the arithmetic mean over

1) Obtain the time interval and the diagnostic process for a hospital:
   - Inputs: _HOSPITAL
   - Outputs: _TIMEINTERVAL, _DIAGNOSTICPROCESS
   - Solution: sequence(HOSPITAL_DIAGNOSTICPROCESSTIMEINTERVAL_SERVICE)
   - List of atomic processes:
     – HOSPITAL_DIAGNOSTICPROCESSTIMEINTERVAL_SERVICE:
       * Inputs: _HOSPITAL
       * Outputs: _TIMEINTERVAL, _DIAGNOTICPROCESS
   - Best fitness: 1.0
2) Confirm if, given a town, country and a price, it is possible to buy coffee and whiskey:
   - Inputs: _COUNTRY, _TOWN, _RECOMMENDEDPRICE
   - Outputs: _COFFEE, _WHISKEY
   - Solution: sequence(TOWNCOUNTRY_HOTEL_SERVICE, HOTELRECOMMENDEDPRICE_COFFEEWHISKEY_SERVICE)
   - List of atomic processes:
     – TOWNCOUNTRY_HOTEL_SERVICE:
       * Inputs: _COUNTRY, _TOWN
       * Outputs: _HOTEL
     – HOTELRECOMMENDEDPRICE_COFFEEWHISKEY_SERVICE:
       * Inputs: _RECOMMENDEDPRICE, _HOTEL
       * Outputs: _COFFEE , _WHISKEY
   - Best fitness: 0.975
3) Obtain the maximum price of a book given the academic item number of the author:
   - Inputs: _ACADEMIC-ITEM-NUMBER
   - Outputs: _MAXPRICE, _BOOK
   - Solution: sequence(ACADEMIC-ITEM-NUMBER_BOOKAUTHOR_SERVICE, AUTHOR_BOOKMAXPRICE_SERVICE)
   - List of atomic processes:
     – ACADEMIC-ITEM-NUMBER_BOOKAUTHOR_SERVICE:
       * Inputs: _ACADEMIC-ITEM-NUMBER
       * Outputs: _AUTHOR , _BOOK
     – AUTHOR_BOOKMAXPRICE_SERVICE:
       * Inputs: _AUTHOR
       * Outputs: _MAXPRICE , _BOOK
   - Best fitness: 0.975
4) Get the maximum price of a book, its type and the recommended price in dollars using the academic item number of the author:
   - Inputs: _ACADEMIC-ITEM-NUMBER
   - Outputs: _MAXPRICE , _BOOK-TYPE , _RECOMMENDEDPRICEINDOLLAR
   - Solution: sequence(ACADEMIC-ITEM-NUMBER_BOOKAUTHOR_SERVICE, AUTHOR_BOOKMAXPRICE_SERVICE, anyorder(BOOK_RECOMMENDEDPRICEINDOLLAR_SERVICE, BOOK_AUTHORBOOK-TYPE_SERVICE)
   - List of atomic processes:
     – ACADEMIC-ITEM-NUMBER_BOOKAUTHOR_SERVICE:
       * Inputs: _ACADEMIC-ITEM-NUMBER
       * Outputs: _AUTHOR , _BOOK
     – AUTHOR_BOOKMAXPRICE_SERVICE:
       * Inputs: _AUTHOR
       * Outputs: _MAXPRICE , _BOOK
     – BOOK_RECOMMENDEDPRICEINDOLLAR_SERVICE:
       * Inputs: _BOOK
       * Outputs: _RECOMMENDEDPRICEINDOLLAR
     – BOOK_AUTHORBOOK-TYPE_SERVICE:
       * Inputs: _BOOK
       * Outputs: _BOOK-TYPE
   - Best fitness: 0.9225
5) Get the weather, map and hotel given the city:
   - Inputs: _CITY
   - Outputs: _WHEATHERSEASON, _MAP, _HOTEL
   - Solution: sequence(split(CITYCITY_MAP_SERVICE, CITY_WHEATHERSEASON_SERVICE, DURATIONCOUNTRYCITY_HOTEL_SERVICE))
   - List of atomic processes:
     – CITYCITY_MAP_SERVICE:
       * Inputs: _CITY
       * Outputs: _MAP
     – CITY_WHEATHERSEASON_SERVICE:
       * Inputs: _CITY
       * Outputs: _WHEATHERSEASON
     – DURATIONCOUNTRYCITY_HOTEL_SERVICE:
       * Inputs: _CITY, _DURATION, _COUNTRY
       * Outputs: _HOTEL
   - Best fitness: 0.9417

Figure 4. Description of the web services compositions used for test.

Table I
AVERAGE RESULTS ($\overline{\chi} \pm \sigma$) FOR THE TEST EXAMPLES

Example	1	2	3	4	5
Time (s)	25,70 ± 17,29	137,91 ± 7,52	167,37 ± 6,31	227,94 ± 26,80	155,73 ± 12,35
$\frac{I^n_{root}}{I_{obj}}$	1,00 ± 0,00	1,00 ± 0,00	1,00 ± 0,00	1,00 ± 0,00	1,00 ± 0,00
$\frac{O_{root}}{O_{obj}}$	1,00 ± 0,00	1,00 ± 0,00	1,00 ± 0,00	1,00 ± 0,00	1,00 ± 0,00
fitness	1,00 ± 0,00	0,98 ± 0,00	0,96 ± 0,02	0,92 ± 0,00	0,92 ± 0,00
depth	2,40 ± 0,52	3,00 ± 0,00	3,60 ± 1,26	8,10 ± 1,91	7,30 ± 2,31
$\#atomicProcess$	1,00 ± 0,00	2,00 ± 0,00	2,00 ± 0,00	4,00 ± 0,00	3,10 ± 0,32
$\#sequence$	0,90 ± 0,32	1,00 ± 0,00	1,10 ± 0,32	2,00 ± 0,94	1,10 ± 0,88
$\#split$	0,00 ± 0,00	0,00 ± 0,00	0,10 ± 0,32	0,40 ± 0,70	0,50 ± 0,53
$\#splitJoin$	0,00 ± 0,00	0,00 ± 0,00	0,00 ± 0,00	0,00 ± 0,00	0,00 ± 0,00
$\#anyOrder$	0,10 ± 0,32	0,00 ± 0,00	0,20 ± 0,63	0,50 ± 0,53	0,70 ± 0,67
$\#choice$	0,00 ± 0,00	0,00 ± 0,00	0,20 ± 0,42	0,30 ± 0,48	0,70 ± 0,82

10 runs and σ is the standard deviation over the 10 runs, which reflects the robustness of the probabilistic algorithm to obtain similar results regardless the followed pseudo-random sequence.

The values that have been used for the parameters of the evolutionary algorithm are: $maxT = 100$, $initialTimesRun = 20$, population size $= 200$, crossover probability $= 0.9$, mutation probability $= 0.03$ (per gene), maximum depth of the tree $= 12$, $\omega_1 = 0.9$, $\omega_2 = 0.05$, $\omega_3 = 0.05$, percentage of the individuals to apply local search $= 1\%$.

The first thing that must be noticed is that, in all the test examples an acceptable solution has been found in all the runs. This means that $I^n_{root} = I_{obj}$ and $O_{root} = O_{obj}$. On the other hand, the maximum possible fitness has been reached for three of the tests, while in the other two the fitness is a 2% lower than the best fitness (Table 4). Going into the details for each test:

- Test 1: this test is very simple, as it is just an atomic process and not a services composition. However, it has been included to verify that also under simple conditions the algorithm works properly (the best fitness was always reached). The number of atomic processes is always the right one, while the depth is slightly over the minimum possible value (2). The number of control structures is always null, except for *sequence* as this node is inserted in the tree by the rule of the start symbol.
- Test 2: in this example all the executions reached the best possible services composition (two atomic processes connected in a sequence).
- Test 3: the number of atomic process that have been used in this composition is always the correct one (2). However the depth has a high variability. This means that some unnecessary nodes (control structures) have been generated.
- Test 4: again, the number and type of atomic processes is correct (always very close to the maximum fitness), but the composition has been obtained with different structures in each of the runs. Therefore, there is variability in the depth and, also, in the type of control nodes.
- Test 5: the number of atomic processes is near the best one (3), but it has been reached with different structures: *sequence*, *split*, *anyOrder* and *choice* have been used to generate the different solutions.

IV. Conclusions

A genetic programming algorithm for web services composition has been presented. The algorithm is able to compound services using different control structures and generates compositions following a context-free grammar. A full validation has been done with a repository of 1,000 services, showing a very good performance. In all the tests and runs a valid solution was found and, moreover, in many cases the best possible composition was obtained. As future work, a pruning method will be added to improve the size of the obtained compositions.

References

[1] F. Curbera, W. A. Nagy, and S. Weerawana, "Web Service: Why and How," in *Proceedings of the OOPSLA-2001 Workshop on Object-Oriented Services*, Tampa, Florida, USA, 2001.

[2] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services*. Springer, October 2003.

[3] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara, *OWL-S: Semantic Markup for Web Services*, World Wide Web Consortium (W3C), November 2004, available at http://www.w3.org/Submission/OWL-S/.

[4] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana, *Business Process Execution Language for Web Services, Version 1.0*, November 2002, standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation.

[5] M. Klusch and A. Gerber, "Fast composition planning of owl-s services and application," in *ECOWS '06: Proceedings of the European Conference on Web Services*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 181–190.

[6] Z. Wu, K. Gomadam, A. Ranabahu, A. P. Sheth, and J. A. Miller, "Automatic composition of semantic web services using process and data mediation," in *Proceedings of the 9-th International Conference on Enterprise Information Systems (ICEIS'07)*, Funchal, Portugal, 2007, pp. 453–461.

[7] P. Traverso and M. Pistore, "Automated composition of semantic web services into executable processes," in *Proceeding of the Third International Semantic Web Conference*, ser. Lecture Notes in Computer Science, vol. 3298. Hiroshima, Japan: Springer, 2004, pp. 380–394.

[8] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau, "Htn planning for web service composition using shop2," *Journal of Web Semantics*, vol. 1, no. 4, pp. 377–396, 2004.

[9] D. Linner, H. Pfeffer, and S. Steglich, "A genetic algorithm for the adaptation of service compositions," in *Proceedings of the 2th International Conference on Bio-Inspired Models of Network, Information and Computing Systems (BIONETICS 2007)*. Budapest, Hungary: IEEE Computer Society, 2007, pp. 277–281.

[10] J. Rao, P. Kungas, and M. Matskin, "Composition of semantic web services using linear logic theorem proving," *Information Systems*, vol. 31, no. 4-5, pp. 340–360, 2006.

[11] L. Aversano, M. di Penta, and K. Taneja, "A genetic programming approach to support the design of service compositions," *International Journal of Computer Systems Science and Engineering*, vol. 4, pp. 247–254, 2006.

[12] U. Kuster, B. Konig-Ries, and A. Krug, "Opossum - an online portal to collect and share sws descriptions," in *Proceedings of the 2th IEEE International Conference on Semantic Computing (ICSC 2008)*. Santa Clara, California, USA: IEEE Computer Society, 2008, pp. 480–481.

[13] L. Eshelman, *Foundations of Genetic Algorithms*. Morgan Kaufmann Publishers, 1991, vol. 1, ch. The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination, pp. 265–283.