A genetic algorithm for the adaptation of service compositions

David Linner, Heiko Pfeffer, and Stephan Steglich
Technische Universität Berlin
Franklinstr. 28/29
10589 – Berlin (Germany)
{david.linner|heiko.pfeffer|stephan.steglich}@tu-berlin.de

ABSTRACT

The view on applications in large-scale open systems shifted to a service-oriented perspective, where each functional feature forming an application is regarded as service. The services which constitute an application can be physically spread over different network nodes and can be even provided by different administrative entities. According to the vision of the BIONETS project we are additionally facing a dynamically changing computing environment, which entails a dynamically changing set of available services. We investigate how service compositions, on which novel applications are based on, can flexibly be adapted to the changing conditions in the computing environment, while going beyond late-binding mechanisms. We apply methods of genetic programming to modify the structures describing service compositions to find compensation for types of services no longer available to applications. In this paper, we describe the current state of our efforts on complex algorithms for service composition transformation, based on the application of genetic operators to graph based service composition representations.

Keywords

Service-oriented Architecture, Service Composition, Genetic Algorithm

1. INTRODUCTION

Large-scale open systems like the Web required a novel view on the design of networked applications and lead to the emergence of service-oriented software architecture styles (SOA). Although the early and trend setting realization of this style (e.g. WSA[1], BPEL[2]) are quite similar to component-oriented technologies the key principle of late-binding implies a significant advancement with regard to dynamicity. Services are selected and bound on demand at runtime. If a service disappears, another one that implements the same functionality is selected. High-level services can be realized by simply composing several services. In this case the so called service composition defines the execution order for the constituting services and rules the flow of data among them. Creating high-level services through service composition saves development time and resources by preserving the advantage of late binding.

In the BIONETS project we are also facing a large-scale open system, but instead of the Web, which is mainly build for reliable networks, the envisioned BIONETS service platform has to cope with high network dynamicity. Hence, the availability of services continuously changes and several types of services may be unavailable for a longer time. While the first challenge can be addressed by late binding mechanisms, the latter requires an approach to substitute service types no matching service is available for. We assume that a service with a particular functionality (certain type) can be exchanged with a composition of services with different types and vice versa. In this regards we do not aim at a perfect substitution of a service type, but rather at a best effort functional approximation through services of other types.

In this work we describe genetic operators to transform service compositions into a bindable form, if for at least one constituting service of the composition no instance is available. In the next chapter related approaches are introduced and discussed. Chapter 2 outlines service model and the service composition model we apply the genetic operators to. Chapter 3 briefly describes how we are going to evaluate the appropriateness of service compositions. In 4 a cross-over operator and a mutation operator are explained. 5 concludes the current state of our work and points out how we will proceed.

1. RELATED WORK

The emergence of service-orientation as architectural view began with the success of Web Services and still Web Services and Business Processes are the driving force for this continuity of this trend. Accordingly, most approaches addressing service composition and dynamic methods for the adaptation of service composition are documented in the Web Services domain. Most of the approaches that utilize genetic algorithms aim at efficiently solving the late-binding problem.

Initially, a service composition is given as blueprint, which describes the execution order and the dataflow for a set of abstract services. The abstract services refer to a type of services with a particular functionality. To execute a service composition each of the constituting abstract services need to be replaced with a real service instance. If one abstract service can be bound with several service instances, a selection has to be made. [3]and [4] describe approaches that use genetic algorithms to find an optimal binding for all abstracts services of a service composition with respect to non-functional properties (e.g. QoS parameters). With our approach we address the step before the actual binding. We utilize a genetic algorithm to vary the structure of the composition itself (execution order and dataflow of constituting services), based on an estimation of services available for a later binding. In this context, the goal is to substitute an abstract service by a finite number of different abstract services or vice versa.

In [5] the actual composition problem is addressed, i.e. automatically creating service compositions (the actual blueprint) from scratch by a combination of AI planning method and genetic algorithms. Although we do not aim at creating entire service compositions, this approach is close to ours with regard to the assumptions about the search space. However, while Yan et al. focus on the inputs and outputs of services and limit their approach to informative services, we introduce the semantic element service *Gain* to also address services changing states without providing a corresponding feedback as output.

2. MODELING AND DESCRIBING SERVICE COMPOSITIONS

Within this section, a graph-based service composition representation is introduced. Therefore, the assumed service model is outlined first, based on semantic service descriptions. Following, *Service Compositions* are defined as a non-empty set of services and two control graphs specifying the workflow and dataflow, respectively.

Services itself are considered as atomically executable parts of application logic, providing outputs as the result of the processing of its inputs, whereby the execution of the service is independent from outer computations and data structures. In order to limit the amount of constrictive assumptions, our service model basically relies on I/O descriptions of services. Those descriptions are initially restricted to primitive data types to ease the transformation of service compositions. Beside, services may generate a special type of semantically described outputs, referred to as *Gains*. Those gains are globally managed within a *Gain Queue*, where they are accessible for all services within a service composition. Those gains are assumed to possess an importance beyond simple I/O passage within the service composition and thereby considerably constitute the services' functionality. For instance, a Restaurant Finder service may get a String as input, specifying the current location of the user by an address within a city. The output of the service then could be a Boolean, indicating whether a restaurant

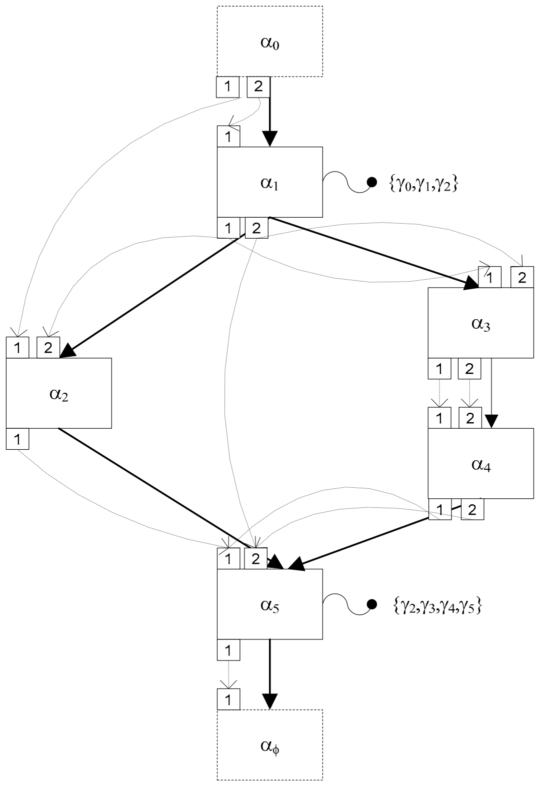


Figure 2: Exemplary Service Composition

```
FUNCTION Cross with composition mother and composition father
  RETURNING composition first_child if stable and composition second_child if stable

  SET weight to proportionate number of transition to cross (0 < weight < 1.0)

  SET transitions_from_mother TO weight * number of transition in mother
  SET transitions_from_father TO weight * number of transition in father

  INIT cross_transitions_mother WITH a random, but less than transitions_from_mother
       number of transition from mother
  INIT cross_transitions_father WITH a random, but less than transitions_from_father
       number of transition from father

  CALL SwapTransitions WITH cross_transitions_mother and cross_transitions_father RETURNING
  first_child and second_child

  CALL RepairPortMappings WITH first_child
  CALL RepairPortMappings WITH second_child
END FUNCTION
```

Figure 1: Crossover algorithm in pseudo code.

was found in, e.g., a radius of 2 kilometers. In case a restaurant was found, a map is pushed to the requesting user indicating the location of the restaurant. Within the presented service model, such a map is specified as a semantically described gain. Notably, this gain is a key feature of the service, specifying its functionality more profoundly than the simple I/O pattern.

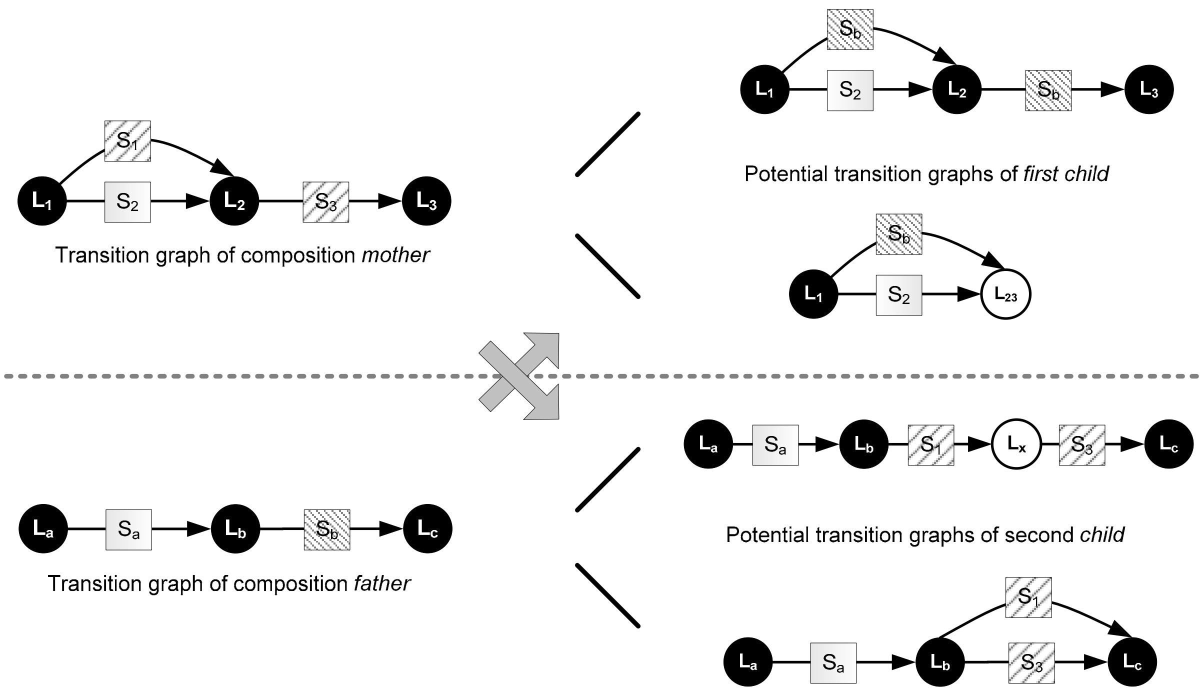
Thus, a service composition can be regarded as a set of services, that are executed in a predefined order and either pass their outputs to other services where they are used as inputs or release them as final outputs of the whole service composition. The passage of outputs within the service composition is based on so-called *ports*, enabling the forwarding of a special output to an input port of another service. An exemplary service composition based on the introduced service model is illustrated in Figure 2. A service is drawn as a rectangle encapsulating an atomic action α_i. The bold arrows define the workflow of the single services, i.e. their execution order. The smaller squares attached to the rectangles on top and on bottom illustrate the input and output ports, respectively. After the execution of a service, its outputs are passed along the dashed lines to input ports of other services. The dashed rectangles denote the pseudo services α_0 and α_ϕ, wrapping the user input and the final output of the service composition. Gains γ_i are annotated by curled arrows leaving the rectangles. Since the generation of gains can depend on the actual input values of the service, a service is always annotated with all gains that can possibly be generated by its execution. Which gains are finally produced by a service execution thus cannot be determined offline, but has to be evaluated at runtime based on specific input values.

Note that services are assumed to encapsulate atomic application logic, which has to be independent from outer computations. In case two services are meant to interact, they thus have to occur multiple times within the service composition. For instance, referring to the previously introduced example, α_1 and α_4 may be the same service, i.e. the service executes a part of its application logic dependent from its inputs and passes some output to service α_3. This service may react to this input, execute some application logic on its own and pass its results back to α_1 (labeled with α_4 in Figure 2).

In order to control the execution and I/O passing within service compositions, two graphs are defined. The *Workflow Graph* is defined as a program graph, specifying the execution order within a service composition. It consists of a set of locations and directed transitions connecting them. Each transition is labeled with a *guard* and an *action*. While the former one is a propositional logic formula ensuring that the transition can only be passed if all inputs of the service and required gains are available, the latter one represents the service logic, whose execution entails the generation of further gains and outputs. The second control structure is built by a *Dataflow Graph*, containing the same locations as the according Workflow Graph; it connects output and input ports of services and thereby defines the flow of data during service composition execution.

3. SERVICE COMPOSITION EVALUATION

Automatic service composition is regarded as a very complex problem still to be achieved without drastic assumptions on the computing environment [6]°[7]. However, the main complexity is not entailed by the composition of services itself, but by the automatic generation of appropriate service descriptions that enable a feasible service discovery afterwards. By proposing an algorithm for service composition transformation, we aim at building new



S1
L1
S2
L2
S3
L3
Transition graph of composition mother
Sb
L1
S2
L2
Sb
L3
Potential transition graphs of first child
Sb
L1
S2
L23
La
Sa
Lb
Sb
Lc
Transition graph of composition father
La
Sa
Lb
S1
Lx
S3
Lc
Potential transition graphs of second child
S1
La
Sa
Lb
S3
Lc

Figure 3: Potential results of transitions crossover.

service compositions that are similar to those already available. In case two service compositions are estimated to provide the same or nearly the same functionality, the semantic description of the already present service composition can be overtaken for the newly created one, solving the problem of automatically creating appropriate service descriptions for new built services.

Since the generation of gains during the execution of a service may depend on the actual service inputs, services cannot be evaluated offline with regard to their functionality (which is considerably dependent from the gains it produces), but have to be estimated during runtime. Therefore, newly created service compositions are compared with existing ones in a *Silent Execution Mode*. Therefore, two compositions are executed within a Sandbox [8] like environment, where their execution does not directly affect the current state of the computing environment. Depending on a distance function taking both generated outputs and gains of a service composition as parameters, the compositions are evaluated according to their similarity. In case the distance between two compositions is small enough, the semantic descriptions are overtaken.

4. SERVICE COMPOSITION TRANSFORMATION

In the last section we showed how we are going to estimate the appropriateness of service compositions for solving particular problems. In this section we discuss the modification of service compositions by genetic operators. Since service compositions are described as graphs, the modification of a service composition is realized as a structural transformation on its graph representation. The aim of the transformation is to vary the service composition structure in order to:

1. Replace service types by more accurate ones, with respect to the addressed problem
2. Substitute services referred in the workflow, when no corresponding service instance is available in the current computing environment

For these purposes we started developing two genetic operators to be applied to service composition graphs as introduced in section 2. The first one is a *crossover* operator, which involves at least two service compositions. The second one is a simple *mutation* operator to be applied to single service compositions. In the following the underlying algorithms are briefly explained.

4.1 Crossover

The crossover operator takes two service composition graphs, in the following referred to as *mother* and *father*. The service compositions should be known to perform semantically similar tasks to forward the usefulness of the result. The operator randomly selects elements from both graphs and exchanges them. In practice, the algorithm for the crossover operator is composed of two steps. In the first step service references (given as transitions) from both compositions are selected and swapped, in the second step the dataflow is repaired. The overall algorithm is summarized as pseudo-code in Figure 1.

```
FUNCTION RepairPortMappings WITH composition
  CLEAR port mappings of composition from removed transitions
  SET preferred_output_ports to transition output ports that are not part of any port mapping to an
  inport port

  FOR each new transition target_transition in composition
    CALL RepairInputs with target_transition and target_transition RETURNING candidate_port_mappings
    ADD one port mapping for each input port of target_transition in candidate_port_mappings TO
    composition, while preferring those with an output port contained in preferred_output_ports
  END FOR

  FOR each transition target_transition in composition whose input ports are not target of any port
  mapping in composition
    CALL repairInputs WITH target_transition and target_transition RETURNING candidate_port_mappings
    ADD one port mapping for each input port of target_transition from candidate_port_mappings TO
    composition, while preferring those with an output port contained in preferred_output_ports
  END FOR
END FUNCTION

FUNCTION RepairInputs WITH target_transition and current_transition RETURNING candidate_port_mappings
  FOR each directly preceding transition pre_transition of current_transition
    FOR each input port input_port of the service in target_transition
      IF service of pre_transition has an output port output_port with the same type as input_port
      AND there is no port mapping in composition to input_port so far THEN
        ADD new port mapping from output_port to input_port to candidate_port_mappings
      END IF
    END FOR

    CALL RepairInputs WITH target_transition and pre_transition RETURNING new_candidate_port_mappings
    ADD new_candidate_port_mappings TO candidate_port_mappings
  END FOR
END FUNCTION
```

Figure 4: Algorithm to repair dataflow graph in pseudo code.

For swapping transitions between the workflows a random number of transitions form both graphs is selected. These two sets of selected transitions are cut out of the graphs they originated from. The set of selected transitions (i.e. the referenced services) form the composition graph *mother* is then pasted into *father* and the set of selected transitions from *father* is pasted into *mother*. For this pasting the locations of the cut out transitions are used in order to recreate a consistent workflow. The transitions pasted into the workflow are by random arranged sequential or parallel, either. Figure 3 illustrates this exchange of transitions. If there are less transitions to paste available than legs between locations that need to be fixed, transitions are randomly cloned or two locations combined.

After swapping the transition between mother and father, the overall dataflow of the resulting graphs is likely to be broken. Thus, a rearrangement of the port mappings between the old and the new transitions is required. For this purpose, the input ports of all transitions are checked. If they are not assigned to an output port of a transition precedent in the workflow, a back-tracking search in the workflow is performed to find an output port (including the overall composition inputs) with the same type like the input port to be bound. Instead of taking the first match, all candidates are collected and afterwards one is chosen by random. The algorithm is summarized in Figure 4. If nonetheless input ports remain unbound, the entire service composition is discarded.

4.1 Mutation

The *mutation* operator removes information from the service composition or includes new information. In practice, the application of the operator to a service composition may either change dataflow or workflow and dataflow. In the first case, a random number of connections from service output ports to service input ports are broken up and the input ports allocated with correctly typed constants. In the second case, transitions are removed from the workflow or new transitions added. The removal of transitions may cause an inconsistency in the workflow and require the combination of the two locations spanned by the removed transition. If the workflow is consistent again, the dataflow is repaired with an algorithm analog to the one described in Figure 4. If the dataflow cannot be recovered, the mutated service composition is discarded.

5. CONCLUSION AND OUTLOOK

In this work we outlined a model for the description of service compositions and explained two genetic operators tailored to this model. Our work aims at evolving service compositions that cannot be applied in their current computing environment, either because no appropriate service instances can be found or no binding with sufficient accuracy. We implemented prototypes for the genetic operators as well as a simulation environment that gives us the freedom to evaluate the appropriateness of our approach. The first results looked very promising. At the moment we are running test series to get reliable data for later publication.

The evolution of services is a crucial element of the bio-inspired service life-cycle described in [9]. Finally, we plan to integrate the developed principles with a service runtime environment to an autonomic service platform for highly dynamic computing environments.

6. ACKNOWLEDGMENTS

This work has been partially supported by the EC within the framework of the BIONETS project IST-FET-SAC-FP6-027748, www.bionets.eu.

REFERENCES

[1] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, D. Orchard: Web Services Architecture, Available: http://www.w3.org/TR/ws-arch/, 2004

[2] R. Khalaf, N. K. Mukhi, and S. Weerawarana, "Service-Oriented Composition in BPEL4WS," in Proceedings of the 12th International World Wide Web Conference (WWW 2003) Alternate Track Papers and Posters, Budapest, Hungary, May 2003.

[3] C. Gao, M. Cai, H. Chen, "QoS-aware Service Composition Based on Tree-Coded Genetic Algorithm," Computer Software and Applications Conference, 2007. COMPSAC 2007 - Vol. 1. 31st Annual International/ , vol.1, no., pp.361-367, 24-27 July 2007

[4] Di Penta, M., Esposito, R., Villani, M. L., Codato, R., Colombo, M., and Di Nitto, E. "WS Binder: a framework to enable dynamic binding of composite web services". In Proceedings of the 2006 international Workshop on Service-Oriented Software Engineering (SOSE '06). ACM Press, New York, NY, 74-80, Shanghai, China, May 27 - 28 2006

[5] Yuhong Yan; Yong Liang; Han Liang, "Composing Business Processes with Partial Observable Problem Space in Web Services Environments," Web Services, 2006. ICWS '06. International Conference on/ , vol., no., pp.541-548, Sept. 2006

[6] M. H. ter Beek , A. Bucchiarone and S. Gnesi, "A Survey on Service Composition Approaches: From Industrial Standards to Formal Methods". Technical Report 2006-TR-15, Istituto di Scienza e Tecnologie dell'Informazione, Consiglio Nazionale delle Ricerche, 2006.

[7] Koehler, J., Srivastava, B.: "Web service composition: Current solutions and open problems". In: ICAPS 2003 Workshop on Planning for Web Services. p 28-35, (2003).

[8] P. Cicotti, Michela Taufer, and Andrew A. Chien. DGMonitor: "A Performance Monitoring Tool for Sandbox-Based Desktop Grid Platforms". In Third International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO), CD-ROM / Abstracts Proceedings, 26-30 April 2004.

[9] H. Pfeffer, D. Linner, I. Radusch, and S. Steglich: "The bio-inspired Service Life-Cycle": An Overview. Proceedings of the 3rd IEEE International Conference on Autonomic and Autonomous Systems (ICAS'07), CD-ROM, Athens, Greece, June 19-15 200