Search-based Testing of Service Level Agreements

Massimiliano Di Penta, Gerardo Canfora, Gianpiero Esposito
Valentina Mazza, Marcello Bruno
dipenta@unisannio.it, canfora@unisannio.it, gianpiero.esposito@unisannio.it
valentina.mazza@unisannio.it, marcello.bruno@unisannio.it
RCOST - University of Sannio, Via Traiano, 82100 Benevento, Italy

ABSTRACT

The diffusion of service oriented architectures introduces the need for novel testing approaches. On the one side, testing must be able to identify failures in the functionality provided by service. On the other side, it needs to identify cases in which the Service Level Agreement (SLA) negotiated between the service provider and the service consumer is not met. This would allow the developer to improve service performances, where needed, and the provider to avoid promising Quality of Service (QoS) levels that cannot be guaranteed.

This paper proposes the use of Genetic Algorithms to generate inputs and configurations for service-oriented systems that cause SLA violations. The approach has been implemented in a tool and applied to an audio processing workflow and to a service for chart generation. In both cases, the approach was able to produce test data able to violate some QoS constraints.

Categories and Subject Descriptors

D.2 [**Software Engineering**]

General Terms

Performance, Reliability

Keywords

Service Level Agreement, Search-Based Testing, Quality of Service

1. INTRODUCTION

Service Oriented Architectures (SOA) are introducing a major shift of perspective in software system development. Moving away from traditional paradigms, they propose software be used instead of being owned [19]. In addition, SOA promotes the development and the adoption of challenging mechanisms such as automatic service discovery and replacement [17], smart monitoring [3] or QoS-aware composition [22]. While these mechanisms tend to promote the diffusion of SOA, its adoption for the development of business-critical application requires services having high reliability. This raises the need for adequate testing strategies and approaches tailored to SOA. Clearly, a lot can be learned and reused from testing approaches for traditional systems, and in particular for distributed systems, Web applications, component-based systems and for real-time systems. Nevertheless, as pointed out by Canfora and Di Penta [7], there are several issues that make SOA testing different:

1. *lack of observability of the service code and structure* for users and system integrators, causing testability problems;
2. *dynamicity:* a service-oriented system can be described as a workflow of abstract services that can be dynamically bound to concrete services known only at runtime. This poses integration testing issues, in that endpoints to be tested with the workflow might not be known *a priori*;
3. *lack of control:* a user/integrator has no control over the evolution of the services she/he is using. Their behavior can change and the integrator might not be aware of such a change;
4. *cost of testing:* invoking services can imply a cost for the tester, if service usage is charged, and also a waste of resources for the provider.

This paper focuses on the testing of SLAs. A SLA is negotiated between a service provider and a service consumer (i.e., an integrator, or an end-user), and guarantees to the service consumer a given QoS level, sometimes depending on how much she/he is willing to pay for the service usage. In other words, a SLA constitutes a form of contract — often specified using languages such as WS-Agreement [1] — between service provider and consumer, and its violation would cause lack of satisfaction for the consumer and loss of money for the provider. For this reason, before offering a SLA, a service provider would limit the possibility that it can be violated during service usage. This paper explores the use of Genetic Algorithms (GAs) to generate test data causing SLA violations. For a service-oriented system such violations can be due to the combination of different factors, i.e., (i) inputs, (ii) bindings between abstract and concrete services, and (iii) network configuration and server load. In the proposed approach, GAs generate combinations of inputs and bindings for the service-oriented system causing

GECCO'07, July 7–11, 2007, London, England, United Kingdom.

SLA violations. The proposed fitness combines a distance-based fitness that awards solutions close to QoS constraint violation, with a fitness — inspired from what proposed by Wegener *et al.* [20] — guiding the coverage of target statements. The approach has been applied to an audio processing workflow and to a service for generating charts, and in both cases it was capable of generating testing data causing SLA violations.

The remainder of this paper is organized as follows. After a review of the literature in Section 2, Section 3 details the proposed approach and the tool support. Section 4 reports and discusses the case studies. Finally, Section 5 concludes the paper.

2. RELATED WORK

This section discusses related work, in particular work dealing with testing of services and service-oriented systems, and work dealing with the use of GAs for test data generation.

2.1 SOA Testing

Although the research on testing of services and service-oriented systems is still at an early stage, a number of approaches dealing with various testing levels and activities have been developed. Tsai *et al.* [18] and Bertolino and Polini [4] proposed a framework to extend the UDDI (Universal Description, Discovery, and Integration) registry to support Web service testing. As discussed in the introduction, the lack of observability is a relevant issue when testing services. A possible strategy is to perturbate SOAP (Simple Object Access Protocol) messages to check whether the service is robust to these perturbations [16]. A black-box strategy for the generation of test data from XML-schema defined into Web service interfaces has been addressed by Bai *et al.* [2]. They mainly propose the use of equivalence class categories to generate test data, without providing any automatic data generation machinery. Recently, Martin *et al.* proposed a preliminary framework to automatically perform Web service robustness testing [14], while Fu *et al.* highlighted the need for proper testing of exception code [11]. We share with Martin *et al.* the idea of generating test cases to cause failure in some service properties. However (i) we use an evolutionary approach, both from a black-box and from a white-box perspective and (ii) our focus is to violate SLA rather than to let the service crash. When a service evolves, its functionality or QoS can vary, triggering the need for regression testing [6]. Like in reference [6], we focus on QoS, however from a different perspective: while from a regression testing point of view the propose is to ensure that QoS properties are preserved when the service evolves, our aim is to test QoS constraints before advertising the service and negotiating SLAs.

2.2 Evolutionary test data generation

Search-based optimization techniques have been successfully applied to tackle different testing problems, and in particular to generate testing data. Most of the relevant references are reported and discussed in a survey by McMinn [15]. In this context, it is worth to relate our work with approaches dealing with code coverage testing and with stress testing.

Regarding coverage testing, and as it will be explained in Section 3, our GA evolves the population so to cover

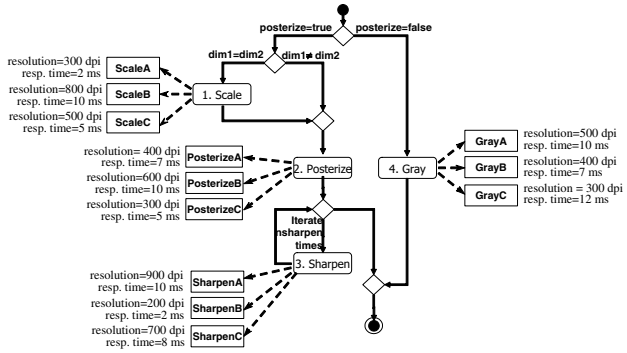


Figure 1: Example of composite service

workflow expensive paths and then tries to violate QoS constraints. To achieve the first objective, we use an approach inspired to what proposed by Wegener *et al.* [20]: their fitness guides the coverage of a target statement by combining the distance (referred as *approach level*) of that statement from the Control Flow Graph (CFG) p-node that paths away from the target statement (referred as *critical node*) with a distance from meeting the critical node condition (referred as *branch distance*).

Wegener and Grochmann applied GAs for testing the temporal correctness of real-time systems [21]. Our black box approach is similar to their one, although considering any kind of QoS attribute, including domain specific attributes (e.g., image resolution), and not just the response time. Our white-box approach combines their approach with the Wegener's distance fitness [20]: QoS-expensive paths are covered, and then the GA evolves the test data to violate QoS constraints. Briand *et al.* [5] used GAs for stress testing of real-time systems. However, in their case the problem was mainly to determine schedule causing failures. In our case, the workflow schedule is fixed and the QoS depends on inputs, bindings and on other external factors. Similarities can be found with the work of Garousi *et al.* [12] They perform stress testing on UML models of distributed systems, also accounting for network traffic. This is also a viable solution for Web services, although it requires the Web service behavior to be modeled as a UML model, which may or may not reflect the actual behavior.

3. APPROACH DESCRIPTION

As mentioned in the introduction, this paper deals with the generation of test data for a service-oriented system[1] causing SLA violations. Let us consider the workflow in Figure 1, representing an image processing composite service. The service takes as input an image in a specific format, characterized by its horizontal and vertical dimension ($dim1$ and $dim2$), a Boolean value indicating whether the image needs to be converted into a poster, and the sharpening level ($nsharpen$). According to the input options, the composite service performs some filtering operations on the image by invoking external services. For each filter (*Scale*, *Posterize*, *Sharpen*, and *Gray*), hereby referred as *abstract services*, some semantically equivalent *concrete services* are available,

[1]That can be on its own a service, mentioned as composite service and described with a workflow.

each one, however, ensuring different QoS (response time and resolution of the output image). Bindings are chosen using optimization approaches, such as those proposed by Zeng *et al.* [22] or by Canfora *et al.* [9], that determine the (near) optimal set of bindings that ensure a QoS constraint satisfaction and that optimize a given objective function.

At execution time, there may exist combinations of bindings and workflow inputs that cause SLA violations, i.e., violations of QoS constraints. Let us consider, for example, that the service provider guarantees to the service consumer a response time less than 30 ms and a resolution greater or equal to 300 dpi. For the workflow in Figure 1, for example, let us consider as inputs $posterize = true$, an image having a size smaller than 20 Mb (which constitutes a precondition for our SLA), $dim1 = dim2$ and $nsharpen = 2$. Let us also consider that the abstract services are bound to *ScaleA*, *PosterizeB*, *SharpenC* and *GrayB*. In this case, while the response time will be lower-bounded by 24 ms and therefore the constraint would be met, the resolution of the image produced by the composite service would be of 200 dpi, corresponding to the minimum resolution guaranteed by the invoked services. In other cases the scenario can be much more complex, in particular when combinations of inputs for each service invoked in the workflow can, on its own, contribute to the overall SLA violation. Violations of some QoS attribute constraints, e.g., response time or throughput, can also depend on the network and server load. Dealing with such a factor is, however, out of scope of this paper and will be considered as part of our future work.

3.1 White Box Approach

Service composition white box testing can be pursued by integrators that, before offering a SLA, want to ensure that the composition is able to meet a given QoS level. They have the composition source code available, written using WS-BPEL or any traditional programming language. The test data generation process is composed of two steps, detailed in the following two subsections.

3.1.1 Step I - Determine QoS-risky paths

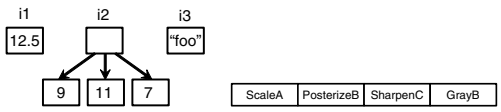
The first step aims to identify which workflow paths (hereby referred as *risky paths*) are likely to exhibit high values for upper-bounded QoS attributes (e.g., response time) and low values for lower-bounded QoS attributes (e.g., resolution). This step is performed by considering (i) QoS value estimates for services composing the workflow, and (ii) estimated upper-bound number of executions for each loop, as declared by the service provider. In other words, the global workflow QoS is estimated according to aggregation formulae defined by Cardoso [10] and then used for binding purposes by Canfora *et al.* [9]. To determine the risky paths for a particular QoS attribute, concrete services having the highest (or the lowest for lower-bounded attributes) value are considered. Since loops are considered to be executed a fixed number of times, the risky paths can be identified by using a linear search, without the need for using any particular heuristic.

3.1.2 Step II - Generating test cases

Once a QoS-risky path has been identified, we use GAs to generate test cases that i) cover the path and ii) violate the SLA. Because expensive paths have been identified according to QoS estimates, while the GA fitness for test

```
<complexType name="ArrayOf_xsd_int">
  <complexContent>
    <restriction base="soapenc:Array">
      <attribute ref="soapenc:arrayType"
            wsdl:arrayType="xsd:int[]" />
    </restriction>
  </complexContent>
</complexType>
...
<wsdl:message name="ServiceOperationRequest">
  <wsdl:part name="i1" type="xsd:float" />
  <wsdl:part name="i2" type="impl:ArrayOf_xsd_int" />
  <wsdl:part name="i3" type="xsd:string" />
</wsdl:message>
```

(a) Excerpt of composite service WSDL



(b) Input encoding

(c) Bindings encoding

Figure 2: Genome representation

data generation relies on measured QoS values, there is no guarantee that test cases violating the SLA also follow the risky paths. Nevertheless, such paths constitute a "starting point" to search for SLA violations. The GA generates new individuals, intended as workflow inputs plus bindings between abstract and concrete services. The bindings are enacted on the workflow by replacing abstract end-points with concrete ones, and then the workflow is executed with the generated inputs. During the execution, the service QoS is observed through monitoring mechanisms and, together with workflow coverage information, is fed back to the GA to permit the individual's fitness evaluation.

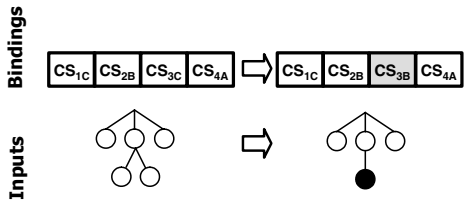
The genome representation is composed of two data structures, as shown in Figure 2:

1. a forest, where each tree represents a composition input, encoded according to the XML schema defining its type, in case it is not a primitive value (integer, float, Boolean, String). Figure 2-b shows a representation for inputs defined according to the WSDL excerpt of Figure 2-a;

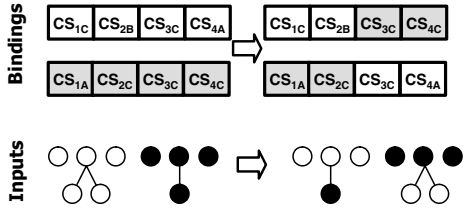
2. an array, containing a slot for each abstract service in the workflow.

The choice of alternative, better representations will be part of our future work. The mutation operator (Figure 3-a) randomly decides if mutating the inputs, the bindings or both. Bindings mutation is quite simple: the endpoint of a randomly selected service is changed to one of the available concrete services corresponding to such an abstract service. Regarding inputs, the operator randomly mutates a (sub)tree of one of the inputs.

- *Sequences* are handled by generating random elements of each type defined in the sequence itself;



(a) Mutation



(b) Crossover

Figure 3: GA operators

- *Choices* here handled by randomly generating one of the elements it defines;
- *Occurrence indicators* are handled by generating a random number of elements between *minoccurs* and *maxoccurs*;
- *Leaves*, i.e., primitive values, are mutated as follows. *Integer* and *floats* are replaced by random values in a range specified by the tester before starting the testing data generation. *Booleans* are mutated between true and false. For *strings* it is either possible to randomly generate a random string bound by a maximum length, or to randomly pick one or more items from a user-defined list (e.g., a dictionary). The former can be used when the service accepts as input any string, while the latter is more useful when the sequence of legal inputs is limited.

It is worth to note that the generated inputs must be, on their own, in agreement with the SLA. For instance, if the service provider guarantees that the service is able to apply a filter on an image of up to 2 Mbytes in 10 s, a larger input should not be considered as part of our testing range, unless one wants to perform robustness testing.

The crossover (Figure 3-b) is also randomly applied to inputs, bindings or both. For bindings, we used the standard one-point crossover. For inputs, a random sub-tree was randomly selected, in the same position, on the two parents and then swapped producing the offspring. Individuals generated by means of crossover and mutation operators are subject to repairing, to ensure they fulfill the XML schema of service input parameters. The selection is made through a roulette wheel selection operator. The type of GA adopted is a simple GA with elitism.

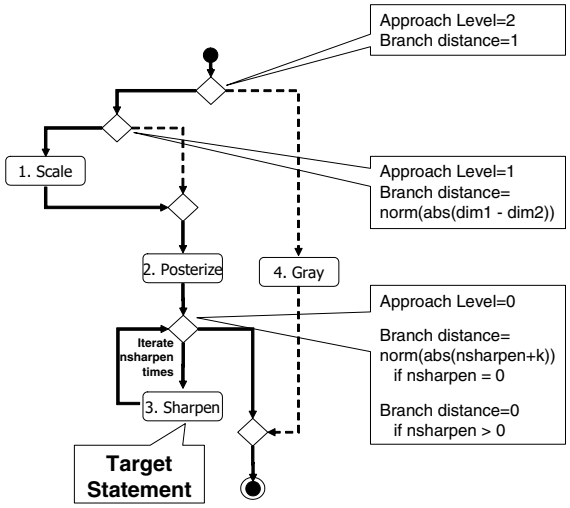


Figure 4: Computing the fitness defined by Wegener et al.

The fitness function accounts for different factors. To let the GA generate testing data that causes SLA violations, the fitness must be a function of how far an individual is from QoS constraint violation. If considering these constraints written in the form:

$$cl_i \leq th_i,\ i = 1, \ldots, n \tag{1}$$

where cl_i is the measured value of the $i-th$ QoS attribute for a GA individual and th_i its upper-bound. The difference between the constraint upper-bound and the actual value is expressed as:

$$D_i = th_i - cl_i \tag{2}$$

Equation (2) holds for upper-bounded QoS attributes (e.g., response time). For lower-bounded attributes (e.g., accuracy) the distance corresponds to equation (2) right-hand-side multiplied by -1. One could just consider such a distance as a fitness, however in many cases the QoS depends on the particular path followed in the workflow and on the particular set of services invoked. Since equation (2) gives no information about the path covered, it might be unable to drive the evolution towards constraint violation. Let us consider a path p estimated to be a QoS-risky path (see Section 3.1.1), and let us consider that, to cover such a path, statements $s_1, \cdots, s_n$ must be traversed. For statement s_j, we can consider a proximity function composed of an approach distance and a branch distance for critical nodes, i.e., for nodes pathing away from the target statement [20]:

$$P_j = dependent - executed + m\_branch\_distance \tag{3}$$

where *dependent* indicates the number of critical nodes the target statement depends on, *executed* indicates the number of these critical nodes that have been executed, and *m_branch_distance* indicates the distance, normalized in the interval [0,1] from satisfying the Boolean condition of the critical nodes pathing away from the target statement. If we consider the example of Figure 1, and we consider that,

for instance, covering the QoS-risky path requires the execution of nodes 1,2,3 (1 time), Figure 4 shows approach level values for risky path statements, as well as the branch distance. The branch distance is 1 if the first conditional (*posterize*) is not satistied, 0 if it is satisfied. For the second conditional *dim1-dim2*, it measures the absolute, normalized distance between the two values (if zero, then the condition is satisfied), while for the third conditional it corresponds to the normalized, absolute value of *nsharpen* (plus a constant) if $nsharpen \leq 0$, while it would be 0 if $nsharpen > 0$. It is important to note that the approach distance works well for flag-free CFGs; in presence of flags proper transformations are necessary [13]. By combining equation (3) with equation (2), we obtain the following fitness function for the $j-th$ path and the $i-th$ QoS attribute:

$$F_{i,j} = \frac{total\_gen - current\_gen}{total\_gen} \cdot P_j + \frac{current\_gen}{total\_gen} \cdot D_i \quad (4)$$

where *current_gen* is the generation when the fitness is evaluated and *total_gen* is the total number of generations fixed for the GA. The two fitness factors are dynamically weighted. First, the fitness gives more weight to the QoS-risky path coverage. Then, it tends to award constraint violation, assuming that it can be pursued once a QoS-risky path has been covered.

3.1.3 QoS Monitoring

To evaluate the fitness function, it is necessary to collect dynamic information, needed to compute factors of equation (3) and the QoS values for the invoked services, needed to compute the distance from constraint violation (2). Approach level and branch distance are monitored through probes inserted by automatically instrumenting (only for testing purposes) the workflow before its execution. For each conditional, we probe the approach function between critical nodes and target statements and the branch distance of critical nodes. To monitor QoS, different approaches are used for different attributes:

- *Response time* and *throughput* are directly measured at execution time;
- domain-specific QoS attributes (e.g., photo resolution, number of colors, etc.) are handled by reading the value from the response of each service invoked, assuming that the service returns the output and its related QoS. For example, an image filtering service can also output, other than the transformed image, the related resolution and number of colors. Then, values are aggregated according to proper formulae defined for combinations of attributes and control statements: for example, the number of colors or the resolution of an image processing workflow is the minimum value guaranteed among the invoked services. Further details on how to handle domain-specific QoS attributes are discussed in a paper by Canfora *et al.* [8];
- in all the other cases, if there is no way to measure the QoS value or to read it from the service output, declared QoS values are read from the interface or from the UDDI registry entry of the invoked services, and then aggregated.

To limit service invocations during the GA execution, every time it is necessary to invoke a service with a set of inputs already used for the same service, the previously monitored QoS is used in place of re-invoking the service. Although this approach has the weakness that multiple invocations can produce (slightly) different QoS values — thus possibly producing some false negatives — if we assume that network and server traffic does not change, it would permit to reduce the waste of resources needed for testing purposes.

3.2 Black Box Approach

The "white box approach" has the advantage of using information about coverage of QoS-risky paths to guide the search towards SLA violation. However, it can only be used by composite service developers or providers having the workflow source code available. When such a condition does not hold — and this may be the case of other integrators or third-party certifiers that want to test the composition before using it — test data can be evolved by only considering the QoS constraint distance, i.e., equation (2), as a fitness function for the QoS attribute of interest.

3.3 Tool Support

The approach has been implemented in a Web-based tool that, due to lack of space, is not thoroughly described in this paper. The tool allows the testers for indicating the service composition — i.e., providing the workflow source code and the set of concrete services that can be bound to each abstract service in the workflow — or the simple service to be tested, and the SLA constraints that the GA should try to violate. For each service, the tool also permits, after having parsed the WSDL interface, the specification of ranges or lists of valid values for the service inputs, so to limit the GA search space. The tool has been implemented in Java (using servlet and JSP technology) and relies on the ECJ library[2] for the GA. The workflow instrumentation relies on the JavaCC[3] parser generator.

4. CASE STUDIES

To evaluate the effectiveness of the proposed SLA testing approach, we applied it to two case studies. This section describes the study context, settings and research, questions, and then it reports and discusses the obtained results.

The first case study is an audio processing workflow, containing invocations to the following abstract services:

- *Converter*: performs audio conversions between different formats (A-Law, μ-law, PCM, etc.);
- *Encoder:* encodes an audio file in *wav* format;
- *Decoder:* decodes a *wav* file in PCM;
- *Amplitude:* amplifies the audio signal.

The above mentioned services were composed in a workflow taking as input an audio file URL, plus some options and performing various audio manipulations (see Figure 5). Each service part of the workflow takes as input a file URL, plus a number of options, and returns a new file URL. For each abstract service, three semantically equivalent concrete

[2] *http://cs.gmu.edu/ eclab/projects/ecj/*
[3] *https://javacc.dev.java.net/*

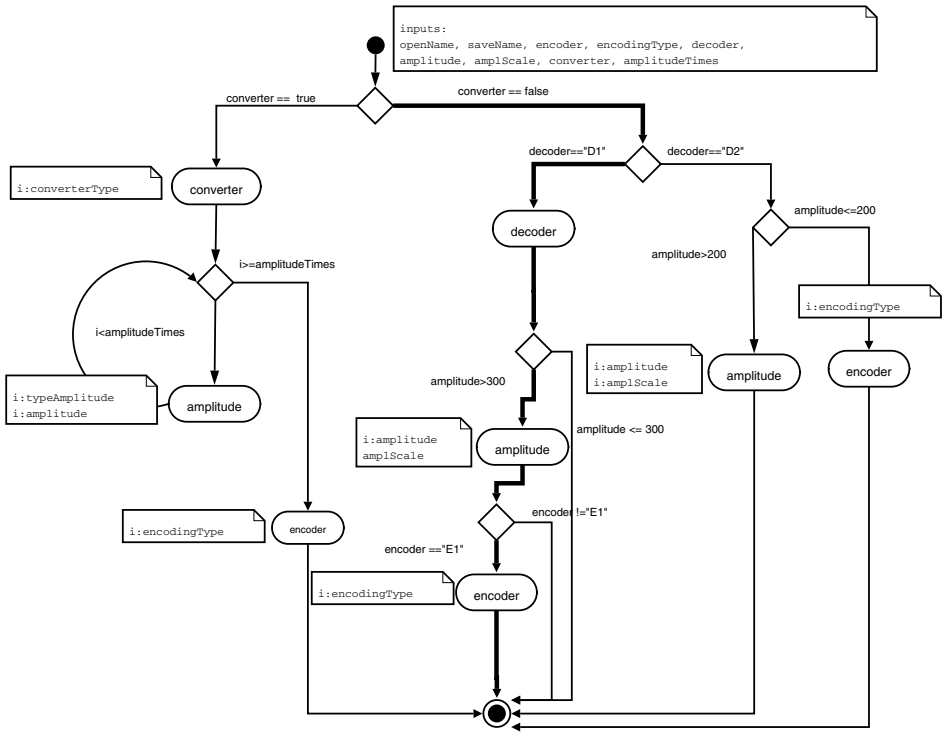


Figure 5: Case study: Audio processing workflow

services are available. They differ for their QoS, in particular they have different *response times* and guarantee different *bit rates* for the output audio file. All these services have been realized by wrapping a library for audio processing[4].

The second case study is performed from a service consumer perspective, i.e., applying the black-box approach to test a service available on the Internet. In particular, we tested the response time property of a service producing charts[5]. Such a service takes as inputs an array of data to be plotted, the chart type (bar chart, pie chart, radar chart, etc.), the height and width in pixels, the output image type (BMP, JPG, PNG, etc.), plus two Boolean values indicating whether the graph should be in colors or single-color and whether the graph should be 3D or 2D.

We used the following GA settings, determined by a trial-and-error procedure: population size of 70 individuals, 100 generations, mutation probability=0.1, crossover probability=0.7. It is important to note that, since we are interested to violate constraints, the stopping criteria could just be the constraint violation, however the paper shows 100 generations to permit a comparison of the different approaches. The type of GA used was a simple GA with elitism for the best 7 individuals. For the first case study GA was run 5 times, to avoid having results biased by randomness. This was not possible for the second case study, in that service usage causes a waste of resources in terms of CPU time and of disk for the provider.

The research questions the case studies aim to answer are the following:

[4] *http://tritonus.org/*

[5] *http://www.berneda.com/scripts/TeeChartSOAP.exe*

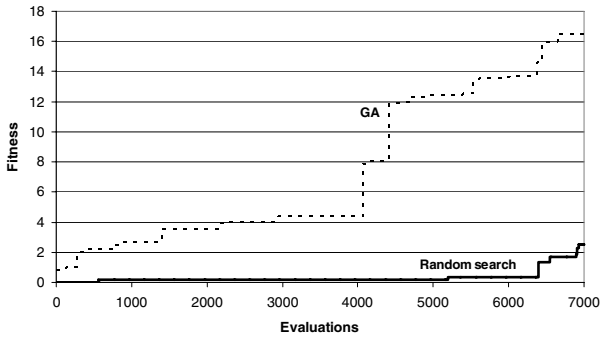
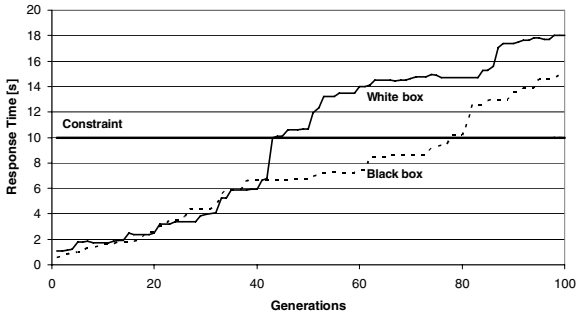


Figure 6: Comparison between GA and random search

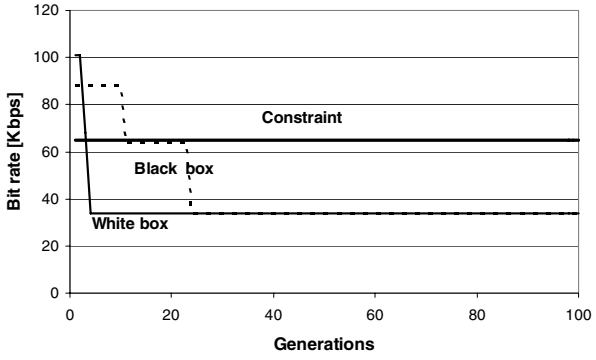
1. *RQ1:* is our GA-based, white box SLA-testing approach better than random search?

2. *RQ2:* is the proposed approach able to generate test cases and bindings that violate SLA constraints?

3. *RQ3:* how the white-box approach performances compare with the black-box ones?



(a) Response time



(b) Bit rate

Figure 7: Audio workflow: white box vs. black box fitness

4.1 Empirical study results

Worst-case QoS estimates for the audio workflow indicate that the path likely to exhibit the highest response time and the lowest bit rate is the one represented with thick edges in Figure 5. As described in Section 3, the algorithm uses the coverage of such a path as a starting point to guide the constraint violation. We fixed a constraint of 12 s (upper bound) for response time and 65 Kbps (lower bound) for the bit rate.

To assess the ability of the fitness to guide the population evolution, it is necessary to compare GA with random search (*RQ1*). Figure 6 shows the white box fitness (4) of the best individual produced by GA and random search, considering for the random search the same number of individuals produced by the GA. In both cases averaged values over 5 runs are shown. Note that the fitness has been inverted with respect to equation (4), i.e., higher values correspond to smaller distances to covering QOS-risky paths and to constraint violation. The Mann-Whitney test performed on best values for the 5 runs after generating 7000 solutions indicates that GA significantly outperforms random search (p-value=0.008).

Figure 7 compares the evolution of the audio workflow

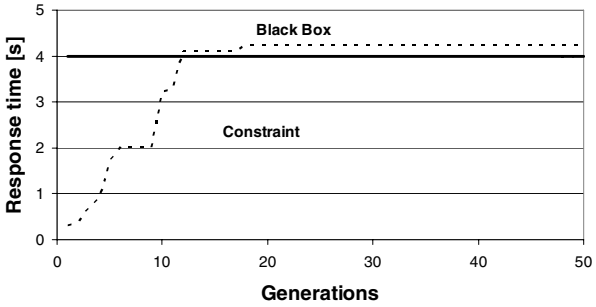


Figure 8: Chart service response time evolution

response time and bitrate for the white box (equation (4)) and black box (equation (2)) fitness. The figure shows average values (response time) and median values (bit rate) over 5 GA runs. While both approaches are able to break response time and bit rate constraints, the white-box approach quickly violates the constraint once the QoS-risky path has been covered. The final value reached for response time is significantly higher as resulted from a Mann-Whitney one-tailed test (p-value=0.017). By looking at the two factors of equation (4) (not plotted in the paper due to space constraints) we noted that the QoS (second factor) remained within the constraint until the coverage of the QoS-risky path was reached. Then, it increased (or decreased depending on the attribute) quicker than for the black-box fitness towards a constraint violation. This provides answers *RQ2* and *RQ3*, i.e., both white-box and black-box approaches are able to generate inputs and bindings that violate QoS constraints, however the white-box approach is able to quickly converge to a solution.

The audio case study permitted the comparison of the different testing approaches, however it gives no information about the capability of the approach to violate QoS constraints for a real service available on the Internet. In this case, however, only a black box approach can be adopted and, as discussed, we limited the GA executions to a single run and the number of generations to 50. Figure 8 shows how the response time increases over the generations until the constraint (4 s) has been violated. This case study increases the external validity of our study, indicating that the proposed approach is able to generate test cases for a real service whose response time depends on the combination of several parameters.

5. CONCLUSIONS

The capability of a service-oriented system to meet the SLA negotiated between the service provider and the service consumer constitutes a critical issue for SOA. This paper presented a GA-based approach to generate test data, intended as combinations of bindings and inputs, causing SLA violations. In particular, the paper proposes a black-box approach, where the fitness awards individuals violating constraints or close to constraint violations, and a white-box approach, combining the black-box fitness with the code coverage fitness proposed by Wegener *et al.* [20] and considering potentially QoS-risky paths as a starting point for searching for violations. Results from a case study audio processing

workflow show that the proposed approach is able to outperform random search, and that the white-box approach is able to converge faster and better than the black-box approach. Also, the approach was able to break response time constraints for a service available on the Internet.

Work-in-progress aims to account for the contribution of network and server load in the QoS, by building service models from monitoring data, also with the objective of limiting service invocations during the testing phase.

6. ACKNOWLEDGMENTS

This work is partially founded by the European Commission VI Framework IP Project SeCSE (Service Centric System Engineering) (*http://secse.eng.it*), Contract No. 511680, and by the Italian Department of University and Research (MIUR) FIRB Project ARTDECO.

7. REFERENCES

[1] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification (WS-Agreement). *http://www.gridforum.org/Meetings/GGF11/Documents/draft-ggf-graap- agreement.pdf*, last accessed: March 9th 2007.

[2] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen. In *WSDL-Based Automatic Test Case Generation for Web Services Testing*, pages 215–220, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[3] L. Baresi, C. Ghezzi, and S. Guinea. Smart monitors for composed services. In *Proc. 2nd International Conference on Service Oriented Computing (ICSOC'04)*, New York, USA, Nov 2004. ACM.

[4] A. Bertolino and A. Polini. The audition framework for testing Web services interoperability. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA 2005), 30 August - 3 September 2005, Porto, Portugal*, pages 134–142, 2005.

[5] L. C. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. In *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1021–1028, 2005.

[6] M. Bruno, G. Canfora, M. Di Penta, G. Esposito, and V. Mazza. Using test cases as contract to ensure service compliance across releases. In *Service-Oriented Computing - ICSOC 2005, Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005, Proceedings*, pages 87–100, 2005.

[7] G. Canfora and M. Di Penta. Testing services and service-centric systems: Challenges and opportunities. *IT Professional*, 8(2):10–17, 2006.

[8] G. Canfora, M. Di Penta, R. Esposito, F. Perfetto, and M. L. Villani. Service composition (re)binding driven by application-specific qos. In *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Chicago, IL, USA, December 4-7, 2006, Proceedings*, pages 141–152, 2006.

[9] G. Canfora, M. Di Penta, R. Esposito, and M. L. Villani. An Approach for QoS-aware Service Composition based on Genetic Algorithms. In *Proc. of the Genetic and Computation Conference (GECCO'05)*, pages 1069–1075, Washington, USA, June 2005. ACM.

[10] J. Cardoso. *Quality of Service and Semantic Composition of Workflows*. PhD thesis, Univ. of Georgia, 2002.

[11] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott. Testing of java web services for robustness. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, pages 23–34, 2004.

[12] V. Garousi, L. C. Briand, and Y. Labiche. Traffic-aware stress testing of distributed systems based on UML models. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 391–400, 2006.

[13] M. Harman, L. Hu, R. M. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Trans. Software Eng.*, 30(1):3–16, 2004.

[14] E. Martin, S. Basu, and T. Xie. Automated robustness testing of web services. In *Proceedings of the 4th International Workshop on SOA And Web Services Best Practices (SOAWS 2006)*, October 2006.

[15] P. McMinn. Search-based software test data generation: a survey. *Softw. Test., Verif. Reliab.*, 14(2):105–156, 2004.

[16] J. Offutt and W. Xu. Generating test cases for web services using data perturbation. *SIGSOFT Softw. Eng. Notes - SECTION: Workshop on testing, analysis and verification of web services (TAV-WEB)*, 29(5):1–10, 2004.

[17] M. Paolucci, T. Kawamura, T. R. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Proceedings of the first International Semantic Web Conference (ISWC 2002)*, volume 2348 of *Lecture Notes on Computer Science*, pages 333–347. Springer-Verlag, June 2002.

[18] W.-T. Tsai, R. J. Paul, Y. Wang, C. Fan, and D. Wang. Extending WSDL to facilitate Web services testing. In *7th IEEE International Symposium on High-Assurance Systems Engineering (HASE 2002), 23-25 October 2002, Tokyo, Japan*, pages 171–172, 2002.

[19] M. Turner, D. Budgen, and P. Brereton. Turning software into a service. *IEEE Computer*, 36(10):38–44, 2003.

[20] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information & Software Technology*, 43(14):841–854, 2001.

[21] J. Wegener and M. Grochtmann. Testing temporal correctness of real-time systems by means of genetic algorithms. In *Quality Week*, 1997.

[22] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5), May 2004.