A Participant Testing Strategy for Service Orchestrations

Flavio Corradini, Francesco De Angelis, Andrea Polini, Alberto Polzonetti
Department of Mathematics and Computer Science - University of Camerino
Via Madonna delle Carceri 9, 62032 Camerino - Italy
{*firstname.lastname*}@unicam.it

Abstract

Service Oriented Computing is an emerging discipline that promotes and make easier inter-organization integration of software systems. In such a context interoperability issues are considered a primary threat for a correct integration. Testing can be a fruitful approach helping to reduce interoperability mismatches. In this paper, considering the case of services to be integrated within an orchestration, we propose a testing derivation strategy based on a counter-example based technique. The state explosion phenomenon, caused by the presence of complex data sets, is handled by applying Genetic Algorithm strategies.

1. Introduction

The Service Oriented Architecture (SOA) promises to enable fast and secure integration of software infrastructures belonging to different organizations. The promise is mainly based on the concept of "service" and on the establishment and development of a suite of open standards fostering service interaction and composition. These standards typically define rules on those aspects which generally affect interoperability and composition, such as message formats or interface descriptions. Indeed the most used technology to implement SOA refers to Web Services (WS), and is based on a quite wide set of agreed and standardised information models and formats typically referred as WS-*.

In such a setting particularly interesting is the possibility of composing services from different organizations, in order to provide higher level functionalities. So, for instance, a travel agency could compose services from different airlines companies, hotels, and museums in order to provide a higher level service permitting to plan holidays in a given place for a defined time period.

In such a setting communication standards can certainly help to reduce risks of interoperability mismatches. Nevertheless they can not prevent application related mismatches. These refers to possible different assumptions made by the interacting services on their mutual behaviour. With reference to the travel agency service this would mean that standards certainly help to reduce the risk that incompatible data and message format will be used. Nevertheless in case the travel agency assumes a different functional behaviour from the airlines company, for instance assuming that reservations still on "confirmation pending" status, from the airline company perspective, are indeed confirmed booking.

Different approaches can be applied to deal with application related interoperability mismatches in a open environment, i.e. in which services can enter and exit at any time. In restricted application domains one possibility is certainly that of raising the standardization level to the application level. Nevertheless the most general solution is to apply techniques permitting to highlight behavioural mismatches.

Testing is certainly one of the most used approach to highlight integration problems through the use of integration test suites. Nevertheless testing is an engineering activities that must be always reconsidered when applied in a new context. Testing should try to take profit of new opportunities for control and observation, and should address emerging hurdles. For instance run-time integration and discovery is certainly one of the major hurdle for testing service composition. The result is that in a certain sense the full application is never available before run-time. At the same time the availability of formal models, describing service integration, can be exploited in the testing activities such as test cases derivation.

In this paper, with reference to orchestration of services, i.e. a special kind of service composition in which a special process, the orchestrator, takes the role of coordinator, we present an approach for test cases derivation based on the formal manipulation of the orchestration definition. In particular the approach combine model checking and genetic algorithms techniques to derive test cases to be used in order to check the behaviour of the service that will play a role within a given orchestration. In particular genetic algorithms techniques are used in order to deal with data generation and to avoid state explosion related issues.

Next section provides some background material related

978-1-4244-2917-2/08/$25.00 ©2008 IEEE

to model checking and genetic algorithms. Then in Section 3 we describe the steps composing our approach and in Section 4 we illustrate the application of the approach with a toy example. Finally we draw some conclusions and opportunities for future work in Section 6.

2. Technical Background

This section shortly introduces some background material required to better understand the approach proposed in this paper.

2.1. Service composition

Integration of services available over the Net in order to perform more complex tasks is certainly one of the most appealing characteristics of the Service oriented approach.

Service integration can be mainly pursued through two different and orthogonal approaches. The first one foresees the description of the interaction of many services on a peer basis. Such kind of description, typically referred as choreography, describes the actions that a set of interacting services has to carry on to fulfill a given task. In such a setting involved services interact with each other according to what is specified for the corresponding role in the choreography. In case one of the participating service does not stick to the expected behaviour the whole choreography could fail.

Another relevant paradigm to service integration foresees the presence of a service that acts as the coordinator of the integration and invokes the different services involved in the integration in order to reach the final goal. In this case services are not integrated on a peer basis and relevant interactions, reported in the orchestration description, are only those referring to communications among the orchestrator and the integrated services.

The second approach to integration is certainly the most mature one and the most used in practice. An orchestration can be defined using many different languages, being traditional programming languages one possible choice. Nevertheless the Web Service-Business Process Execution Language (WS-BPEL) [2], seems currently to have the greatest chances to become the leading standard for orchestration description. This is a language based on a XML syntax that can be used to define executable orchestration models.

From the testing perspective service orchestration presents important challenges due to the fact that integrated services could belong to different external organizations. Moreover it is possible that service participanting to an orchestration are discovered and bound only at run-time. On the other side the orchestration provides a model of foreseen service interactions that could be fruitfully exploited for test cases derivation. Indeed, our idea explore the possibility of using model checking related techniques, on the orchestration description, for test cases derivation.

2.2. Model Checking and Genetic Algorithms

Model checking [6] is a very effective technique to deal with formal analysis and verification of complex software system specifications. A model checker is able, given an operational model of the system, and a property defining specific requirements on the same model, to show if the property actually holds or not. Moreover, in case the property is not satisfied the model checker points to a counterexample showing a precise model execution violating the property.

Since its first inception many tools have been proposed and developed to carry on model checking, nevertheless all of them share the same principle. In particular the system is typically represented by an operational model defining a system state space and the transitions among such states. The property instead is generally specified through a logical formula such as a Linear Temporal Logic (LTL) formula ([7]). Given the model and the formula, the model-checker exhaustively explores all the possible successive system configurations looking for possible violations of the formula. When a violation is detected, the model checker reports the sequence of decisions and the actions that it took to reach the falsifying configuration. One of this sequence is generally referred as a counterexample.

The possibility to exhaustively explore the state space for a falsifying configuration results particularly appealing also for testing purpose. The general idea here is to derive test cases from counter-examples of negation of expected properties. For instance stating that does not exist a configuration in which a call will be performed with a given value will result in a counterexample showing how to reach a similar configuration.

Beside the great power of this techniques the application of model checking is often made complex, if not impractical, as consequence of the well known state-explosion phenomenon. This problem arise when the number of possibly different system configurations becomes too big to allow a complete state exploration. To overcome this problem many heuristics have been proposed in order to avoid the necessity of a complete state space exploration, in some cases giving place to almost-correct approaches. The state-explosion problem is particularly relevant when the model is augmented, as it is for the case considered in this paper, with information concerning data.

Research on techniques to reduce the state-explosion phenomenon is a very relevant and investigated problem. In general using additional models and taking additional assumptions different sets of states are merged together. When the state explosion phenomenon is due to the inclusion of complex data within the model one possibility is to explore different versions of the same models derived through the selection of specific values for the included complex data. Clearly this strategy used to select the values strongly affect the quality of the resulting verification phase. As discussed in the next section in our approach the selection strategy applies genetic algorithms techniques.

Genetic algorithms [9] are a technique that mimics natural-evolution processes to find approximate or exact solutions to search problems with large solution spaces, and that are not amenable for exhaustive search. They use concepts from evolutionary biology like inheritance, mutation, selection, and recombination to evolve candidate solutions (represented by chromosomes) toward a solution for the problem.

To apply a genetic algorithm approach it is necessary to represent the solution in terms of "chromosomes" or "individuals", organized as a "population", and to define a *fitness function* to evaluate the goodness of the various solutions identified during the execution.

From an initial solution the algorithm proceeds generating new potential solutions continuously evaluating them until the fitness function is maximized or another halting condition is reached. Each new population is generated starting from the preceding one combining the various "individuals" and applying genetic operators. Major issue with this kind of algorithms is the design of the structure of the initial solution and the method for its evaluation using the fitness function as the objective. Other relevant parameters concern the size of the population, the admitted recombination techniques, and the individuals mutation rate.

3. The Approach

A service orchestration describes the integration of a set of services to provide higher level functionalities. Such a description typically defines a sequence of data manipulation and direct invocations to a set of services. Testing can play an important role, in such a setting, as a mean to verify if a service, to be integrated in the orchestration, actually behaves accordingly to what expected by the orchestrator. This is a difficult topic within the service domain also due to the fact that services are often run by third party organizations and can also be discovered and bound only at run-time. The technique we propose aims at deriving test cases to assess services to be integrated in an orchestration. Nevertheless we do not specify a whole testing process which still has to emerge within the service community. The test cases we define can be used to assess a service off-line or even on-line, but before the final binding, when suitable support is provided.

Different techniques can be used to derive test cases to test services involved in an orchestration. As usual the selection of a black-box test case derivation technique is strongly related to the modeling techniques used to define participants behaviour. Indeed WS standards, such as WSDL, typically convey only syntactic information and do not leave much room for the application of testing strategies except for those based on data definition such as partition testing.

In the approach introduced in this paper we assume that invocations made by the orchestrator to participant services are annotated with pre and post-conditions. This assumption is inspired to the work of Baresi et al. [3] which use invocation annotations to permit run-time monitoring of participant services. Indeed in the mentioned work the authors also define a specific language, called WS-CoL, suitable to express constraints on service invocations.

Availability of annotations for service participant invocations makes applicable contract-based test case derivation strategies. Nevertheless such an approach would not take into consideration the implications deriving from the integration of all the other services involved in the orchestration. In particular the presence of other services could make not possible some invocations specified by the constraints. Indeed in order to reduce and derive more powerful participant test suites is important to consider the orchestrator and the other participants also in the derivation of test cases; instead of considering only the model of the single service. In particular it is our intuition that the best test suites are those composed of test cases corresponding to execution of the orchestration code involving the greatest number of participants. Main objective of our test case derivation strategy is to identify this kind of executions and derive from these executions test cases to be used to assess the various participants in isolation. The necessity of having test suites for participant services strongly connoted with characteristics coming from the final environment is a particularly important one since, within a service computing domain, it is in general not possible to perform a real integration testing phase. The final application is often integrated only at run-time.

The strategy we propose is organised in successive steps and a subset of them have to be executed more than once until the "best" test suite (according to the defined adequacy criteria) has been identified.

Orchestration invocations annotation (1) first step of the approach is the annotation of the orchestration definition with constraints for the invocations to participant services. In order to apply the approach these annotations should identify finite sets for the acceptable values. This limitation is mainly consequence of the application of a model-checking based technique as discussed in point 3.

To express the constraints we include java statements within the BPEL specification. This choice will make easier the transformation of BPEL to java code as described in the following point.

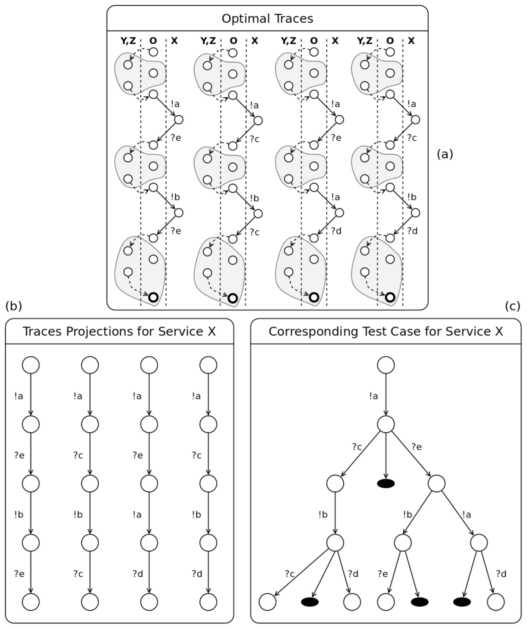
Optimal Traces
Y,Z
O
X
!a
?e
?c
!b
?d
(a)
(b)
(c)
Traces Projections for Service X
Corresponding Test Case for Service X

Figure 1. From traces to test cases

From BPEL to Java (2) second step of the approach is the transformation of the orchestration definition into a format suitable for model checking processing. We have developed a small proof of concept tool that is able to translate a BPEL orchestration into Java code. This choice is mainly related to the decision of using the model checker Java Path Finder (JPF). Nevertheless this is not the only possible option. For instance in [4] the authors describe a quite mature tool to transform a BPEL definitions into BIR code which is the input format for the BOGOR model checker.

Input data generation (3) Once the annotated orchestration has been defined the next step concerns the definition of a set of starting values (population) for the parameters defined for the service exposed by the orchestration. A specific assignment of values for the input parameters identifies one possible execution of the orchestration. Therefore a set of assignments identify a set of executions of the orchestration. Some of this execution are, from the testing point of view, good and some are not. In our work we define the quality of an assignment in terms of the service participants interactions that the assignment is able to stimulate. After that a set of assignments has been evaluated we apply genetic algorithms techniques in order to recombine the values in the current population to derive a successive version of the population. The strategy we use for such derivation removes half population and maintains in the next version of the population those value assignments permitting to reach the greatest score in terms of covered "orchestration-participants" interactions. The remaining half population is derived recombining the values of the most promising assignments, i.e. those assignments that individually get the greatest scores. It is worth noting that the initial population has to be directly defined by the orchestration developer or possibly defined using a random strategy. As it is the rule for genetic algorithms based strategies the starting population strongly affect the quality of the obtainable results.

Counterexample identification (4) Given an initial assignment for the orchestration, the successive step applies a counterexample-based technique to identify interesting execution to be used for testing purpose. To carry on this step we suggest to use a model-checker providing to it properties specified in terms of reachability constraints. These properties state that some orchestration exit points are not reachable given the defined value assignments and constraints on participant services invocations. As a result the model-checker will explore the execution state space and will return possible executions (counter-examples) that indeed are able to reach the defined exit points. Figure 1(a) graphically reports possible traces for the case of an orchestration (O) involving three services (X, Y, Z). An execution trace is constituted by a sequence of invocations made by the orchestrating service to all the services involved in the orchestration.

We experimented the approach using Java Path Finder (JPF) that is a model checker tool that takes as input programs written in java. This tool checks reachability by default (i.e. in case you do not provide a property it will check if it is possible to reach an exit point). At the same time JPF solves the constraints specified for the invocations looking for a counterexample. So, it will check with all the possible correct outputs.

Trace evaluation (5) Once the model checker has identified a trace this has to be evaluated to assess its "quality". In our approach this is defined in terms of the number of interactions of the orchestrator with the participant services highlighted by the trace. After that all the traces derived from a population has been evaluated we can calculate the quality of the whole population. In case the required quality has not been reached and the number of iterations is lower than a given threshold or the quality did not increase in the last *n* iterations, the process goes back to step 3.

From traces to test cases definition (6) The final result of the previous steps is a set of execution traces. Objective of this step is instead the definition of a test suite for the various participants. To do this we need to operate a projection on the derived traces. The projection aims at identifying the interactions of the orchestrator with one specific participant service (the process will be then repeated for each participant). In Figure 1(a) we have highlighted the interactions of the orchestrator with service X. For the sake of comprehension we modeled service X with an Input-Output Transition System (IOTS). In particular X accepts messages `a` and `b` and can reply with messages `c`, `d`, and `e`. Nevertheless a constraint defined on the first invocation declares that after a first invocation a possible correct answers are only c and e. According to this and other constraints the model checker will generate a set of traces of which a subset is shown in Figure 1(b). In this figure we can observe that no traces show an answer d after invocation a.

From a set of traces we derive then an equal number of projections. At this point a test case will be defined as a decision tree derived through a synthesis of the different traces. In particular to derive the decision tree we start considering all the projections starting with the same invocation. Then we recursively add all the possible interactions shown by any of the defined traces, and we continue until all the traces reach the end node. Finally in each state in which the tree waits for a message from the service we add a sink state indicating that all the answers not highlighted by any trace lead to a failure, and then to the identification of a fault in the service under test. Figure 1(a) shows the test case that has been derived applying the algorithm to the projections in 1(b). Executing this test case a test driver will start sending message a waiting then for an answer from the service. Depending from the observed output the service will follow a different path in the tree. At the same time when more than one message can be sent to the service under test the driver will randomly select one of them.

Participant test suite definition (7) a test suite for a participant is derived through the application of the algorithm sketched in the previous step to all projections starting with any possible message. The process is also repeated for each participant. Therefore at the end of the process a test suite to evaluate each participant will be available.

4 Explanatory example

In this section we present an example of data input generation for a simple orchestration that mimics the calculation of triangle areas. Inputs to the orchestration are represented by the length of the edges e_1, e_2, e_3. To find the area the simple formula $S = bh/2$ is used where b is the length of the base of the triangle, and h is its height. The equilateral triangle is an exception because to know its area is enough the formula $e^2\sqrt{3}/4$ where e is the length of an edge.

To solve the problem the first step in the orchestration is to determine the nature of the triangle. If the triangle is equilateral the determination of the area is straightforward instead if the triangle is isosceles or scalene we must decide which edge is considered the base and find the corresponding height. Moreover, a non-equilateral triangle can be right-angled. This simplify the choice for base and height because these can be represented by the two edges leaving from the right angle. These two edges can be found easily being the hypotenuse the longest edge.

Three services are used to solve the problem. One service is exploited to classify the triangle using the lengths of its sides. A triangle can be equilateral, isosceles or scalene. Moreover, a non-equilateral triangle can be right-angled. Another service is used to choose one edge to be consider as the base and to calculate the height of the triangle from that base. This service is used only if the triangle is isosceles or scalene and there is not a right angle. Finally, a third service simply implements a method to calculate the area. This service have a method with respectively one or two arguments. If only an argument is given the used formula is valid for equilateral triangles, else if two arguments are given they represents the base and the eight. In this case the usual formula applies.

The orchestration starts with e_1, e_2, e_3 as inputs. The first task performed inquiries the first service to determine the type of the triangle. If it is equilateral the area is calculated inquiring a method from the service who calculate areas with one arguments. If the triangle is not equilateral, we inquiry the first service to know if it is right-angled. If it is, we cut of the highest edge and use the remaining edges as base and height, else if the triangle is not right-angle the service who calculate base and height is invoked. In both cases the method that calculates the area is invoked with two arguments.

Although the orchestration is very simple the automatic determination of inputs for testing can lead to the trouble of having three edge that do not shape a triangle. This situation arises when, given three edges, there is an edge that is greater to the sum of the other two. In this situation there is an exception thrown during the determination of the triangle type. Moreover, we must test all the situations representative of the triangle types we can found included those in which all the services are involved. For this random generation is not enough.

Genetic algorithms generation of data manages quite well this problem because tuple of data are generated to maximize the coverage of the orchestration i.e. to maximize the number of situation we can encounter handling triangles. In our example the best coverage we can have is given by three triangles (one equilateral, one right-angled, one isosceles or scalene) that considered as a whole can cover all the possible interaction between the involved services.

The genetic algorithm can start for example with the tuple $\langle 10, 8, 6\rangle$, $\langle 10, 14, 15\rangle$, $\langle 5, 10, 10\rangle$ but this not assure the best coverage (we have two scalene and one isosceles but does not consider the right-angle case nor equilateral triangles). After recombination, possibly in more than one iteration, we can have $\langle 10, 14, 15\rangle$, $\langle 5, 10, 10\rangle$, $\langle 10, 10, 10\rangle$ and we have improved the coverage including the equilateral case but we have not reached the targeted coverage. It is worth noting that the scalene and the isosceles cases will be maintained in the population given the greater score of this subset. Finally, after some further recombinations we can have $\langle 10, 14, 15\rangle$, $\langle 10, 14, 10\rangle$, $\langle 10, 10, 10\rangle$ considering all the possible cases. Also in this case in each iteration the equilateral and isosceles triangle will not be removed. These data induce several traces from which our technique is able to extrapolate test cases with their relative test data.

5 Related Works

Although testing is a very important engineering activity there are not many works in literature addressing service testing and in particular BPEL testing. One exception is *BPELUnit* [11]. BPELUnit is a XML-based test framework for BPEL composition that works with a custom language for test specifications and provide a java test runner that can be extended to support multiple engines, such as Active BPEL and Oracle BPEL Server. Nevertheless BPELUnit does not focus on test cases derivation.

Derivation of test cases from counterexamples has been firstly proposed by [1]. Verification of web services by model checking and testing is investigated in [10] where the BLAST model checker is used and Web Services are modeled using OWL-S specifications.

Genetic algorithms (GA) are global search-based strategy that has been proved suitable for software testing in several works [13] [14]. A GA-based system for dynamic test data generation is described in [12] where the problem of test data generation is reduced to the one of minimizing a function. A discussion on combining model-checking and genetic algorithms can be found in [8], where a framework for exploring very large state spaces is presented.

6 Conclusions and Future Work

Service Oriented Architecture and Web Services are respectively an emerging discipline and a technology that promise to revolutionize the way in which organizations will interact. As for any new domain is necessary to reconsider all the development activities to understand which are the new opportunities and the new challenges. This is certainly true also for testing activities. In this paper we proposed a novel approach to testing of those services to be integrated by a service orchestrator. Defining the approach we tried to take advantage from the higher availability of formal models and standard formats. At the same time we tried to identify which one could be the most promising test suite to be executed on a service participant. Our intuition is that the best test suites could be derived from those executions that show the greatest number of run-time interactions among the integrated services and the orchestrator.

The discussed approach combines Model Checking and Genetic Algorithms techniques in order to identify the execution traces to be used for test cases derivation. The combination of this two techniques permits to reduce the risk of the occurrence of the state-explosion phenomenon. The approach is certainly time consuming given that model checking have to be executed many times. Nevertheless the process has to be executed only once given an orchestration.

The research is certainly still on going and the approach has to be validated on bigger and more realistic case studies, nevertheless some encouraging results have been already obtained on some small examples, in particular within the e-government domain [5]. Particularly critical seems the validation of our intuition that suggests to search for test cases within those executions showing the greatest number of interactions. If this assumption could seem reasonable it is necessary to validate it with a more systematic study aiming at comparing different techniques.

References

[1] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specifications. In *ICFEM 1998.*

[2] T. Andrews et al. Business Process Execution Language for Web Services, Version 1.1. *Standards proposal by BEA Systems, IBM, and Microsoft Corp.*, 2003.

[3] L. Baresi and S. Guinea. Dynamo: Dynamic monitoring of ws-bpel processes. In *ICSOC 2005*, pages 478–483, 2005.

[4] D. Bianculli, C. Ghezzi, and P. Spoletini. A model checking approach to verify BPEL4WS workflows. In *SOCA 2007*, pages 13–20, 2007.

[5] F. Corradini, F. De Angelis, A. Polini, and A. Polzonetti. Improving Trust in Composite e-Services via Run-Time Participants Testing. In *7th Int. EGOV Conf.*, Torino (Italy), 2008.

[6] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[7] A. E. Emerson. Temporal and modal logic. *Handbook of theoretical computer science (vol. B): formal models and semantics*, pages 995–1072, 1990.

[8] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. *Int. Journal on Soft. Tools for Technology Transfer (STTT)*, 6(2):117–127, 2004.

[9] J. H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[10] H. Huang, W. Tsai, R. Paul, and Y. Chen. Automated model checking and testing for composite web services. In *ISORC 2005*, pages 300–307, 2005.

[11] P. Mayer and D. Lübke. Towards a bpel unit testing framework. In *TAV-WEB '06*, pages 33–42, New York, NY, USA, 2006. ACM.

[12] G. Mcgraw, C. Michael, and M. Schatz. Generating software test data by evolution. Technical report, Reliable Software Technologies, Sterling, VA, 1997.

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

[14] J. Wegener, A. Baresel, and H. Sthamer. Suitability of evolutionary algorithms for evolutionary testing. In *COMPSAC 2002*, pages 287–289, 2002.