A Model Checking Strategy to Test Services in Orchestrations

Flavio CORRADINI¹, Francesco DE ANGELIS¹, Guglielmo DE ANGELIS², Andrea POLINI¹, Alberto POLZONETTI¹,

¹Department of Mathematics and Computer Science, University of Camerino, Via Madonna delle Carceri 9, Camerino, 62032, Italy
{flavio.corradini, francesco.deangelis, andrea.polini, alberto.polzonetti}@unicam.it
²ISTI-CNR, Via G. Moruzzi 1, Pisa, 56124, Italy
gugliemo.deangeli@isti.cnr.it

Abstract: Very late binding, run-time integration of software elements owned and managed by third parties, run-time changes. These are just some of the characteristics of the service oriented computing paradigm which strongly affect static and dynamic verification capabilities. In this domain verification and testing communities have to face new issues and revise existing solutions; possibly profiting of the new opportunities that the new paradigm makes available. In this paper, focusing on service orchestrations, we propose a novel approach to automatic test case generation aiming in particular at checking the behaviour of services participating to a given orchestration. The approach exploits the availability of a runnable model and uses model checking techniques to derive test cases suitable to detect possible integration problems.

Keywords: Counter-example based Testing, Service Testing, Service Orchestrations

1. Introduction

Service Oriented Computing (SOC) is a software production paradigm for which the maturity of the technology is rapidly increasing. Furthermore, it is getting more and more technically easy to implement, deploy, and compose a software system based on service integration.

In very general terms testing is an activity which tries to provide some evidence on the correctness of a piece of software. This objective is pursued performing a finite set of experiments and checking if the observed effects are in line with those expected.

The SOC domain seems to have specific characteristics that implies ad-hoc testing strategies. Usually, services are owned, controlled, and run by different organizations. Moreover, service integration often happens just at run-time, possibly without any previous investigation on its possible effects. As a result it would be useful to apply verification activities also at run-time. Therefore, if unit (service) testing does not seem so much affected by the service oriented computing paradigm, this is certainly not the case for integration testing activities.

Our approach address this problem considering the behavioral aspect of the service without considering non-functional properties. Behavioral run-time integration of services possibly owned by different organizations still presents unsolved issues, and in our opinion a lot of research is still needed in this domain regarding testing activities. The research for solutions could follow many different paths possibly involving different research communities [1, 2].

This paper focuses on a strategy for automatic test case derivation for orchestrated services. The definition of the strategy has been influenced by the issues discussed
above. In particular one of the objective has been trying to derive test suites strongly focused on service integration issues. Moreover we try to reduce test suite dimension in order to reduce the time spent in testing activities when the derived tests have to be performed on-line on running services.

The presented research profit from the availability of service models suitable for formal manipulation and addresses the automatic derivation of test cases for services to be integrated within a service *orchestration*. This is an approach to the composition of different services that foresees the introduction of a special service (the orchestrator) that coordinates, in a centralized way, the work of all the others.

Our proposal here is to derive test cases from the orchestration definition and from the specification of the expected behavior for the candidate orchestrated services. From these specifications, and applying formal techniques, we derive a test suite to be used to assess the compliance of a participant implementation with the one expected by the orchestration. The test suite can be considered an integration test suite since it is produced considering the whole orchestration and all participants during the test case derivation process. Therefore applying our technique it is possible to trace implicit interactions among the orchestrated services that are hidden within the orchestrator code.

In the next section we provide a general overview of the test derivation strategy, while for some background material we should refer to the literature for lack of space. Section 3. discuss an example and overview the BPT tool. Section 4. discusses related works and finally in Section 5. we draw some conclusions and opportunities for future work.

2. The opt Testing Strategy

The Orchestration Participant Testing strategy (opt) provides a general approach to derive test cases for services to be composed within an orchestration. As a result, derived test cases aim at assessing if a service can correctly behave when integrated within an orchestration. Therefore, a positive result to a test execution does not necessarily lead to the identification of a bug in the integrated services, instead it just shows that the service cannot be integrated in the specified orchestration.

In order to derive meaningful test cases the opt strategy foresees, as first step, the transformation of the orchestration specification in a formalism suitable for being processed by a model checking algorithm. Successively running the model checking and specifying relevant integration properties the opt strategy intends to derive counter-examples representing orchestration executions considered relevant for integration purpose. Finally, the derived counter examples are manipulated in order to isolate, from each trace, the interactions with each involved service. The result of this projection step, applied on all the derived counter-examples, will be a set of test suites for each service involved within an orchestration.

In the following of this section, through the usage of a running example, we detail the various phases of the opt strategy. The strategy is obviously technologically agnostic. To be really applicable it is necessary to instantiate it referring to real orchestration languages, real model checkers and real test harnesses.
2.1 Running Example

In the following we describe a simple orchestration example we will use to illustrate the opt strategy.

Let us assume that we would like to orchestrate mathematical services in order to calculate the following function:

\[
f(x, y) = \begin{cases} 
  x + y, & \text{if } \text{rnd}(0, 30) \geq 15 \\
  (x \mod 14)! + (y \mod 14)!, & \text{else}
\end{cases}
\]  

where: \( x, y \in \mathbb{Z}, \mod \) is the modulo operation, and the \( \text{rnd}(a, b) \) function returns a random integer within the specified range \([a, b]\).

Specifically, we assume to have access to the following mathematical services: (i) the AddService that takes two integers as input and it returns their sum as an integer; (ii) the RndIntService that takes two integers \( a, b \) as input and it returns a random integer within the range \([a, b]\); (iii) the ModService that takes two integers as input \( a, n \) and it returns the modulo on the division of \( a \) by \( n \) as an integer; (iv) the FactService that takes an integer as an input and it returns the factorial as an integer.

Within this simple example we would also consider those scenario where the communication between the orchestration and any of the orchestrated services has to comply with an interaction protocol. Thus, we also describe the interface exported by FactService we want to integrate. In particular it provides: a main method (\texttt{fact}) implementing the factorial, and a configuration method (\texttt{setParameter}) used to set the input parameter for the \texttt{fact} operation. In this case, the proper interaction protocol with FactService is composed by two steps: invoke the operation \texttt{setParameter}, calculate the factorial of the last configured input parameter invoking the \texttt{fact} operation. As an example, Figure 1 depicts an automaton describing the interaction protocol of the service FactService which we are referring to.

In conclusion, the orchestrated process we intend to implement takes two input parameters and returns an integer. Reflecting the arithmetical properties of the involved operations, the orchestrated process can be implemented invoking some services in sequence and other in parallel.

2.2 Counter-examples Analysis of Orchestrations

Among all the possible executions of an orchestration the objective of opt is to identify and select those executions considered more “tricky” with respect to integration issues. From such executions opt will derive a set of test cases for each service participating in the orchestration. Considering the scenario described in Section 2.1 opt generates the various test suites for possible implementations of the services AddService, RndIntService, ModService, and FactService.

In very general terms a service orchestration can be generally reconducted to a graph model that is suitable for automatic manipulation. Moreover, the testers must specify
reachability properties on the derived model. The definition of such reachability properties implements the formalization of the criteria that the testers guess as “useful” for integration testing purpose, and that will drive the test case derivation given the specific orchestration. Thus, model checkers will process the transition system based representation of the service orchestration verifying such reachability properties and looking for counter-examples invalidating them. Intuitively interesting reachability properties, for integration testing purpose, could be those leading to the identification of traces in the orchestration involving many different services.

Indeed, the trick consists in formulating propositions on trap properties [3] that actually admit as counter-examples some configurations of the parameters in the service orchestration. In opt the definition of the trap properties to use is currently left to testers experience and knowledge of the specific orchestration. Nevertheless further study could lead to the identification of patterns to be used independently from the specific context.

For instance, a simple trap property (for simple orchestration) could be: “it does not exist an orchestration execution that, starting from the initial node, reaches the final node and includes two invocations of the fact operation of the FactService.” This property can be expressed for example in a LTL formula as $G \neg (\text{fact} \land F(\text{fact} \land F(\text{END})))$, where we assume that the proposition fact assumes the true value in case the corresponding method has been invoked last, while END represents the termination of the process.

Given an orchestration and a trap property the execution of the model checker will return, in case they exist, a set of possible traces for which the trap properties results to be invalidated. In other words the counter-example will correspond to execution in which the method fact has been invoked twice before exiting the orchestration.

The counter-examples returned by the model checker can be described by a tree structure representing the different paths that, starting from the initial state, can lead to a state violating the property. We can represent the returned counter-examples using an LTS model where: each state in the LTS represents an interaction between the orchestration and an orchestrated service, each transition whose label is prefixed with “?” models an invocation from the orchestration to an orchestrated service; each transition whose label is prefixed with “!” models a reply from an orchestrated service to the orchestration. In addition, all the transitions can be labeled with the parameters exchanged during the interaction (i.e. the name of the service operation and either the actual parameters, or the return values). Figure 2 shows a trace for the running example described in Section 2.1 when $x = 40$, and $y = 100$ and using the trap property in (2.2).

It is worth mentioning that the opt strategy, in order to derive a closed model to which model checking can be applied, requires to include in the described process the specification of the behavior for the composed services. Two different scenarios in modeling third-party services can be prefigured. The first scenario foresees that the specification of the behavior of the composed services is directly defined and modeled by the designer of the service orchestration. In this case, the orchestrator specifies which are the behaviors that, according to its expectations, remote services should provide in order to be bound to the orchestration. The second scenario foresees that the models are directly specified by the provider of the service to be composed. The two approaches have pro and cons. In the first case test cases are more general and useful for being used to test any service. Nevertheless it puts on the shoulders of the orchestrator developer.
the task of defining the models. In the second case the models are available but when a different service needs to be integrated the whole test case generation process has to be restarted. Nevertheless in this paper we do not make any assumption on the development process used to derive an orchestration specification and the approach can be equally applied in both scenarios.

2.3 Test Cases Derivation and Execution

Once defined the criteria driving the generation of traces over the service orchestration, testers should form the different test suites for each service participating in the orchestration. Specifically this process is composed of two steps: the reduction of the tree of traces, and the projection of a trace on a service test case.

As described in the previous section, testers formulate propositions on trap properties guessing to select traces that are meaningful for integration testing. However, in real-case applications of the opt strategy not all the traces produced by a model checker may have the same relevance. With respect to the first step, the reduction of the tree of traces, testers may introduce techniques that float traces considered more relevant. Otherwise, they may decide to prune the counter-example tree, adopting trap properties that better fit to the goal of the test session. For example, with respect to integration testing, testers may consider useful only those traces containing interactions containing a given sequence of invocations possibly involving many different services.

The second step, projection of a trace on a service test case, consists in extracting from the counter-example tree a test suite for each service in the orchestration. In particular, visiting the trace tree $T$, we define $T_S$ as the test suite for the service $S$. Here only the states modeling the interactions between the orchestration and the service $S$ are considered.

The last step in the opt strategy is the actual execution of the test cases derived from the orchestration over the specific service implementations. For each service under test, the testers will refer to the trace projections as test cases and oracles for the testing session.

3. Execution of The Running Example

This section describes how a tester can use the opt strategy to generate test cases from a service orchestration. This section presents a scenario where the test cases are obtained as output of the BPT tool, our tool that implement the opt strategy starting from BPEL specification and WSDL description of the participants. BPT uses model
checking and counter-example based techniques in order to select those executions that could better highlight integration issues. Each execution trace derived from the BPEL specification analysis, is converted in test cases that can be used to assess the behavior of services participating to the orchestration.

**BPT** is based on a set of three basic components, each one carrying on a different activity. Specifically **BPT** is composed integrating: (i) The BPEL2JPF component, (ii) The Model Checker component (based on Java PathFinder - JPF), (iii) The Test Suites Generator component. Figure 3 depicts the workflow among the components of the framework.

The BPEL2JPF takes as input a BPEL specification, and for each service in the orchestration both the specification of the service interface (specified using the Web Service Description Language - WSDL [4]), and a skeleton implementation of the same service. BPEL2JPF translates such specifications in a model suitable for being processed by the model checker (i.e. JPFModels). Specifically, **BPT** integrates Java PathFinder (JPF)(http://javapathfinder.sourceforge.net/) as the model checker to be used in order to derive counter-example traces.

JPF is a system for verifying programs written using the Java language and works exploiting executable bytecode. JPF can be extended implementing properties to check, listeners to observe the execution, and specific objects, called choice generators, to drive the backtracking functionality during the model exploration.

Note that for each participating service, BPEL2JPF generates only a skeleton representation of the service and interconnects it to the JPF model of the orchestrator. The modeling of the expected behavior of a participating service should be defined either by the orchestrator or by the tester that has to complete the Java code generated by BPEL2JPF.

Let us consider the case where a service orchestrator defined the service \( f(x, y) \) by means of a BPEL orchestration. The tester specify also the trap properties for JPF and has to provide an initial configuration for the input data-set of the BPEL process. It is important to remark that the tester should mitigate the risk that any of the “interesting” branch of the orchestration is not covered due to the selection of the input data-set. In this simple example we can assume that the tester generated a random data-set uniform distributed on the values \( x, y \) over \( \mathbb{Z} \).

Thus, from the BPEL specification, from both the WSDLs and the models of the expected behavior of the orchestrated services, and from the input data-set, **BPT** returns the set of execution traces resulting as a projection of the counter examples selected by the model checker with respect to the trap property. In this specific example, the tester can use the test cases generated by means of **BPT** validating the integrability of two different implementation of the **FactService**: namely \( \text{fact}_a \) and \( \text{fact}_b \). In most of the cases, the two different implementations of the **FactService** may supports different range of input parameters throwing an overflow exception depending on different thresholds.
Specifically, let’s imagine that: \texttt{fact}_a can handle input within $[0..20]$, while \texttt{fact}_b can handles integers within $[0..10]$.

The unit test process reveals that both the service implementations can run only under some pre-conditions. In this scenario, the unit test of the services would reveal that both the implementations actually behave as expected by the \texttt{FactService}, but over a revealed threshold, the two services cannot be used for implementing \texttt{FactService}. Analyzing if the pre-conditions of the service implementation are compliant with the service orchestration is only due to the tester. In other words, for each service implementation the tester analyzes the orchestration looking for integration issues.

Let us consider that the test cases generated by BPT include the test case derived for \texttt{FactService} from Figure 2. Thus, the tester will invoke \texttt{fact}_a passing it $in_{fact} = 12$ as parameter. As $in_{fact} \leq 20$, and assuming that \texttt{fact}_a is correctly implemented, the service will reply in compliance with its behavioral model (e.g. $4,790016e8$). Similarly, the tester will invoke \texttt{fact}_b. However, in this case, the service implementation throws an exception (because of $in_{fact} \geq 10$). In this case, the tester reveals an inconsistency between \texttt{fact}_b and the abstract behavior expected by the \texttt{FactService}.

Indeed, the test session showed that with respect to the service \texttt{FactService}, the implementation \texttt{fact}_a can be integrated within the orchestration described in Section 2.1, while \texttt{fact}_b cannot. Now, let us consider that the tester would validate the integrability of another implementation of the \texttt{FactService}: \texttt{fact}_c. In this case, we image that interaction protocol implemented by \texttt{fact}_c is unknown to the tester and it is also slightly different from the protocol foreseen by the service orchestrator (see Figure 1). In particular, we imagine that \texttt{fact}_c has to be reset in order to change the configuration of the input parameter (see Figure 4). Clearly, in this scenario, the \texttt{FactService} interface required by the orchestration is a subset of the interface exported by this service implementation. Specifically, \texttt{fact}_c also exports an operation that resets the status of the service.

As the orchestration used in this running example includes a sequence of invocations to the \texttt{FactService}, the test cases generated by BPT revealed that the \texttt{fact}_c cannot be integrated within the orchestration. Concluding, by means of BPT testers can derive a test cases for each service within the orchestration. Each test case targets in revealing potential integration issues of the service orchestration. Indeed, the test cases validates if the usage of a service implementation within an orchestration is compliant with the service pre-conditions or with service protocol. Testers can use the test cases generated by BPT as an acceptance validation for the implementation of the orchestrated services.

4. Related Works

Our work has been inspired by several works in the area of counter-example based testing techniques. Nevertheless we do not propose a novel technique in this area, instead we use such techniques in the SOC domain, in which the ample availability of
formal models make it a natural choice. A comprehensive survey on the state of the art for counter-example based testing techniques can be found [5].

With respect to the translation of BPEL specifications to models suitable for being manipulated by a model checker, an interesting work is reported in [6]. In such a case the authors describe a BPEL translation toward a BIR model (input format of the BOGOR model checker). Nevertheless in this case the translation was done for the sake of static verification. Moreover the derived models did not consider data, so they could not be used in our approach.

In [7] the authors check the behavior of service participants instrumenting, within the BPEL process, each invocation to a service with pre- and post-conditions expressed in a XML based constrain language. The approach can complement our proposal, being pre- and post conditions used to derive models for the invoked services.

5. Conclusions and Future Work

The paper presented a test case derivation strategy, called $\sigma_{opt}$, aiming at checking the behavior of services involved within an orchestration. The strategy is based on a counter-example derivation technique and permits to derive test cases considering the whole service composition. The strategy has been implemented in a tool which assumes that the orchestration has been specified using BPEL and participant interfaces using WSDLs documents although the details are not discussed here to a lack of space. Nevertheless, the future work agenda is quite full and does not comprise only the completion of the tool. As first we intend to investigate on practical techniques permitting to reduce the state explosion phenomenon, a first investigation already started using genetic algorithms. Finally we intend to extend the experimentation to real case studies in order to have a real validation of the approach.

References


