A Counter-example Testing Approach for Orchestrated Services

Francesco De Angelis, Andrea Polini
Department of Mathematics and Computer Science
University of Camerino, Italy
{francesco.deangelis,andrea.polini}@unicam.it

Guglielmo De Angelis
ISTI – CNR
Via G. Moruzzi, 1
56124 Pisa, Italy
guglielmo.deangelis@isti.cnr.it

Abstract—Service oriented computing is based on a typical combination of features such as very late binding, run-time integration of software elements owned and managed by third parties, run-time changes. These characteristics generally make difficult both static and dynamic verification capabilities of service-centric systems. In this domain verification and testing research 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 an approach to automatic test case generation aiming in particular at checking the behaviour of services participating in a given orchestration. The approach exploits the availability of a runnable model (the BPEL specification) and uses model checking techniques to derive test cases suitable to detect possible integration problems. The approach has been implemented in a plug-in for the Eclipse platform already released for public usage. In this way BPEL developers can easily derive, using a single environment, test suites for each participant service they would like to compose.

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

I. 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. Nevertheless, from a Software Engineering perspective, the new paradigm seems to pose new challenges and the activities of service design and service composition do not appear so straightforward [1]. Among the various software engineering activities testing certainly asks for new techniques and further investigations.

A recent European Community document declared that “the area of testing services and their specifications has until now received limited attention from the research community …” [2]. Indeed looking at the technical programs of the main software engineering and service computing conferences, held after 2005, really few papers have been published on service testing. We think that this is probably due to the inherent difficulties posed by the SOC paradigm rather than to the fact that issues on service testing are not perceived by the research community.

In very general term, testing is a Software Engineering activity which aims at discovering faults in a piece of software, and to indirectly raise the confidence of a piece of software in case no faults are discovered by a test session. This objective is pursued performing a finite set of experiments and checking if the observed effects are in line with those expected. Testing power is strongly interrelated with software qualities such as controllability and observability. The first quality refers here to the possibility of manipulating a software object, both to set specific internal parameters, and to stimulate it with selected values. The second quality refers to the capability of a tester object of observing the effects of the manipulation phase.

Trying to understand how testing activities could change in a given domain we need to understand how, and how much controllability and observability are affected. Indeed the SOC domain seems to have specific characteristics that tend to reduce both qualities. 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.

Indeed run-time integration testing of services possibly owned by different organizations still presents unsolved issues. In [3], Canfora et al. discuss how aspects such as, lack of observability of the service code and structure, dynamicity and adaptiveness of SOC systems, lack of control on third party services, and lack of trust between service providers, limit the testability of service-centric systems. In our opinion a lot of research is still needed in this domain. The research for solutions could follow many different paths possibly involving different research communities [4], [3].

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 both 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 the possible 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 orchestration code.

For the sake of comprehension, in the next section we provide some introductory material on service orchestration, testing and model checking. Then in Section III we provide a general overview of the test derivation strategy, while in Section IV we detail the design and implementation of the BPT tool which provide a real instrument to apply the test derivation strategy. Section VI discusses related works and finally in Section VII we draw some conclusions and opportunities for future work.

II. BACKGROUND

A. Service Orchestration

The specification of a service orchestration encompasses the definition of roles and functionality to be aggregated in order to provide a more complex functionality to external users [1]. The orchestration model always foresees that one of the parties involved in the process (the orchestrator) takes the control over all the business-process interactions. The orchestrator is responsible for defining both the order in which services are invoked, and the conditions under which a certain service may or may not be integrated within an orchestration execution. Moreover the orchestrator is responsible for adopting strategies permitting to assess the behavior of composed services in order to not integrate misbehaving services. Considering a more architectural related perspective, the orchestration can be considered as the glue code that permits to coordinate different components to derive a more complex functionality. The term orchestration obviously derives from the music domain, where the concurrent performance of many musicians is composed by the orchestra director who establishes when and how each musician (or group of them) must play.

On the technological side one of the most significant proposals permitting the specification and execution of service orchestrations is currently represented by the Business Process Execution Language (BPEL) [5].

BPEL defines a model for describing the behavior of a business process (the orchestrator) based on the expected interactions between the process itself and the composed partners. BPEL assumes that all the interactions with each partner occur through Web Services operations. In particular, BPEL distinguishes its instructions in basic activities and structured activities. A basic activity is an instruction that interacts with something external to the process itself (i.e. for receiving, replying, or invoking services external to the business process). Structured activities includes all those statements meant for controlling the process flow (e.g. loops, branching, etc.).

Once the BPEL process has been defined, it is possible to execute it using one of the available BPEL engine, being ActiveBPEL1 probably the most prominent example. A BPEL engine is the implementation of an interpreter for the BPEL formalism. The execution of the BPEL process will result in invocations to the external services composed in the orchestration. In this scenario our objective is to automatically derive from the BPEL specification a set of test cases that will permit to assess the behavior of services to be possibly integrated within the orchestration.

B. Testing and Model Checking

Model checking is a technique to statically prove or disprove that a model satisfy a property. In the general case it takes as input an operational model of the system, such as an automaton, and a temporal property to be verified on the model. The idea of the techniques is to use algorithms able to logically explore the state space of a Kripke structure in which each state is labeled with the set of atomic proposition that hold in that state. In case a state, in which a violation of the property is detected, is reached during the exploration, the technique reports to the user the whole execution trace, called counter-example, that, starting from the initial state, can bring to the violating state. Otherwise, in case no violation is reported, the model is guaranteed to satisfy the property.

Model checking techniques were firstly proposed by different researchers in the early 80’s and they have received a lot of attention in the last years. Many different model checkers are today available, each one based on the use of different exploration strategies or supported temporal logic formalisms, such as LTL, CTL, Hennessy–Milner Logic, and many others.

As model checking applies to models, it is not, in general, a sufficient means to prove that a real implementation of the

1http://www.activebpel.org/
system is correct. Real systems are much bigger than the models they derive and the refinement process can introduce faults. In general, it is generally necessary to complement static techniques with dynamic verification techniques, such as testing, so aiming at checking the correctness of the real system implementation on a set of sample executions, carefully selected.

Indeed one of the most complex issue for testing is certainly the selection of the samples to use (i.e. test cases) to derive enough confidence on system behavior. Often the test case selection activity is carried on by manual means and typically it requires high volume of budget. For such reason many different techniques have been proposed trying to automate the derivation step. Among the others, a very interesting technique suggests the generation of test cases from formal models using model checking.

Specifically, the very general idea of using model checking for testing purpose assumes that the tester specifies the characteristics that a test case should have through a test purpose. Usually, having an operational model of the system under test, such characteristics are expressed by means of a formula using temporal logic. Negating the defined formula the tester obtains what is called a trap property saying that the property does not hold on the model. A trap property is then passed to the model checker. In case an execution trace satisfying the formula exists, the model checker will return it to the tester as a counter-example, since the trace violates the trap property. At this point the tester will only need to make concrete the counter-example in a test case for the real system.

III. THE OPT 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, the 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 fault in the integrated services, instead it just shows that the service should not be integrated in the considered 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. Successfully running the model checking and specifying relevant integration properties (in negative form) 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 simple 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 checker and real test harness.

Nevertheless, when real technologies and real case studies are considered, any counter-example based technique can generally suffer of the state explosion phenomenon. This is a complex issue and many different approaches have been proposed to mitigate the phenomenon, such as bounded model checking, symbolic algorithms, abstraction and other. The decision on which approach to use is out of scope for the general strategy (i.e. opt) and can depend from low level technical details. Indeed we postpone such considerations to next section, where we discuss the choices we took in the BPT tool to handle the problem.

A. Running Example

In the following we describe a simple orchestration example which is used to illustrate the opt strategy. The example is loosely based on the one described in [6].

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

$$f(x, y) = \begin{cases} 
  x + y & \text{if } \text{rnd}(0, 30) \leq 15 \\
  (x \% 14)! + (y \% 14)! & \text{else} 
\end{cases}$$

(1)

where: $x, y \in \mathbb{Z}$, $\%$ 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:

- The AddService that takes two integers as input and it returns their sum as an integer.
- The RndIntService that takes two integers $a, b$ as input and it returns a random integer within the range $[a, b]$.
- 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.
- 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 a given interaction protocol. Thus, we assume to have a description of the interface exported by FactService we want to integrate. In particular such an interface provides: a main method (fact) implementing the factorial, and a configuration method (setParameter) used to set the input parameter for the fact operation. In this case, the proper interaction protocol with FactService is composed by two steps: invoke the operation setParameter, calculate the factorial of the last configured input parameter invoking
the fact operation. As an example, Figure 1 depicts the 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 others in parallel. In particular Figure 2 depicts the orchestrated process implementing the function (1) we used as an explicative scenario.

**B. 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 III-A, and the orchestration depicted in Figure 2, opt permits to generate the various test suites for possible implementations of the services AddService, RndIntService, ModService, and FactService.

In very general term a service orchestration can be reconducted to a graph model (such as an Extended Finite State Machine) where each computational statement in the service orchestration can be represented by a statement within a node, and each invocation or reception of a message from a composed service will be represented by an edge. Depending on the tools to be used for the next steps, and on the kind of verification to be applied, different representations will be derived. In the following, in order to describe the opt strategy, we will generally refer to a generic graph representation of an orchestration.

After having reconducted the orchestration specification to a model suitable for automatic manipulation the following step asks to testers to 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 step 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.

Applying opt, testers will formulate test purposes in form of reachability properties. The negation of such specifications automatically lead to the definition of trap properties [7]. Model checking algorithms use the trap properties in order to identify traces actually satisfying the specified test purpose. Specifically, when such traces exist, they are obtained as counter-example to the derived trap property.

In opt the definition of appropriate test purposes is currently left to testers experience and knowledge of the specific orchestration. While it would be interesting to explore the possibility of automatically deriving useful test purposes, just on the base of the orchestration structure, this possibility has not been investigated in depth yet.

As an example, a simple trap property (for a simple orchestration) generated with the objective of having two invocations of the fact operation in the trace, could be:

“it does not exist an orchestration execution that, starting from the initial node, reaches the final node of the automaton and includes two invocations of the fact operation of the FactService.”

this property can be expressed in a LTL formula as $G\neg(fact \land F(fact \land F(END)))$, 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 reconducted to a tree structure description 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 3 shows a trace for the running example described in Section III-A when $x = 40$, and $y = 100$ and using the trap property in (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.

C. Test Cases Derivation and Execution

Once the criteria driving the generation of traces over the service orchestration, have been defined, 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 trees composed from traces, and the projection of a trace on a service test case.

As described in the previous section, testers express test purposes with the aim to select traces useful 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 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.

```

Listing 1. Test Suite Projection

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. Listings 1 describes in pseudo-code the algorithm for the test suite projection when \( T \) is visited using the depth-first search.

Figure 4 depicts the test suite obtained from the projection of the trace depicted in Figure 3 on the service FactService.

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.

IV. THE BPT TOOL

In this section we describe the architecture and the usage of the BPEL Participant Testing (BPT) tool. BPT provides a real implementation of the strategy sketched in the previous section. Specifically, the tool generates test suites for service participants assuming that the orchestration specification is expressed using the BPEL language [5].

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 in the orchestration.

BPT is based on a set of three basic components, each one carrying on a different activity. Specifically BPT is composed integrating:

- The BPEL2JPF component
- The Model Checker component (based on Java PathFinder - JPF)
- The Test Suites Generator component

Figure 5 depicts the workflow among the components of the framework.

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 [8]), 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)\(^3\) 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.

In our opinion this aspect should not be considered as a major obstacle, since the orchestrator generally makes assumptions on the values that could be returned from the remote services. So we mainly ask to codify such an assumption. What the orchestrator in generally should ignore is the business logic used to calculate the returning values. If currently we ask to use the Java programming language to encode the knowledge on participant services, more abstract models could be used. Obviously in such a case a transformation to code suitable for JPF manipulation need to be provided. Services participating in an orchestration are typically represented using models internally containing non-deterministic choices. In JPF this can be encoded using data types for which an enumeration of possible values is defined. Thus in case the orchestration need to integrate a simple hotel booking service that can reply “yes” or “no” the service will be represented through a simple non-deterministic service that uses a simple data structure, which can assume both values. JPF will use the data structure, during the exploration of the resulting model, assuming that the answer can be both “yes” and “no”. Data returned by the service, as result of a non-deterministic choice, will result in different paths to be explored, and than in the observation of different traces within the BPEL code.

In addition to the JPFModels generated by the BPEL2JPF component, JPF also gets as input a set of test purposes. JPF uses such test purposes in order to produce trap properties used to identify those execution traces that can be shown as counter-examples. Only the counter-examples would be returned by JPF in order to proceed with their projection over each participant, and then for the generation of the test.

\(^2\)BPT is available at http://bptesting.sourceforge.net/

\(^3\)http://javapathfinder.sourceforge.net/
suite. In BPT, we defined a very simple language to specify test purposes. In particular we assume that the test purpose which result interesting for the tester are those expressing the possible structure of a sequence of method invocations.

In the current implementation, in order to improve usability for people non acquainted with temporal logic formalisms, BPT provides a set of configurable patterns for test purposes that permit to further specify the structure of a successful execution of the orchestrator. JPF loads the test purposes in form of rules from a configuration file. In the following we describe the notation used for each pattern:

- the “>” symbol is used to express a strict sequentiality among method invocations. “a > b” point to the invocation of the method a and then the following invocation of b. This is equivalent to build the LTL trap property “G¬(a ∧ Xb)”.
- the “::” symbol to express sequentiality among method invocations. “a :: b” point to the invocation of the method a and then the invocation of b in the same path. This is equivalent to build the LTL trap property “G¬(a ∧ Fb)”.
- the “!” symbol to express the absence of a method invocation in a path. “!a” indicates the lack of the invocation of a in the considered sequence. This is equivalent to build the LTL trap property “Fa”.

As an example, the “a :: b > c !d” rule can be used to discriminate the paths in which there are the invocations of a, then the invocation of b and c in a strict sequence, and then the absence of the invocation of the method d. The BPT tool generates the related trap property in a transparent way for the user to generate the counterexamples using the model checker.

Moreover, in order to mitigate the risk of the state explosion phenomenon, that given the presence of ample data sets is always present, BPT requires that the parameters defined in the BPEL process interface are set by the tester to smaller data set before running the model checker. So, for instance, in case of an integer parameter the tester can decide that the value to consider are in the interval [0,100]. We are currently investigating on an automatic strategy that could be useful to mitigate the state explosion phenomenon and that reduce the needs for human intervention, nevertheless the current implementation requires human intervention to reduce such a risk. explosion.

The tool generates test suites starting from an integration scenario given by the orchestration. Using BPT the tester should consider as “good” test suites those deriving from traces containing interactions with several different services. This can be specified providing properties declaring the non existence of those execution paths. When the counterexamples traces have been returned the Test Suites Generator is used to derive and store a test suite for each participant. In particular each test case within a participant test suite results from the projection over the traces identified in the previous phase considering only those messages associated to the specific participant. The result is a test suite for each participant that can be used in order to test if a service implementation can safely be included in the orchestration or not.

A. Model Generation - The MDE approach

The BPEL2JPF component has been implemented according to the principles of model-driven engineering (MDE).

In particular, the implementation of the BPEL2JPF has been developed considering the Eclipse platform in the context of the Eclipse Modeling Framework (EMF)\(^6\).

EMF provides tools and run-time support to produce a set of Java classes for models, or supports the implementation of model manipulators such as model-to-model and model-to-code transformation. In the current implementation we adopted openArchitecturWare (oAW)\(^3\) that is a templates-based model-to-code generator. Figure 6 depicts the transformation process we defined within the BPEL2JPF.

BPEL2JPF permits to derive Java programs containing statements and directives for the JPF model checker. A detailed description of JPF is given in Section IV-B.

BPEL2JPF takes as input the specification of a BPEL orchestration. The first step in the transformation chain is to represent it as an EMF model. For such models the Eclipse platform provides parsing functionality that can be combined with specific code generation tools. The automatic processing is possible since the tool links to the metamodels for BPEL and WSDL (i.e. the BPEL and the WSDL schemas).

The generator process the EMF representation of the BPEL orchestration by means of an oAW component called Outlet. Such component uses target-model templates implemented according with an oAW-specific language called Xpand. Specifically, Xpand allows the manipulation of the models conforming to a metamodel expressed in EMF.

The generation of the models for the JPF of a BPEL orchestration provides translation for almost all the BPEL constructs but the flow operator. The template in Xpand is available for download from the web site of the BPEL Participant Testing Project.

Furthermore, we also integrated the capabilities of the BPEL2JPF within the BPEL Project Eclipse plug-in\(^6\) Specif-

\(^4\)http://www.eclipse.org/modeling/emf/
\(^5\)http://www.openarchitectureware.org/
\(^6\)http://www.eclipse.org/bpel/
ically, we integrated the model generator within an environment supporting the graphical specification (e.g. the drawing) of BPEL orchestrations.

B. Model design - BPEL to JPF Models

As described in the previous section, the target model we generate from the BPEL process is a set of Java programs modeling the source orchestration and integrating the functionality of JPF. In particular, the artifacts that BPEL2JPF generates are:

- a Java model for each orchestrated services. Specifically, each orchestrated service is mapped to a Java skeleton. Indeed, Web Services invocations are modeled as method calls.
- a Java class representing the BPEL process. The BPEL variables and the activities are replicated with their counterparts in Java using attributes, conditional statements, repetition statements, and method invocations.
- a data representation for the input data set of the BPEL orchestration.
- a Java main for the overall system that runs JPF on the generated model.
- Several utility classes used to instrument the model to allow the tracing of the system exploration.

Let us consider the invocation to the mod operation of the ModService in the Figure 2.

Listing 2 shows the BPEL fragment that includes the invoke activity and the declaration of the variables it uses. In the code, the variable ModIn is typed as a complex-type named mod that is composed of two integer field: a and n. Similarly, the variable ModOut has type modResponse, that is a complex-type composed by the integer field return. Among the other attributes, the invoke activity includes the name of the remote operation to invoke, it declares ModIn as input variable, and ModOut as output variable.

As reported in Listing 3, BPEL2JPF generates a Java class for each variable (i.e. ModInClass, and ModOutClass). Each class is instantiated into a Java object having the same name of the BPEL variable. As described in the previous section, for each participant service is generated a Java skeleton suitable for the interconnection with JPF. Such generation is done by means of the WSDLs. Thus, the remote operation used in the BPEL invoke statement is mapped to a method of the Java skeleton having a similar interface (i.e. name and parameters). Finally, each Java object is than used either as input parameter to the skeleton’s method call or to store the corresponding return value.

For the sake of completeness it is important to note that the mapping has not been fully performed yet. In particular constructs such as pick and wait still miss the definition of a mapping.

V. Execution of the Running Example

This section describes how a tester can use BPT in order to generate test cases from a service orchestration. The section presents a scenario where the test cases that are obtained as output of the BPT tool can be actually executed over the composed service class implementations.

As described in Section III-A, let us consider the case where a service orchestrator defined the service \( f(x, y) \) by means of a BPEL orchestration and he/she defined the participants skeletons using the Java programming language.

At this point the tester specify the test purposes using the patterns introduced in Section III-B. Specifically, the property (2) is coded in the JPF configuration file specifying the rule “fact: fact” saying that we look for execution in which the method fact in invoked twice. Furthermore, the tester has to provide an initial configuration for the input data-set of the BPEL process. As discussed in the previous section, 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 uniformly 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 JPF component with respect to the test purpose.

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 fact_a and fact_b.

In most of the cases, the two different implementations of the FactService may support different range of input parameters throwing an overflow exception depending on different thresholds. Specifically, let’s imagine that: fact_a can handle input within [0..20], while fact_b can handle 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 behaves as expected by the FactService, but over a revealed threshold, the two services cannot be used for implementing FactService. Analyzing if the pre-conditions of the service implementation are compliant with the expectation of the service orchestration can be done by the tester. In other words, for each service implementation the tester analyzes the orchestration looking for potential integration issues.

Let us consider that the test cases generated by BPT include the test case depicted in Figure 4. Thus, the tester will invoke fact_a passing it \( in_{fact} = 12 \) as parameter. As \( in_{fact} \leq 20 \), and assuming that fact_a is correctly implemented, the service will reply in compliance with its behavioral model (e.g. 4,790016e8). Similarly, the tester will invoke 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 fact_b and the abstract behavior expected by the FactService.

Indeed, the test session showed that with respect to the service FactService, the implementation fact_a can be integrated within the orchestration described in Section III-A, while fact_b cannot.

Now, let us consider that the tester would validate the integrability of another implementation of the FactService: fact_c. In this case, we image that interaction protocol implemented by 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 fact_c has to be reset in order to change the configuration of the input parameter (see Figure 7). Clearly, in this scenario, the FactService interface required by the orchestration is a subset of the interface exported by this service implementation. Specifically, 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 FactService, the test cases generated by BPT revealed that the fact_c cannot be integrated within the orchestration in Figure 2.

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.

VI. RELATED WORKS

Our work has been inspired by several works in the area of counter-example based testing techniques. Starting from the first papers on the subject appeared in the late 90’s [9], [10], the technique has received a lot of attention and it has been applied in many different context (e.g. ASM specifications [11]). 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 in [12].

With respect to the translation of BPEL specifications to models suitable for being manipulated by a model checker, an interesting work is reported in [13]. In such a case the authors describe a BPEL translation toward a BIR model (input format of the BOGOR model checker). A similar approach is described in [14] where the process is translated using a Finite State Process notation. Nevertheless in both cases the translation was done for the sake of static verification. Moreover the derived models did not consider data, so they could not be used for our approach.

An interesting approach to BPEL testing is reported in [15] where an XML-based test framework for BPEL composition is proposed. The use of counter-example based techniques to derive test suites from BPEL processes specifications is described in [16], [17]. Nevertheless in these papers the focus is mainly related to testing the whole process and not to derive test cases for the single participants.

In [18] 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.

An approach to model-based testing of services has been presented in [19]. In such case services are described using
a state machine formalisms and algorithms for checking conformance relations are used to generate the test cases. Nevertheless test cases derived using models of single participant result in test suites that are more focused to unit test than to check services integration.

**VII. CONCLUSIONS AND FUTURE WORK**

This paper presents a test case derivation strategy, called opt, aiming at checking the behavior of services to be possibly included in a service 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 the BPT tool which assumes that the orchestration has been specified using BPEL. BPT transforms the BPEL model into Java code suitable for being verified using Java PathFinder (indeed the translation from BPEL to JPF is one of the contribution of our work). Moreover BPT provides a simple language that permits to express test purposes that are successively encoded into trap properties for being processed by JPF. As a result using BPT the tester is strongly supported in the derivation of test suites to be used to assess the behavior of the services he/she wants to integrate. It is worth mentioning that a proof-of-concept version of BPT is already available as an Eclipse plug-in.

The tool has been already applied to a small case study showing the feasibility of the approach. Nevertheless the future work agenda is quite full. As first we intend to investigate on practical techniques permitting to reduce the state explosion phenomenon, a first investigation already completed. At the same time we intend to enrich the property pattern set giving the possibility to express more complex properties. Finally we intend to extends the experimentation to real case studies in order to have a more comprehensive validation of the approach.

**REFERENCES**


