Towards stochastic inference driven SOA testing
Bayesian Networks for Services Architecture Testing (BN4SAT)

Ariele P. Maesano
Laboratoire d’Informatique de Paris 6 UPMC,
Simple Engineering France
Paris, France
ariele.maesano@lip6.fr
ariele.maesano@simple-eng.com

Fabio De Rosa
Simple Engineering Italia
Roma, Italia
fabio.de-rosa@simple-eng.com

Abstract—This paper reports the progress of a project whose goal is to develop a SOA testing environment in which grey-box and black-box testing strategies for large services architectures are driven by stochastic inference. The testing environment architecture is compliant with the UML Testing Profile and is endowed of an inference engine that chooses the next test case to run. A BN model of the services architecture has been defined. In order to cope with the size and density of the BN for even small services architectures, techniques of model-driven inference by compilation that allows quick generation of arithmetic circuits directly from the services architecture model and the test suite have been developed.

Keywords: SOA, Web services, UDDI, contract-based, model-driven, Bayesian network, BN, troubleshooting, testing, TTCN-3, AC, CNF, UML

I. INTRODUCTION AND BACKGROUND

With the spreading of Internet and Internet-related technologies, a large number of applications, systems and devices will be connected and will collaborate, allowing the automation of business processes that support daily activities. Service orientation and Service Oriented Architecture (SOA) are the breakthroughs that allow organizations to implement distributed architectures of loosely coupled systems in order to achieve flexible, dependable and secure business automation.

In the contract-based, model-driven approach [15], Service Oriented Architecture is a design and implementation style that has three main distinctive properties [15]: (i) the collaboration among participant systems is carried out through the exchange of services regulated by service contracts, (ii) service contracts are formal models of the service function and of the providers and consumers interfaces and external behaviors, including security and quality of service exigencies and constraints, (iii) service contracts do not include any information about system (provider and consumer) internals and implementations. In this conceptual framework, a services architecture is a network of participant systems, connected by service contracts, that collaborate in order to achieve business goals.

Validation of services architectures is a huge and complex task. The challenge is the compliance of the participants systems actual functions, interfaces and behaviors, realized by their implementations, with those specified by the contracts that they underwrite. Systems’ implementations are private to the participants and hidden. Neither formal methods nor white-box testing can be applied to the validation task by the SOA Architect and Integrator, who, by definition, does not have any access to participants’ internals.

Of course, the only purpose of testing is to find errors, and a successful test is a test that fails [1]. Firstly, the objective of SOA testing is troubleshooting, i.e. provoking service failures - transitions from correct to incorrect service. Furthermore, the SOA testing activity should provide information and support to the debugging teams in order to detect errors - participants’ states that make failures happen and to discover faults - participants’ implementation defects that cause errors - and vulnerabilities - participants’ implementation weaknesses that allow injected faults causing errors. In Reference [2], a well-known authoritative paper, authors provide widely accepted definitions for the terms employed in this paragraph.

SOA validation process shall be conducted mainly by planning, designing, implementing, performing and evaluating, within an appropriate testing environment: (i) black-box tests - functional, non functional, security - on the SOA participants (Systems Under Test - SUT’s) and (ii) grey-box tests on the SAUT (Services Architecture Under Test).

From the testing viewpoint, a services architecture is a collection of nodes connected by links conveying messages or remote procedure calls (rpc). Some or all of these links are observable (grey-box stance). Conversely, the node interiors are never observable (black-box stance). A test run takes place when a Tester - a human being or a system - in a specified configuration of the services architecture and of the states of the resources managed by the nodes, submits to a node of the services architecture a specified stimulus, for instance by sending a message, and compares the services architecture actual response - the exchange of messages or rpc's between nodes following the stimulus - with the expected response. A test case, in a specified SAUT, is composed of a collection of specified messages and rpc's (test oracle) arranged in an interaction, and a representation
of the state of the resources for each node (test context). A test run matches (mismatches) if the actual response corresponds to (differs from) the expected one (the oracle). The test match/mismatch is the only information available to SOA Testers.

SOA testing is a complex and labor-intensive task, including several activities that can be classified in three categories: (i) test case production, (ii) test run execution and evaluation, (iii) test campaign dynamic planning [6]. The automation of these activities is the target of several research projects.

Automatic test case generation for SOA testing concerns functional, non-functional and security testing [13] [3] [12] [6]. Test cases are - statically or dynamically - generated from service requirement models (model-driven, requirement-based or contract-based testing) or following ad hoc methods, including random generation.

Automatic test run execution and evaluation is realized by test environments in which the execution and the evaluation of test suites can be programmed and monitored. One of the most interesting approach is Testing and Test Control Notation (TTCN-3) [23], a test specification language and environment, standardized by the European Telecommunications Standard Institute (ETSI). Tests are defined in TTCN-3 at an abstract level - Abstract Test Suite (ATS) - that is independent from the implementation platforms. Some commercial and free TTCN-3 test environments are available.

Whether the test cases are produced by hand or automatically generated, and whatever the automation level of the test environment would be, a SOA testing strategy is efficient if: (i) it provokes the greatest number of diversified service failures with the smallest number of test cases and test runs, and (ii) returns the most useful information to Testers, helping them manage the test scheduling, and to Debuggers, assisting them in detecting errors and discovering faults and vulnerabilities. The final objective of an efficient testing/debugging cycle is to increase the confidence of the stakeholders in the functional and operational dependability and security of the services architecture and of each participant. In services architectures, because of the mandatory grey-box stance, this goal constitutes a severe challenge. The availability of intelligent methods and tools, able to drive a test campaign could really improve service and system engineering and services architectures' dependability and security. To our knowledge, there are not yet applications of probabilistic inference methods to drive SOA testing execution strategies.

This paper reports the progress of a project (Bayesian Networks for Services Architecture Testing - BN4SAT) focused on the application of a stochastic inference approach and technology (Bayesian networks) to SOA testing strategy. The project is conducted in cooperation between the Laboratoire d'Informatique de Paris VI (LIP6 - Université Pierre et Marie Curie - Paris VI - France), the Centre National de Recherche Scientifique (France) and Simple Engineering, a European group specialized in the design and governance of services architectures.

The remainder of this paper is organized as follows. Section 2 presents the related work. Section 3 describes the architecture of the testing environment. Section 4 aims at clarifying SOA grey-box and black-box testing methods. Section 5 presents a Bayesian Network (BN) model for SOA testing. Section 6 describes a Bayesian inference technical architecture (inference by compilation) especially designed for speeding the inference process about SOA testing. Some conclusions and perspectives of future work are drawn in section 7.

II. RELATED WORK

Among the techniques that have been developed to diagnose complex systems, stochastic methods are considered appropriate and useful when: (i) there is no complete knowledge of the system (black-box, grey-box stance) - the diagnostic process is undertaken in presence of uncertainty; (ii) the evidence data domain is to big to be completely analyzed. In relation to these two points, assessing the reliability of a system corresponds to evaluating the probability that the system will satisfy its requirements [5].

In Reference [26] is presented a seminal work in which Bayesian networks are used to support input partitioning test methods, aimed at understanding which kind of stimulus provokes software errors. In fact, starting from a partitioning of the input domain, the Bayesian network inference helps fast discover what partition or combination of partitions provokes failures. This kind of test strategy allows the dynamic choice of the next test case in order to determine quickly the input associated to the faulty behavior of the system.

An alternative Bayesian network approach to diagnose complex system proposes the use of fault trees. Fault trees, a well known diagnostic technique, allow locating the faulty components by means of inquiries. The construction of the fault tree for a complex system proceeds in a top-down fashion, from events to their causes, following the system decomposition, until elements revealing faults of basic components are reached. In Reference [4] the authors use fault trees to define the minimal cut set (minimal sets of components that need to be all defective to cause the system failure). The fault tree is transformed into a Bayesian network, which is able, starting from a failure evidence, to locate the component or set of components with the highest fault probability.

The use of probabilistic inference through belief networks [19] or variants [4], appears to be adequate for: (i) search and identification of faulty participants of a large services architecture and (ii) probabilistic classification of the fault type, when the only available information is the match/mismatch between the observable actual behavior and the expected one.

III. SOA INFERENCE-DRIVEN TESTING ENVIRONMENT

The architecture of the BN4SAT testing environment is compliant with the OMG's UML Testing Profile (UTP) specification [18], which is generally accepted as the abstract architecture of a black-box testing environment. The testing
environment is built of five component types: (i) SUT, (ii) Test Component, (iii) Arbiter, (iv) Scheduler, (v) Engine. Each participant of the services architecture under test (SAUT) is a System Under Test (SUT).

A Test Component has a range of basic testing capabilities. It is able: (i) to transmit oracle compliant messages and rpc calls/replies to a SUT; (ii) to accept messages and rpc calls/replies from a SUT; (iii) to compare the SUT's messages and rpc calls/replies with the test oracle (match/mismatch), (iv) to set a local test verdict (pass, fail, inconclusive, error - see later). These basic capabilities allow more elaborate test components to transparently intercept the service exchange between participants (SUT's) of the SAUT and to check it, and to emulate the test case behavior of a SUT.

The Arbiter receives from the Test Components local test results, with eventually local test verdicts and complementary information, and produces a final test verdict.

The Scheduler drives the execution of the test runs - to start and stop a test run, the Scheduler starts and notifies the involved Test Components.

The Engine, that is the component added by the BN4SAT approach, is notified by the Arbiter with test verdicts, and manages the Scheduler by handling the test run sequence on the basis of probabilistic inference from the test verdicts.

The standard values [18] of the test verdict are: (i) pass - the test run matches; (ii) inconclusive - the test run mismatches, but it is impossible to characterize the mismatch as a expression of a service failure or of a test environment error, (iii) fail - the test run mismatches, and the mismatch is manifestly the expression of a service failure; (iv) error - the test run mismatches, and the mismatch is evidently the expression of a test environment error.

The abstract test environment architecture can be declined on a TTCN-3 environment, in which the described components are declared by means of appropriate statements of the TTCN-3 language. Moreover, several authors have already recognized the suitability of TTCN-3 for Web service testing and also introduced the idea of deriving abstract test interfaces (ATS) from a Web services' WSDL description [27] [25] [24].

In the BN4SAT approach, the SAUT Deployment description file is a UDDI V3 Data Structure [17] configuration file and its related WSDL files. Note that the basic UDDI V3 representation has been extended - through the standard UDDI extension mechanism - in order to represent use links between participants that are described as UDDI Business Services. The standard UDDI Data Structures allow representing services provided by Business Entities through Business Services and Binding Templates. Use links enable to describe, for each participant, the services that it uses, the participants that make those services available to it, and the ports where the service uses take place.

A Test campaign is conducted by submitting a test suite to the SAUT. A test suite is made of a collection of end-to-end test cases. Each end-to-end test case is composed of: (i) a end-to-end test case interaction, that is a XML inferset describing the ordered interchange of SOAP messages between SUT's [22]; (ii) the collection of SOAP messages, (iii) the collection of Fact Bases - one for each SUT - representing the resource states. Each SUT of the architecture shall be able to implement a technical service (InstallContext Service) whose main operation is install(factBase), allowing it internalizing the context of the test case from the Fact Base.

IV. SOA FUNCTIONAL TESTING METHODS

An "atomic" service contract is a complete and detailed description of service requirements, i.e. of a function, of an interface and of an external behavior, including security and quality of service exigencies and constraints. Atomic services are combined in more complex compound contracts or composed in services architectures, where they are associated to the use links between participants.

The initial target of our research is SOA functional testing, that is testing the provider functional behavior against the function specification, represented by the functional clauses of the service contract.

The basic black-box functional testing method is the comparison between the message or rpc reply actually issued by the SUT when submitted to an oracle stimulus in a specific test context, and the corresponding elements specified by the test oracle. This method works well for purely informative services, i.e. services whose function is to deliver information from the provider to the user, without any change of the state of the resources managed by the provider. But for a state/transition service, i.e. a service whose function is a transition of the state of the resources managed by the provider that is valuable to the user, the match between the oracle and the actual response does not give enough evidence that the test has passed.

A popular approach to contract-based function definition is Design by Contract™ [16]: a function is defined by a triple constituted by (i) an operation signature (operation name, argument type, result type), (ii) preconditions - conditions upon the state of resources involved in the operation (before its execution) and the operation argument data, and (iii) postconditions - conditions upon the state of the resources involved in the operation after its execution and the operation result data.

The Design by Contract™ approach is agnostic about the function implementation, hence it is particularly well-suited for service function specification. An ideal implementation of the service function should be able to perform: (i) the check of the service function preconditions (provider's resources state and operation argument); (ii) when all the preconditions are satisfied, the execution of the function that brings the provider's involved resources to the specified state and produces the specified result, which both satisfy all the postconditions; (iii) otherwise, when at least one of the precondition is not satisfied, the refusal of the function execution and the absence of alterations of the states of the involved resources.

A positive test case is a test case in which the service function is invoked through a request message or rpc, in a context where its preconditions are satisfied. The test oracle
specifies the operation expected result contained in the response message or in the rpc reply. A negative test case is a test case in which the service function is invoked, through a request message or rpc, in a context where at least one of its preconditions is not satisfied. The test oracle specifies the refusal of the operation execution.

Note that negative and positive test cases challenge two parts of the operation implementation that, in a modular design approach, should be kept separate (we don't make any assumption about implementation modularity): the former tests the code that implements the preconditions check and the latter tests the function implementation. These kinds of test cases are able to highlight only certain types of service failures, while other types remain concealed. For instance, a failure that is not revealed by this kind of test is the result of bad code modularity: the preconditions are satisfied, the execution takes place and the result is as expected, but the involved resources state transition has been badly performed or not performed at all. Other kinds of risky failures are the hidden unwanted side effects: preconditions are satisfied, the function is correctly performed, the result is as expected, postconditions are satisfied too, but the operation execution provokes alterations of resources whose states are not directly modified by the operation and shall stay unchanged. (invariants in the Design by Contract™ terminology).

A method for checking the service operation implementation is based upon the definition and use of operation related basic transparency services. Transparency services are queries retrieving data on the state of the provider resources involved in the evaluation of preconditions and postconditions. For a particular service (myService), the related state-before inquiry service (myService_SB_inquiry) retrieves all the data on the state of the resources that are involved in the evaluation of the myService operation preconditions and the related state-after inquiry service (myService_SA_inquiry) retrieves all the data on the state of the resources that are involved in the evaluation of the myService operation postconditions. Basic service test cases can be combined with the appropriate SB_inquiry and SA_inquiry test cases to build self-checked test cases.

For instance, a positive self-checked test case for myService is an ordered triple composed of: (i) a myService_SB_inquiry test case, whose oracle specifies the question and the answer containing the data representing a state verifying the preconditions of the subsequent myService test case; (ii) the myService test case, whose oracle specifies the request and the response containing the operation result relating the successful execution of the service operation in the test context; (iii) the myService_SA_inquiry test case, whose oracle specifies the question and the answer containing the data representing a state verifying the postconditions of the previously executed myService test case. A self-checked test case that is more effective against unwanted side effects is obtained by adding the retrieval of data about invariants to both the SB_inquiry and the SA_inquiry test cases.

The behavior of an interceptor test component that sits between a service user and a service provider and is able to run self-checked test cases can be concisely presented as follows. When solicited by the user participant with a service request, if the request matches the myService test case oracle, the interceptor runs the corresponding myService_SB_inquiry test case against the provider (otherwise the test verdict is set to fail and the error is located in the user). If the answer mismatches the oracle, the verdict is set to error: it is highly probable that the fact base has not been correctly uploaded and installed within the provider SUT. If the myService_SB_inquiry test matches, the interceptor will run sequentially the corresponding myService test case and the myService_SA_inquiry test case. If both match, the verdict is set to pass. If both mismatch, the verdict is set to fail. If the myService test matches and the myService_SA_inquiry test mismatches, the verdict is set to fail: the fault is situated with high probability in the implementation of the service function. If the myService test mismatches and the myService_SA_inquiry test matches, the verdict is set to fail: the fault is situated with high probability in the implementation of the service interaction. The positive self-checked test case interceptor is easily implemented in TTCN-3 [22].

V. BN MODEL FOR SOA TESTING

Bayesian networks are direct acyclic graphical models that represent stochastic variables and dependencies between them [19]. As a troubleshooting system, the purpose of Bayesian inference is to establish which is the next test to run in order to locate faulty service implementations of the participants, i.e. participants whose defect is supposed to be the cause of the service failure. The task of the engine is the management of an efficient sequence of tests that allows highlighting a maximum of service failures with a minimum number of tests runs [5].

The Bayesian network for SOA testing (BN4SAT) is built directly from the SAUT UDDI Description files, the WSDL files and the Test suite. We will call this collection of elements the Test Campaign Package.

The BN model is made of six stochastic Boolean variable types: (i) Participant, (ii) Port, (iii) ActionType, (iv) Action and (v) Transaction. The probability distribution of some of these variables can be initialized by an expert judgment.

A Participant variable represents the probability distribution of the Boolean state {notFaulty, faulty} of a participant of the architecture. The BN compiler creates as many Participant variables as many UDDI Business Services are declared in the UDDI Description file.

A Port variable represents the probability distribution of the Boolean state {notFaulty, faulty} of an instance of an interface used by a participant. For each participant, the BN compiler creates as many Port variables as many interfaces to the other participants it uses (they are declared in the UDDI description file), and its arcs to the user Participant variables.

An ActionType variable represents the probability distribution of the Boolean state {notFaulty, faulty} of a type of action (e.g. the issue of a request message, of a response message, of a rpc, of a rpc reply) accessible at a participant's interface. The BN compiler creates from the the WSDL files
as many ActionType variables as many actions types are performable by a participant at its used interfaces and, for each variable, the arc to the corresponding Port variable.

An Action variable represents the probability distribution of the Boolean state \{pass, fail\} of an action, as specified in a test case - its value is evidence supplied by the test environment. The BN compiler creates as many Action variables as many actions for each test case declared in the Test suite and, for each variable, the arc to the corresponding ActionType variable. The Action variables that participate to an end-to-end transaction (see later) are ranged in a lattice representing the temporal precedence relationship between them.

A Transaction variable represents the probability distribution of the Boolean state \{pass, fail\} of an end-to-end transaction test case The BN compiler creates as many Transaction variables as many end-to-end test cases declared in the Test suite and, for each variable, the arcs to the Action variables of the end-to-end transaction.

VI. BN COMPIlATION AND INFErrence METHODS

The BN4SAT engine cycle can be concisely described as follows:

(i) the Engine invokes the Scheduler with the indication of the test case to run;
(ii) the Scheduler prepares and launches the test on the Test environment;
(iii) the Test environment runs the test and returns the local test verdict with some complementary information to the Arbiter;
(iv) the Arbiter sets a global verdict and; if the global verdict is error, it raises an exception, otherwise it assigns a probability distribution to the verdict and returns it with complementary information (the evidence) to the Engine;
(v) the Engine puts the evidence in the appropriate Action variable, triggering the BN inference process;
(vi) the result of the inference process is the choice of the next test case to run.

To summarize, for a test campaign, the BN is built once, and the BN inference is performed several times on the same BN structure.

A BN for SOA testing, instance of the model presented in the previous paragraph, can exhibit a very huge size, with thousand of nodes, and can be very dense, with a large number of connections and therefore with large conditional probability tables. The BN inference classical methods, such as lazy propagation [14], variable elimination [10], Shafer-Shenoy [21] attain theirs limitations. Current implementation techniques could be inadequate in terms of response time or even not able to process the amount of data.

For these reasons, an important thread of our research has been concentrated upon techniques of BN compilation, more precisely upon BN inference by compilation. Our first results show that BN inference by compilation is an effective and fast way of managing inference over large and dense BN’s. Inference by compilation is based on the idea that each Bayesian network can be interpreted as a multi-linear function (MLF) and therefore implemented by an arithmetic circuit (AC), the standard model for computing polynomials, which is the result of a computational factorization of the MLF [9].

A MLF over a set of variables A is a sum of terms, where each term is a product of the variables of the set. There are consequently \(n = 2^n\) distinct terms and \(2^n\) distinct MLFs over A. Multi-linear functions can contain an exponential number of terms. The computational factorization is necessary to reduce the amount of calculation.

The MLF contain two kinds of propositional variables: (i) evidence indicators - variables that can be observed (the Action variables in the BN4SAT model); (ii) network parameters - variables that cannot be observed (their probability distribution is calculated by the conditional probability tables of the Bayesian network).

A large number of probabilistic queries can be computed using partial derivatives of the AC [11]. In fact, assigning the proper values to the evidence indicators (the leaves of the circuit), the result to the query \(P(e)\), “e” being the evidence, is obtain by computing the values of each node until the root in a bottom-up fashion. The value of the root is the result of the query. The marginal posterior probabilities can be obtained by computing a downward pass (the real purpose of the compilation). These are probabilities that specific elements may fail in a next test. These probabilities drive the choice of the case that will highlight the failure. The case with the highest probability of failure is requested for the next test. Another important advantage of the inference by compilation is the fact that multiple inferences can be executed on the same AC, i.e. without BN recompilation.

There exist various techniques able to transform BN's into AC's [8]. In Reference [9] the authors report a method for the encoding of the BN in Conjunctive Normal Form (CNF), followed by the compilation of the CNF into the deterministic disjunction negation normal form (d-DNNF), that can be easily transformed into an AC.

Traditional CNF generation methods are multi-step: in the first step a coarse version of the CNF is built, which is optimized to a more concise version in a second step [9].

Our method is inspired by these approaches, but utilizes the BN4SAT model in order to infer the topology and content of the “virtual” BN and instead generates an optimized CNF directly from the Test Campaign Package, skipping the intermediate generation of the Bayesian network and the CNF optimization step. After that, the optimized CNF is compiled into the AC by means of the already mentioned classical technique [9]. Our results show that the technique allows to quickly generate the optimized CNF for architectures of large size and complexity, for which classical methods are not practically exploitable.

VII. CONCLUSION AND FUTURE WORK

The research activity whose progress is reported in this paper is centered on the use of BN inference by compilation as a troubleshooting tool and a test strategy manager of an automated SOA testing environment.

The first phase of the research was focused on:

- the definition of the SOA functional testing problem as requirement-based (contract-based), model-driven
grey-box and black-box testing of services architectures;
- the design of a general architecture for a SOA testing environment that can be piloted by a probabilistic inference engine, and the examination of available frameworks for testing, first of all TTCN-3;
- the specification of the BN model for SOA testing;
- the research of methods and algorithms for quick probabilistic inference on the BN model for SOA testing (inference by compilation).

We consider that the results of this phase are stable and can constitute a basis for further research steps:
- the full integration between the inference engine and an available TTCN-3 compliant testing framework;
- the evaluation of BN4SAT on realistic services architectures; we - both the industrial and the academic partner - are looking for cooperation with SOA development and deployment projects in the industrial and in the research domains, for example in the Healthcare sector;
- the research of new methods for requirement-based, model driven, black-box and grey-box SOA testing;
- the evaluation of the suitability of the developed methods and tools to SOA security and non functional testing;
- the evaluation of alternative models for SOA testing, especially for the opaque (non observable) regions of the services architecture, or when there is only knowledge of dependencies between participants, that can be represented by fault trees;
- the improvement of inference by compilation with new methods and algorithms for: (i) CNF to AC transformation and (ii) model-driven direct AC generation from the Test Campaign Package.

ACKNOWLEDGMENT

Arielle P. Maesano is tenant of a fellowship partly supported (funded) by the Association Nationale de la Recherche et de la Technologie (ANRT - France).

REFERENCES

[17] OASIS, Universal Description Discovery & Integration, V3.0.2 (UDDI).
[23] ETSI, Testing & Test Control Notation (TTCN-3