# 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 contains an inference engine that receives test verdicts and triggers Bayesian inference 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 have been developed. These techniques allow quick generation of arithmetic circuits directly from the services architecture model and the test suite.**

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

## I. INTRODUCTION

With the spreading of Internet and Internet-related technologies, a large number of applications, systems and devices are connected and collaborate, allowing the automation of business processes that support daily activities. *Service Orientation* and *Service Oriented Architecture* (SOA) are the methods that allow organizations to put into operation 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: (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 systems (*participants*), bound by service contracts, that collaborate in order to achieve business goals.

Validation of services architectures has to do with the compliance of the participants' implemented functions, interfaces and behaviors with those specified by the contracts that they underwrite. 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 does not have any access to participants' internals.

The 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 identify *errors* (participants' *states* that make failures happen) and to discover *faults* (participants' *defects* that cause errors) and *vulnerabilities* (participants' *weaknesses* that permit *injected faults* causing errors). In [2], a well-known authoritative paper, authors provide widely accepted definitions for the terms employed in this paragraph.

The goal of a SOA Testing is to let stakeholders increase their confidence in the functional and operational dependability and security of the services architecture and of each participant, at the lowest cost. In large services architectures, this goal constitutes a severe challenge. The availability of intelligent methods and tools able to pilot an efficient test campaign could greatly improve service and system engineering and services architectures' dependability and security.

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 to SOA testing strategy[1]. In particular, the paper presents a gray-box testing model for services architecture, a Bayesian network (BN) model for services architecture testing and a technical architecture that enables quick generation of arithmetic circuits (AC) directly from the defined models (inference by compilation) as well as enabling a fast inference cycle.

The remainder of this paper is organized as follows. Section 2 reports background and terms used in our research, whereas Section 3 presents the state of the art. Section 4 describes the architecture of the testing environment. Section

[1]The project is conducted in cooperation by 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.

5 explains SOA grey-box and black-box testing methods. Section 6 presents a Bayesian Network (BN) model for SOA testing. Section 7 describes the Bayesian inference technical architecture (inference by compilation) we have developed with the aim of accelerating the inference process in SOA testing. Conclusions and perspectives of future work are drawn in Section 8.

## II. BACKGROUND

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

From the SOA testing viewpoint, a SAUT is a collection of *nodes* connected by *channels* conveying *messages*, *remote procedure calls* (*rpc*) and *rpc replies*. Some or all of these channels 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* on a SAUT is composed of the specification of: (i) a partially ordered collection of messages, rpc's and rpc replies (*test case oracle*), (ii) the initial states of the resources for each node (*test case context*). A test case run *matches* (*mismatches*) if the SAUT response, in an environment compliant with the test case context, corresponds to (differs from) the expected one (the test case oracle). The test case run match/mismatch is the only information that can be made available to SOA Testers.

SOA testing activities 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 and industrial projects [13] [3] [12] [6] [23].

Whether the test cases are produced by hand or automatically generated, and regardless of the automation level of the test environment, 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) it 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. To the best of our knowledge, there are not yet applications of probabilistic inference methods to the management of SOA testing execution strategies.

## III. STATE OF THE ART

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 approaches 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. 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 from a Web services' WSDL description [27] [25] [24]. Some commercial and free TTCN-3 test environments are available.

The employment of stochastic methods to diagnose complex systems is 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 the presence of uncertainty; (ii) the evidence data domain is much too large 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].

Reference [26] presents a seminal work relating the use of Bayesian networks to support *input partitioning* test methods, that are 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 to quickly discover which partition or combination of partitions provokes failures. This kind of test strategy enables the dynamic efficient choice of the next test case in order to rapidly determine 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* [4]. 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 [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. Research on probabilistic inference computational complexity proposes several methods that are discussed in section 6 [8] [9] [10] [11] [14] [21].

## IV. SOA INFERENCE-DRIVEN TESTING ENVIRONMENT

The architecture of the BN4SAT testing environment is compliant with the OMG's UML Testing Profile (UTP) specification [18], that 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 capabilities. It is able to: (i) transmit oracle compliant messages and rpc calls/replies to a SUT; (ii) accept messages and rpc calls/replies from a SUT, (iii) compare the SUT's messages and rpc calls/replies with the test oracle (*match*/*mismatch*), (iv) set a local test verdict (*pass*, *inconclusive*, *fail*, *error* - see below). These basic capabilities are combined by more elaborate test components (*Interceptors*) to transparently intercept and to check the exchange between participants (SUT's) of the SAUT and to emulate their behaviors.

The Arbiter receives from the Test Components *local test verdicts* and complementary information, and produces *final test verdicts.*

The Scheduler drives the execution of the test runs - it instantiates, 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, and the match is manifestly the expression of compliant behavior; (ii) *inconclusive* - it is impossible to characterize the match/mismatch as the expression of a correct/incorrect behavior, (iii) *fail* - the test run mismatches, and the mismatch is manifestly the expression of a service failure; (iv) *error* - a test environment error (configuration, run time, ...) is detected. In our approach the final verdict of a test run - different from *error*, which is treated like an exception - delivered by the Arbiter to the Engine, is an *evidence*, i.e. a probability distribution of a Boolean stochastic variable such as: {(*pass*,P),(*fail*,1-P)}.

The abstract test environment architecture can be declined on a TTCN-3 environment, in which the described components, except the BN4SAT Engine, are declared by means of appropriate statements of the TTCN-3 language.

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 infoset 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 enabling the test case execution. Each SUT of the architecture shall be able to implement a technical service (InstallContext Service) whose main operation is `install(factBase)and` internalizes the test case context from the Fact Base.

## V. SOA FUNCTIONAL TESTING METHODS

The main Test Component pattern in a SOA automatic testing architecture is the Interceptor. Its basic behavior, that is able to cope with a request/response between two SUT's in the background of an end-to-end test case interaction involving several SUT's, can be concisely described as follows. The consumer SUT performs a request action (by message sending or rpc) towards the appropriate provider SUT that has previously loaded the test context. The consumer request action is trapped by the Interceptor that compares it with the oracle request. If the test matches, then the Interceptor's verdict is '*consumer passes*', otherwise it is '*consumer fails*'.

The test run either stops or continues, depending on the test run parameters. If it continues, the Interceptor performs the oracle request action towards the provider SUT and waits for the response. If the provider actual response matches the test case response oracle, then the Interceptor's verdict is '*provider passes*', otherwise (mismatch or timeout) it is '*provider fails*'. Again, the test run either stops or continues. Regardless of the provider verdict, since the Interceptor can act as a proper functionning provider, it transmits the response oracle to the consumer without interrupting the exchange sequence. This results in higlighting more than one fault of the SUT during a single test run. The Interceptor is able to transmit both the consumer and the provider test verdicts to the test environment (the Arbiter).

The illustrated request/response test case and the associated Interceptor implement the basic black-box functional testing method: the comparison between the action (message sending, rpc, rpc reply) actually issued by a SUT in a specific test context, and the corresponding elements specified by the test case oracle. This method works well for *purely informative services*, i.e. services whose function is to deliver information from the provider to the consumer, 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 provider SUT actual response and the oracle does not give enough evidence that the service has been delivered in compliance with the service contract.

A popular approach to contract-based service function definition is Design by Contract™ [16]: a service function is a three part function constituted by (i) an *operation signature* (*operation name*, *argument type*, *result type*), (ii) *preconditions* - conditions upon the state of the 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. In a request/response service interaction, the request conveys the operation argument, while the response conveys the operation result. Service operation preconditions and postconditions can be specified, in a UML service design environment, by expressions in Object Constraint Language (OCL).

From the functional viewpoint, a test case is the implementation of a *sample* of the service operation invocation. A *functional positive test case* is a test case in which the service function is invoked in a context where its preconditions are satisfied. The test oracle specifies the operation expected result contained in the response. In a *functional negative test case* the service function is invoked in a context where at least one of its preconditions is *not* satisfied. The test oracle specifies the refusal of the operation execution.

The illustrated request/response test case pattern and the basic Interceptor are not able to highlight *hidden operation failures*: for instance, the preconditions are satisfied, the execution takes place and the result is as expected, but the state transition of the involved resources 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 and postconditions are satisfied, but the operation execution provokes alterations of resources whose states *remain* unchanged. (*invariants* in the Design by Contract™ terminology).

A method for checking more deeply, but without any access to the internals, the service operation implementation is based upon the availability of *transparency services*, that are *queries* retrieving data on the state of the provider resources involved in the evaluation of preconditions and postconditions. For a particular service (e.g. *myService*), the related *state-before inquiry service* (e.g. *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* (e.g. *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 test cases can be combined with the appropriate state before inquiry and state after inquiry test cases to build *self-checked test cases*. Hence a *self-checked test case* for *myService* is an ordered triple composed of a *myService* test case and the correlated *myService_SB_inquiry* and *myService_SA_inquiry* test cases.

A *testable* service provider is able to implement for each service, at least the InstallContext Service, and eventually the related transparency services. Moreover, the transparency services shall at least retrieve preconditions and postcondition data, but can also retrieve data about invariants, letting the related self-checked test cases be effective against hidden unwanted side effects.

The behavior of an augmented Interceptor, able to manage self-checked test cases, can be concisely presented as follows. Its behavior towards the consumer SUT is the same as the basic Interceptor's. The basic behavior towards the provider SUT (performing the oracle request, waiting for the provider response) is encapsulated in between the running of the state before inquiry and the state after inquiry test cases. The test provider verdict is formulated not only on the basis of the service response match/mismatch, but also on the state before and state after answers matches/mismatches. These kinds of Interceptors and test cases are easily implemented in TTCN-3 [22].

## VI. 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. This collection of elements will be further referred to as the Test Campaign Package.

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

An *Entity* variable represents the probability distribution of the Boolean state {*notFaulty*, *faulty*} of a business organization that holds one or more participants of the architecture. The BN compiler creates as many Entity variables as many UDDI Business Entities are declared in the UDDI Description file.

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 and, for each Participant variable, the arc to the owing Entity variable.

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) available at a port. The BN compiler creates from the WSDL files as many ActionType variables as many actions types are performable by a participant 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 in

an end-to-end transaction (see below) are arranged 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.

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 verdicts with some complementary information to the Arbiter;

(iv) the Arbiter sets global verdicts and assigns them probability distributions such as {(*pass*,P),(*fail*,1-P)} that return as *evidences* to the Engine;

(v) the Engine puts the *evidences* in the appropriate Action variables, triggering the BN inference process;

(vi) the result of the inference process is the choice of the next test case (Action) to run.

## VII. BN COMPILATION AND INFERENCE METHODS

A BN for SOA testing as described in the previous section can exhibit a very large size, with thousands of nodes, and can be very dense, with a large number of connections and therefore with large conditional probability tables. The classical BN inference methods, such as *lazy propagation* [14], *variable elimination* [10], Shafer-Shenoy [21] are limited. Current implementation techniques are inadequate in terms of response time and ability to process large amounts 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 quick 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 AC is the standard model for computing polynomials and 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 $n = 2^{|A|}$ distinct terms and $2^n$ distinct MLF's over A. 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 through 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 obtained by computing the values of each node until the roots 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 the 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 (action) 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 [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 and then 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 but 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 quick generation of the optimized CNF's for architectures of large size and complexity.

## VIII. CONCLUSIONS 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 an automated SOA testing environment that can be piloted by a probabilistic inference engine, and the examination of available frameworks for testing automation, such as 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).

The results of this phase are so far 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. The industrial and the academic partners are looking for cooperation with SOA

development and deployment projects in the industrial and in the research domains, for example in the Health Care 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 for situations where there is only knowledge of dependencies between participants. These dependencies 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

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

## REFERENCES

[1] K. Arnout, X. Rousselot, B. Meyer, "Test Wizard: Automatic test case generation based on Design by Contract™," draft report, ETH, June 2003,

[2] A. Avizienis, Jean-Claude Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," IEEE Trans. on dependable and secure computing, Vol. 1, No. 1 January-March 2004.

[3] X. Bai, W. Dong, W. Tsai, Y. Chen, "WSDL-based automatic test case generation for Web services testing," Proc. IEEE Int. Workshop on Service-Oriented System Engineering 2005 (SOSE 2005), IEEE Press.

[4] A. Bobbio, L. Portinale, M. Minichino, E. Ciancamerla, "Improving the analysis of dependable systems by mapping fault trees into bayesian networks," Reliability Engineering and System Safety, 71(3):249–260, 2001

[5] J. S. Breese, D. Heckerman, "Decision-theoretic troubleshooting: A framework for repair and experiment," Proc. of the Twelfth Conference on Uncertainty in Artificial Intelligence, Portland, August 1-3, 1996.

[6] G. Canfora, M. Di Penta, "Service oriented architectures testing: a survey. Software Engineering, Springer-Verlag, Berlin, Heidelberg (2009).

[7] M. Chavira, A. Darwiche, "Encoding CNFs to empower component analysis," Proc. 9th Intl. Conf. on Theory and Application of SatisfiabilityTesting (SAT 2006), Springer 2006.

[8] M. Chavira, A. Darwiche, "Compiling bayesian networks using variable elimination," Proc. 20th Intl. Joint Conf. on Articicial Intelligence (IJCAI 2007), Morgan Kaufman, 2007.

[9] M. Chavira, A. Darwiche, M. Jaeger, "Compiling relational Bayesian networks for exact inference," International Journal of Approximate Reasoning, Volume 4, Issue 1-2, May, 2006.

[10] F. Gagliardi Cozman, "Generalizing variable elimination in Bayesian netwoks," Proc. IBERAMIA/SBIA 2000 Workshop, 2000.

[11] A. Darwiche, A differential approach to inference in Bayesian networks, Journal of the ACM, Vol 50, No. 3, May 2003, pp. 280-205.

[12] S. Hanna, M. Munro, "An Approach for specification-based test case generation for Web services," Proc. 2007 IEEE/ACS International Conference on Computer Systems and Applications (AICCSA 2007), pages 16–23, IEEE Press.

[13] R. Heckel, M. Lohmann, "Towards contract-based testing of Web services," Electronic Notes in Theoretical Computer Science 82 No. 6 (2004).

[14] A.L. Madsen and F.V. Jensen "Lazy propagation in junction trees," Proc. 14th Conf. on Uncertainty in Artificial Intelligence, 1998.

[15] L. Maesano L. and F. De Rosa, "simpleSOAD® 2.0 - Architecture & Governance," Proc. 22nd Intl. Conf. on Software & Systems Engineering and their Applications (ICSSEA 2010) - Paris, Dec 7-9, 2010.

[16] R. Mitchell, J. McKim, and B. Meyer, Design by contract, by example, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (2002).

[17] OASIS, Universal Description Discovery & Integration, V3.0.2 (UDDI) ,

[18] Object Management Group, UML Testing Profile, Version 1.0 (2005).

[19] J. Pearl, Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, Morgan Kaufman (1988).

[20] P.A. Pari Salas, B.K. Aichernig, "Automatic test case generation for OCL: a mutation approach," UNU-IIST Report No. 321, May 2005.

[21] G. Shafer, Probabilistic Expert Systems, Society for Industrial and Applied Mathematics, Philadelphia, (1996).

[22] Simple Engineering: Stochastic Decision Aid for SOA Testing - Interim Report - R2011033101, 2011, unpublished.

[23] ETSI, Testing & Test Control Notation (TTCN-3

[24] T. Vassiliou-Gioles "Testing Web Services with TTCN-3," Testing Experience, June 2008.

[25] W. Werner, J. Grabowski, S. Troschütz, B. Zeiss, "A TTCN-3-based Web Service Test Framework," Proc. Software Engineering (Workshops) 2008. pp.375~382 (2008).

[26] D.A. Wooff, M. Goldstein, F.P.A. Coolen, "Bayesian graphical models for software testing," IEEE Transactions on Software Engineering, 28:510–525 (2002).

[27] P. Xiong, R.L. Probert, B. Stepien, "An Efficient Formal Testing Approach for Web Service with TTCN-3," Proc. 13th International Conference on Software, Telecommunications and Computer Networks (SoftCOM) (2005).