An updated version of this paper has been accepted at the 31st Annual ACM/SIGAPP Symposium on Applied Computing (SAC) - Pisa, Italy - April 4-8, 2016

Service functional testing automation with intelligent scheduling and planning*

Lom Messan Hillah
Univ. Paris Ouest, and Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR7606
4 place Jussieu, F-75005, France
lom-messan.hillah@lip6.fr

Ariele-Paolo Maesano
Simple Engineering France
F-75011, France
ariele.Maesano@simple-eng.com

Libero Maesano
Simple Engineering France
F-75011, France
libero.maesano@simple-eng.com

Fabio De Rosa
Simple Engineering France
F-75011, France
fabio.de-rosa@simple-eng.com

Fabrice Kordon
Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR7606
4 place Jussieu, F-75005, France
fabrice.kordon@lip6.fr

Pierre-Henri Wuillemin
Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR7606
4 place Jussieu, F-75005, France
pierre-henri.wuillemin@lip6.fr

ABSTRACT

This paper presents the intelligent automation of functional test of services (unit testing) and services architectures (end-to-end testing) that has been developed by the MIDAS project and is accessible on the MIDAS SaaS. In particular, the paper illustrates how the extreme automation implemented in the MIDAS prototype includes the solutions of tough problems such as: (i) the configuration of the automated test execution engine against large and complex services architectures, (ii) the test input generation based on formal methods and constraint propagation, (iii) the test oracle generation based on state machine execution, (iv) the dynamic scheduling of test cases and the reactive, evidence-based planning of test campaigns with on the fly generation of new test cases, both based on based on probabilistic graphical inference. This paper reports some feedback from real-world case studies in the health-care and in the logistics sectors.

CCS Concepts

•Mathematics of computing → Bayesian networks; Probabilistic reasoning algorithms; •Information systems → Web services; •Networks → Cloud computing; •Theory of computation → Probabilistic computation; Bayesian analysis; •Software and its engineering → Software functional properties; *Ultra-large-scale systems;* **•Computer systems organization →** *Reliability;*

*Draft initially submitted to SAC 2016.

Keywords

service testing; test automation; test generation; test prioritization; test scheduling; test planning

1. INTRODUCTION

MIDAS [1] is a SaaS prototype that provides automation of service test activities such as functional, vulnerability and usage-based testing. In particular, the MIDAS prototype implements a complete intelligent automation solution for unit and end-to-end functional test. The prototype implements test automation methods of all the basic test tasks (input and oracle generation, engine configuration, execution, arbitration, reporting). The MIDAS prototype includes also intelligent optimizing test tasks such as dynamic prioritization of test case through test run time scheduling, and focused, on-the-fly generation and run of new test cases on the basis of evidences (reactive planning).

The MIDAS prototype brings enhanced solutions for the most critical service test automation problems: (i) configuration of the test engine against large and complex services architectures, (ii) test input generation based on formal models and constraint propagation (iii) test oracle generation based on executable specifications, (iv) dynamic test case prioritization and scheduling based on probabilistic graphical inference, (v) reactive planning of test campaigns with on-the-fly, evidence-based generation of new test cases, always based on probabilistic graphical inference. These test automation methods are provided as services by the MIDAS SaaS, through the MIDAS End User API.

The remainder of this paper is organized as follows. Section 2 presents the related work on service test automation. Section 3 illustrates in detail the architecture of the solution and its most important components. Section 4 sketches usages of the prototype in operational environments, i.e. for the test of real-world services architectures in the health-care and logistic sectors. Section 5 discusses the limitations of the current solution. In the conclusion we report some findings, and outline the planned future work.

2. RELATED WORK

Service test and, in particular, end-to-end test of complex services architectures is difficult, knowledge intensive, hard to manage and expensive in terms of labor effort, hardware/software equipment, and time to market. Since the inception of the service orientated approach, service testing automation has been a critical challenge for researchers and practitioners [5, 17]. In particular, tasks such as: (i) automated optimized generation of test inputs [5], (ii) automated generation of test oracles [4], and (iii) optimized management of test suites for different test purposes - such as first testing, re-testing, regression testing [17], has not yet found automation solutions that can be applied to real complex services architecture such as those that are implemented in real-world health-care services architectures [6].

Model-based testing (MBT) utilizes formal models (structural, functional, and behavioral) of the services architecture under test and of the test configuration to undertake the automation of the testing tasks [10]. The "first-generation" MBT research was essentially focused on test input generation. More recently, formal methods, especially SAT/SMT-based techniques have been leveraged [11] that allow the exhaustive exploration of the system execution traces, and efficient test input generation satisfying constraints (formal properties expressed in temporal logic). Jehan et al. [11] use a constraint solver to compute the expected inputs for each particular execution of the business process as extracted from the control flow graph.

Even if Bayesian reasoning and probabilistic graphical inference have been recently considered as an "ideal research paradigm for achieving reliable and efficient software testing" [15], the research on this topic is still in its infancy. In particular, the usage of probabilistic inference as a support of test scheduling is carried out, in our best knowledge, only by Mirarab and Tahvildari [14]. Their approach is static (prioritization for regression testing) but they consider dynamic scheduling, i.e. incorporating as evidence the feedback of the test engine for each test run (the test verdict) [13], as one of the most important subject of their future research.

3. AUTOMATING SERVICE FUNCTIONAL TEST

3.1 Model based automation

The automation of the test tasks (test generation, execution, arbitration, scheduling, reporting and planning) is obtained from a restricted set of models: (i) the service model, (ii) the Services Architecture Under Test (SAUT) model, (iii) the Test Configuration (TC) model, and (iv) the Protocol State Machine (PSM) model.

The service model is the standard definition of the service, i.e., for SOAP services, the WSDL document. The SAUT model is a blueprint, expressed in XML, of the services architecture under test: it allows defining its actual services, their APIs and the dependencies (actual wires) between them. The TC model is a blueprint of the test engine configuration that defines its components (stimulators, mocks, interceptors) and their connections with the SAUT services.

A stimulator is a virtual upstream service that is able to send test inputs to a SAUT service, receive replies and arbitrate them. A mock is a virtual downstream service that is able to receive inputs from a SAUT service, arbitrate them and transmit canned replies. An interceptor is able to catch, arbitrate and forward transmissions over an observed actual wire between two SAUT services. The MIDAS test engine is configured automatically from the SAUT and the TC models.

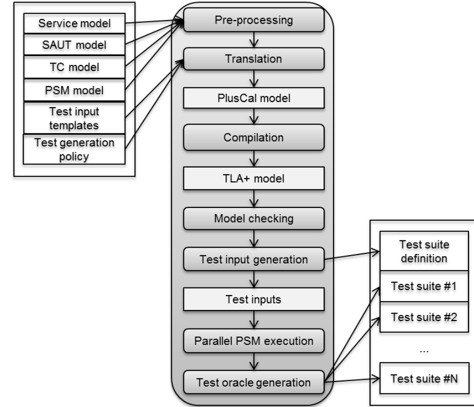

Figure 1: Automated generation of test cases

Each SAUT service and each TC mock is equipped with a PSM (protocol state machine), modeled as a Harel statechart [9], that represents the interaction states of the component and the transitions triggered by message receptions (events), filtered by Boolean conditions (guards) and producing effects expressed as data-flow transfer functions, i.e. expressions that calculate the elements of the message to be sent as functions of the elements of the received messages. The PSM is represented with the W3C standard SCXML [2] as an XML document. PSM conditions and data-flow transfer functions are expressed in JavaScript and XPath. SCXML artifacts are executable specifications of the SAUT service and TC mock external behavior.

For each stateful service, it is possible to define an XML resource that gives a representation of the service state that is observable by the service user. The developer supplies with her service an ancillary "external state" service that implements setState and getState stereotyped operations. The test engine calls these operations automatically in order to initialize pre-conditions (setState) and to check post-conditions (getState) as specified in the PSM. This feature allows double-checking of the stateful service operations, enabling to arbitrate not only the operation reply, but also the getState output.

3.2 Automated generation of test cases

Figure 1 sketches the activity diagram of the automated test case generation. The input artifacts are the service model, the test models (SAUT, TC, PSM), the test generation policy, and input templates. The test generation policy pilots the behavior of the test case generator. For instance, the test generation can be focused on specific types of messages to enable the exploration of specific logical (service operations) and "physical" (SAUT components) regions of the behavior

of the service architecture under test. Optionally, the user can supply test input payload templates that guide the test case generation by focusing, through regular expressions, on relevant ranges of data values (instead of using randomly-generated values) in the operation payload. The outputs produced by the generation task are:

- Test suite definition (TSD) - the test suite definition is an XML document that represents the abstract interaction paths that are the results of the symbolic execution of the collection of PSMs, starting from first test input classes.

- Test suite(s) (TSs) - a test suite is a collection of test cases; each test case is a partially ordered collection of instantiated messages (input, oracles) that instantiates a TSD interaction path.

The preprocessing phase performs consistency checking across all input artifacts, binds the service and mock PSMs through wires and build a parallel state machine which combines all individual PSMs.
The test case generation leverages model checking techniques provided by the TLA+ framework [12] that "implements" the well-known TLA formal specification language based on temporal logic. First, we translate the PSMs into the PlusCal, a TLA+ companion algorithm language supported by the TLA+ framework (Translation activity in Figure 1). The model obtained in PlusCal is then compiled into the TLA+ core language (Compilation activity in Figure 1). TLA+ is backed by the TLC model checker to exhaustively check correctness properties across all possible executions of the system and by the TLAPS proof system that relies on SMT (Satisfiability-Modulo Theory) solvers for checking TLA+ proofs. Through assertions, execution traces that match some criteria - for instance where messages of some specific types, or containing some specific values, are exchanged - are requested to the proof system (Model checking activity in Figure 1). Hence, the TLC model checker achieves the generation of the execution traces by temporal logic constraint propagation. The execution traces are defined as interaction paths in the TSD. Input data are then extracted from the execution traces (Test input generation activity in Figure 1). The obtained test inputs are then supplied one by one to the SCXML parallel state machine built as a composition of the individual PSMs and the SCXML execution engine is invoked (Parallel PSM execution activity in Figure 1). The execution of the parallel state machine is monitored and all events and the associated "messages" are intercepted, allowing the test generator to produce, for each test input, the corresponding test oracles and to constitute a test case that is compliant with an interaction path defined in the TSD and is placed in a test suite (TS).
Thanks to the test generation policy and the templates for the test input payloads provided by the tester, the test input generator can focus on interesting specific ranges of values for the test input payloads using different strategies to generate relevant data. Explored strategies are random, cyclic domain sampling, boundary values, and domain partitioning by analyzing the data-flow transfer functions in the PSMs.

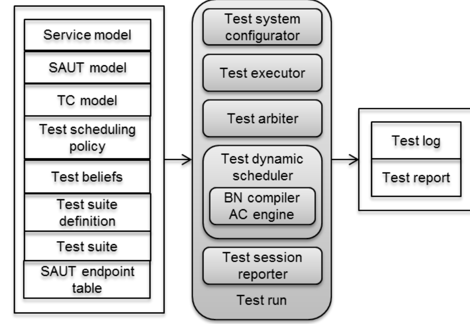

Figure 2: Automated test run with dynamic scheduling.

3.3 Automated test run with dynamic scheduling

The test run engine is configured automatically from the Service, the SAUT and the TC models, and the SAUT endpoint table (Figure 2). The engine configurator generates the test run components (stimulators, mocks and interceptors) from the TC model that are bound to the services under test following the specifications of the Service model, the SAUT model, and the SAUT endpoint table that gives the locations of the service endpoints.
The basic run manager executes and arbitrates the test cases of the test suite in the temporal sequence corresponding to their static order in the test suite (basic run method). More interestingly, the scheduled run manager handles a cycle schedule/execute/arbitrate where the scheduler is able to choose the next test case to run on the basis of past test verdicts (dynamically scheduled run method). The schedule/execute/arbitrate cycle continues until there are no more test cases to run or some halting condition is met. The automated test run method produces machine-readable test log and report.
The MIDAS approach to the prioritization of test cases is entirely original [13]: it is based on the usage of probabilistic graphical models [16] in order to dynamically choose the next test case to run on the basis of the preceding test verdicts. Namin and Sridharan [15] illustrate the well-known advantages of probabilistic inference - intractable exact inference in complex real-world domains, mathematically well-defined mechanism for representation, explicit modeling of uncertainty, management of multiple hypothesis about the state of the system under test, capability of integrating additional information about the state of the system. The Markov assumption (a key principle in Bayesian reasoning) that states that, given the system state S(t), the current action and the current observation, the state S(t) can be estimated conditionally independent of all prior states, actions and observations - is well suited for testing: (i) the presence/absence of a fault does not change during the test session, and (ii) each test run does provide an independent test verdict because the system under test is reinitialized after each test run - following a best practice recommended by the European Telecommunication Standard Institute [8].
The Markov assumption is the basis of the two-step iterative Bayesian inference process: a prediction step updates

the belief of all possible hypotheses of system state based on the prior belief and the actions taken since then, and a further correction step "corrects" the updated belief based on the correspondence between the expected and actual observations (evidences). Such an adaptive procedure is utilized for discovering service failures (the primary test goal), but also for troubleshooting, i.e. locating the faults that best explain the symptoms (failures) in the most efficient way.
For complex systems, such as services architectures, determining which elements are causing troubles is not always straightforward and often prone to inaccuracies. When end to end testing of a service build new release, which is produced by a corrective maintenance action, reveals a failure of another service that, eventually, does not interact directly with the updated one, we are confronted with the exposure of implicit and hidden tight coupling between services that can be considered a "defect" (of both) but is not, strictly speaking, a bug. Its is very important to discover this kind of troubles, and the Bayesian Network approach to troubleshooting bestows several advantages because the BN inference results: (i) are probabilistic, allowing managing the uncertainty in the decision process and the intractability of exact inference, (ii) can be mathematically proven [7], (iii) are knowledgeable, in contrast to other approaches, such as Neural Networks, which act as black boxes.
The dynamic scheduler builds a Bayesian Network (BN) model [16] from: (i) the SAUT model - allowing the association of random Boolean variables with SAUT elements; (ii) the test cases - passive actions in the troubleshooting jargon, because they do not change the state of the system; (iii) the optional users' beliefs on the SAUT - prior probabilities and evidences, such as the information that the maintenance action on the service build has fixed the source of a failure revealed by an identified test case in the preceding test campaign). The BN variable types are associated to SAUT structural elements such as the services and the required/provided interfaces and with SAUT functional elements such as the service operations.
In order to move further the size and computation speed limits, the classical representation of the BN is "compiled" in a more compact structure (the Arithmetic Circuit - AC), adapted to more efficient inference computation [13]. At each test run, the verdicts are inserted as evidences in the AC and the subsequent inference calculates a failure probability for each remaining test case that, combined with a scheduling policy (e.g. max-failure, min-failure, max-entropy...), allows the scheduler to choose the next test cases to run (Figure 3). Full details of the approach are reported in [13].
The scheduler performs policy-driven, dynamic and intelligent prioritization of test cases that aims at: (i) precocious discovery of failures, and (ii) precise localisation of faulty elements (troubleshooting), in different testing contexts such as first test, re-test and regression test.

3.4 Automated evidence-based reactive planning of test campaigns

The full automation of a test campaign requires that the automated generation of test cases be driven not only by models but also by test goals expressed through policies. The scheduler is able not only to drive the choice among a set of existing test cases but also to establish a dynamic

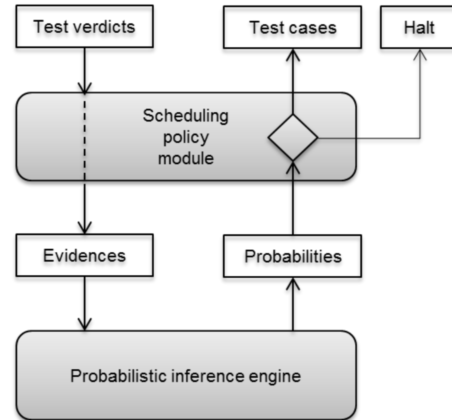

Figure 3: Scheduling cycle.

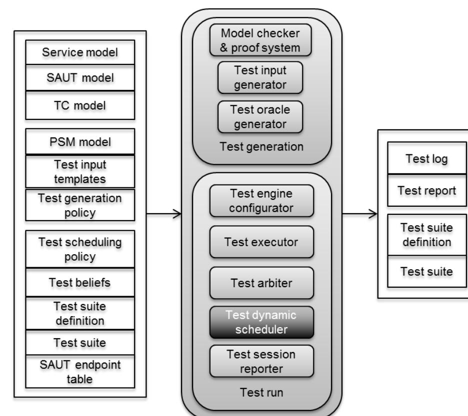

Figure 4: Automated, evidence-based reactive planning of test campaigns.

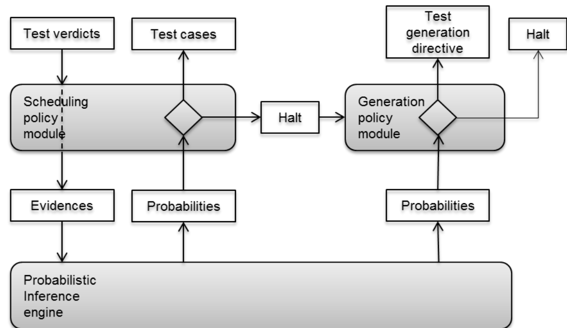

Figure 5: Planning cycle.

relationship between test case prioritization and the model-driven generation of new test cases, and to supply to the generator evidence-driven directives.
With the reactive planning test method, the user can: (i) either provide an initial previously generated or hand-written test suite, that could be a re-testing suite, a test regression suite or a smoke test suite (preliminary testing to reveal simple failures severe enough to reject a prospective service build release), (ii) or invoke the reactive planning test method without any test suite, with a generation policy that provokes a first generation from scratch (for instance, in first testing contexts) (Figure 4).
Within the test session in progress, on the basis of evidences (verdicts) accumulated from the past test runs, the scheduler calculates the degree of ignorance (Shannon entropy) on SAUT elements and eventually recommends the generation of new test cases whose execution would diminish this ignorance (for example by including test cases that trigger scenarios involving scarcely stimulated services of the architecture). If the recommendation is followed, the test generator is solicited by the scheduler with appropriate directives - that are calculated from the test generation policy - about the execution traces to be considered (Figure 5). The newly generated test cases are taken into account by the scheduler (with an updating of the BN/AC) and the scheduled execution is restarted on the new test suite.

4. REAL-WORLD CASE STUDIES

We have a first feedback of the usage of the functional test methods from the MIDAS "pilots", i.e. collections of real-world services and services architectures in the sectors of health-care and logistics that the MIDAS "pilot" partners have been developed in the context of separate business and research projects. In the health-care sector, one of the most relevant operational services architecture targeted by the MIDAS test methods is the Calabria Cephalalgic Network (CCN) [6], an application implemented as a multi-owner standard [3] microservices architecture whose components are physically deployed in separate premises and private clouds.
The logistics pilot is a reference implementation of a standard supply chain management architecture. Companies and institutions place their service builds in the architecture and test the semantic interoperability with the MIDAS test methods.
The feedbacks of the usage of the MIDAS facility reports a real contribution of the MIDAS prototype on issues that have been labeled as: (i) "test case overhead", (ii) "unit testing only", (iii) "poor planning" and (iv) "manageability".
The "test case overhead" issue is particularly critical in the health-care sector and relates to the necessity of creating a huge amount of test cases since the standard services to be tested are specified as generic and the payload structure varies according to their instantiations [3]. In addition, typical payloads transferred in the health-care domain are made of very complex data structures with several thousands of atomic data types. The automated generation of test cases brought by the MIDAS prototype reduces dramatically the effort that was formerly dedicated to test case handwriting. Moreover, home-made, open source and commercial off-the-shelf testing frameworks are able to support only service unit testing. End-to-end test of service compositions with MIDAS requires only the drafting of the appropriate SAUT, TC and PSM models, which is a challenging task, but is accomplished once (the models are relatively stable). The generation/run of test suites that evolve following the maintenance process (the cycle test/debug for first test, re-test, regression test) can be performed in an optimized manner with test scheduling and planning by using policies, input templates, prior probabilities and evidences. Furthermore, with the aforementioned huge amount of test cases, test optimization is a must.
The "poor planning" issue is related to the fact that home-made, open source and commercial off-the-shelf testing frameworks have no support for test optimization (focused test case generation, test case dynamic prioritization). MIDAS policy based generation, intelligent scheduler and evidence-based planning propose solutions to the optimization problem that are technically operational, potentially very powerful and whose evaluation is in progress. We and, above all, our users shall constitute assets of experience and know-how in testing using such advanced features.
Last but not least, with the currently available tools every change in the deployed SAUT (IP addresses, ports, URIs, parameterizations) requires a significant effort of reconfiguration by hands, practically preventing any automated continuous integration approach that require the deployment of distinct "copies" of the SAUT ("manageability" issue). With the MIDAS prototype, the SAUT, TC, PSM models and the generated test suites are independent of the SAUT components' physical locations that are indicated as configuration parameters to be instantiated at test run time (SAUT end-point table).

5. PROTOTYPE LIMITATIONS

This section lists the limitations of the actual prototype in terms of service test automation, optimization and routinization. First of all, the prototype is able to test only SOAP services and full SOAP architectures, even if the test models (SAUT, TC, PSM) are already able to represent REST/XML services, REST/JSON services and services architectures in which all these protocols are present. With the spread of REST/JSON services, this limitation restricts our service test automation scope. The test generator and arbitrator are unable to work with passive oracles, i.e. oracles

that are not messages ready to be transmitted, but message templates in which some parts are left unspecified. This obliges the user to supply in each PSM a complete set of message building rules (JavaScript data-flow transfer functions or literals). This limitation is prejudicial for service test automation and optimization.

Another important usage limitation is that, in the actual prototype, the consistency of the actual configuration of the services architecture under test with the SAUT model and the SAUT endpoint table is not checked per se. Hence consistency errors can be revealed, but not clearly identified and documented, only when a test run scenario touches the improperly defined services and endpoints. This is detrimental for service test routinization. This kind of check can be performed only if the user implements for each service and instantiates in the appropriate endpoint an ping-like ancillary service that allows the test engine to check that a certain API is actionable at a certain URL. This is not a big charge for the user, and we even think that it fulfills a best practice of service design for testability.

In re-testing and regression testing contexts, the inference engine should be able to take into account as evidences the relationship between the failed test cases in the preceding test campaigns and the new service build release. This relationship could be labelled as: "this service build release fixes the defects that were the causes of campaignN-1/failure1, campaignN-2/failure3 ...". Actually, without appropriate information we could only hypothesize that all the failures revealed in the preceding campaigns are repaired in the new service build release, which does not cope with the current debugging/fixing practices. If the information indicated above were supplied with the service build release, the service test optimization could be really improved.

Evolving services architectures grow increasingly complex, both spatially with densely interconnected cyber-physical systems and functionally with emerging behavior, raising a continuous challenge on functional testing. From an algorithmic standpoint, the never-ending increasing complexity of the functionalities will always highlight the limitation of current approaches. Therefore, evolving advanced formal techniques are even more necessary to cope with this growing complexity, both from the scheduler and the test generator perspectives. To leverage and efficiently combine these techniques in a complete automated setting, the functional testing framework must autonomously be able to dynamically induce higher abstractions from in-depth analysis of entire regions of the system's complex behaviour. We believe the Cloud offers the best infrastructure to support such a demanding task, in particular with auto-scaling, resource pooling, and elasticity.

6. CONCLUSION

The collection of functional test automation methods of the MIDAS prototype covers all the service functional test tasks, including the most "intelligent" and knowledge-intensive ones. These test methods bring solutions to tough functional test automation problems such as: (i) the configuration of the automated test engine against large and complex services architectures, (ii) the test input generation based on formal methods and constraint propagation, (iii) the test oracle generation based on service behavior specifications, (iv) the intelligent dynamic scheduling of test cases, (v) the intelligent, evidence-based, reactive planning of test campaigns. Furthermore, the test automation methods are provided as services, allowing the MIDAS SaaS user to invoke them individually, to easily combine them in complex procedures and to routinize their usage in automated integration and delivery workflows. These methods are currently used, "tested" and evaluated by the MIDAS pilot partners. Experiences for assessing and mastering advanced features such as model-based generation, dynamic scheduling and reactive planning for first testing, re-testing and regression testing are in progress on real-world services architectures.

Current known drawbacks of the MIDAS prototype are manageability and usability issues that are the targets of future work: (i) taking into account, beyond SOAP and REST/XML, also REST/JSON service testing, in services architectures that "mix" all these protocols; (ii) checking the alignment of the SAUT actual deployment with the provided SAUT model; (iii) better handling of passive oracles, i.e. oracles generated from incomplete specifications.

7. ACKNOWLEDGMENTS

This research has been conducted in the context of the MIDAS project (EC FP7 project number 318786) partially funded by the European Commission.

8. REFERENCES

[1] http://www.midas-project.eu.

[2] http://www.w3.org/TR/scxml/.

[3] https://hssp.wikispaces.com/.

[4] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *Software Engineering, IEEE Transactions on*, 41(5):507–525, May 2015.

[5] M. Bozkurt, M. Harman, and Y. Hassoun. Testing and verification in service-oriented architecture: a survey. *Softw. Test. Verif. Reliab.*, 23(4):261–313, June 2013.

[6] D. Conforti, M. C. Groccia, B. Corasaniti, R. Guido, and R. Iannacchero. EHMTI-0172. Calabria cephalalgic network: innovative services and systems for the integrated clinical management of headache patients. *The Journal of Headache and Pain*, 15(Suppl 1):D12, 2014.

[7] R. Dechter. Bucket elimination: A unifying framework for probabilistic inference. In E. Horvitz and F. V. Jensen, editors, *UAI '96: Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence*, pages 211–219. Morgan Kaufmann, 1996.

[8] European Telecommunications Standards Institute. Methods for Testing and Specification (MTS); Automated Interoperability Testing; Methodology and Framework. ETSI Guide 202 810 V1.1.1, ETSI, 2010.

[9] D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[10] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and

H. Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):9:1–9:76, Feb. 2009.

[11] S. Jehan, I. Pill, and F. Wotawa. Functional SOA testing based on constraints. In 8th *Int. Workshop on Automation of Software Test (AST)*, pages 33–39, 2013.

[12] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[13] A.-P. Maesano. *Bayesian dynamic scheduling for service composition testing*. Ph.D. Thesis, Université Pierre et Marie Curie - Paris VI, Jan. 2015.

[14] S. Mirarab and L. Tahvildari. A Prioritization Approach for Software Test Cases Based on Bayesian Networks. In M. B. Dwyer and A. Lopes, editors, *Fundamental Approaches to Software Engineering (FASE) 2007*, volume 4422 of *LNCS*, pages 276–290. Springer, 2007.

[15] A. S. Namin and M. Sridharan. Bayesian reasoning for software testing. In G. Roman and K. J. Sullivan, editors, *Workshop on Future of Software Engineering Research (FoSER), at the* 18th *ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*, pages 349–354. ACM, 2010.

[16] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[17] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.