Service functional test automation

Lom Messan Hillah, Ariele-Paolo Maesano, Fabio de Rosa, Libero Maesano, Marco Lettere, Riccardo Fontanelli

▶ To cite this version:

Lom Messan Hillah, Ariele-Paolo Maesano, Fabio de Rosa, Libero Maesano, Marco Lettere, et al.. Service functional test automation. 10th Workshop on System Testing and Validation, Fraunhofer Fokus, Oct 2015, Sophia Antipolis, France. ⟨hal-01240254⟩

HAL Id: hal-01240254

https://hal.sorbonne-universite.fr/hal-01240254v1

Submitted on 8 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright - All rights reserved

Service functional test automation

Lom Messan Hillah, Ariele-Paolo Maesano
Laboratoire d'Informatique de Paris VI, Sorbonne Universités, UPMC
lom-messan.hillah@lip6.fr, ariele.maesano@lip6.fr
Fabio De Rosa, Libero Maesano
Simple Engineering France SARL
fabio.de-rosa@simple-eng.com, libero.maesano@simple-eng.com
Marco Lettere, Riccardo Fontanelli
Dedalus S.p.A
marco.lettere@dedalus.eu, riccardo.fontanelli@dedalus.eu

Abstract:
This paper presents the automation of the functional test of services (black-box testing) and services architectures (grey-box testing) that has been developed by the MIDAS project and is accessible on the MIDAS SaaS. In particular, the paper illustrates the solutions of tough functional test automation problems such as: (i) the configuration of the automated test execution system against large and complex services architectures, (ii) the constraint-based test input generation, (iii) the specification-based test oracle generation, (iv) the intelligent dynamic scheduling of test cases, (v) the intelligent reactive planning of test campaigns. The paper describes the usage of the MIDAS prototype for the functional test of an operational distributed application in the domain of healthcare.

Introduction

Services are everywhere. They are involved in services architectures built of service components that: (i) expose *service APIs*, (ii) interact through service protocols (REST/XML, REST/JSON, SOAP…) and (iii) are deployed independently of each other. The SOA approach has been used for fifteen years to let distributed vertical applications cooperate. More recently, systems have exposed service APIs for interaction with mobile apps. Presently, the internal structure of applications, once monolithic, is going to be designed as a micro-services architecture [9] that is particularly well adapted for cloud deployment. Services are loosely coupled, allowing agility of design, development, integration (continuous integration - CI), delivery (continuous delivery - CD) and deployment.
The **Calabria Cephalalgic Network** (CCN) [3] is a multi-owner distributed application that supports the *headache integrated care processes*,

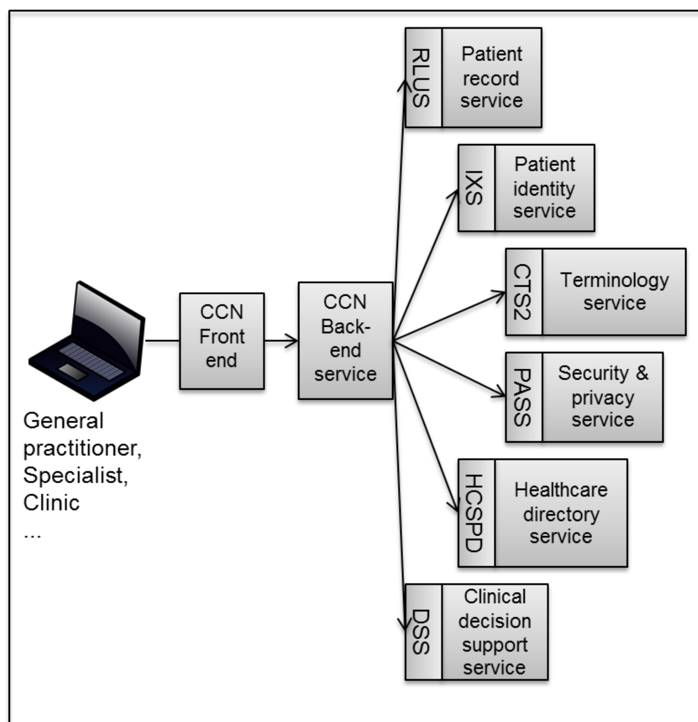

Figure 1. Calabria Cephalalgic Network.

effectively coordinating different care settings (general practitioners, specialists, clinics, labs...) in a patient-centred vision. The application is designed as a services architecture (Figure 1), is operational today and its components' services are physically deployed in different data centres. The service APIs are compliant with the HL7/OMG HSSP international standards (RLUS, IXS, CTS2...) [13].

Dedalus [12], a company specialised in healthcare systems, is in charge of the provision within the CCN of the Patient record, the Patient identity and the Terminology services. CCN service-oriented architecture allows Dedalus to put in place a modular integration process with one separate source code repository and one separate build per service component.

Actually, the service integration process is a full testing process, constituted of all the testing activities: functional, security, fault tolerance and performance test. In order to improve agility and time-to-market, these activities shall be organised in an optimized manner. The service integration and delivery process pattern that is becoming popular is the *CD pipeline* [15], in which the testing activities are placed as *stages* between the *service build* formation and its deployment in the production environment. The transition from a stage to the next is permitted only whether the stage tests *pass*, otherwise the sequence is interrupted and restarts with the check-in of the updated code. An example of service CD pipeline is sketched in Figure 2.

The test tasks in each stage and the chosen sequence of the test stages can and should maximise the effectiveness (the *fault exposing potential* and the *troubleshooting efficacy*) and the efficiency (the *fault detection rate*) of the testing tasks. Test effectiveness and efficiency are important even for completely automated stages - to say nothing about manual ones - that can be heavyweight and can slow the entire process.

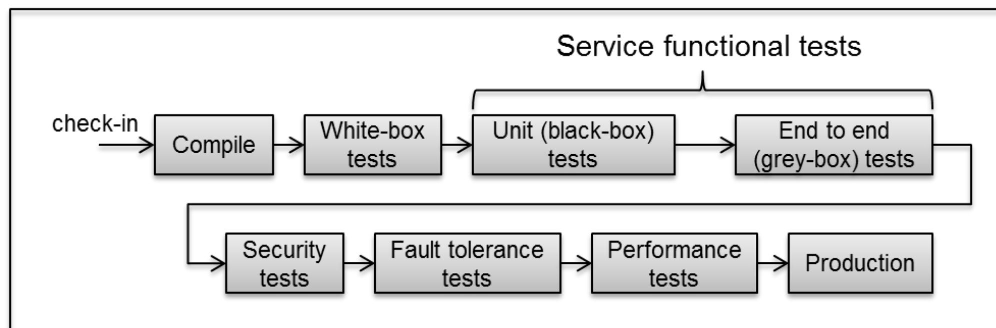

Figure 2. Service integration process as a pipeline.

In the CD pipeline sketched in Figure 2, the successfully constituted build is firstly submitted to acceptance white-box tests. All the subsequent test stages target different aspects of the service external behaviour and are independent of the service implementation technology. The subsequent two stages are about functional test and are detailed in the section '**Automated functional test**'. The security tests follow - they can be effective and efficient only whether the service build passes the functional tests. The last two stages are about quality of service: the fault tolerance tests challenge the *resilience* of the service implementation in the face of failures of the underlying computing resources or the unavailability of the services it interacts with. Lastly, the performance tests concern mainly the service invocation and provision *latency*. The CD pipeline can be more or less automated. A single *CD pipeline stage* can be fully automated whether: (i) its internal tasks can be fully automated and produces automatically a meaningful report and (ii) the automated tasks can be invoked through APIs by the CD server (for instance Jenkins [14]).
This paper reports a solution of automation of service functional test. The section '**Related work**' gives the motivation for doing research on the topic and a short review of the state of the art. The section '**Automated functional test**' presents the prototype developed within the MIDAS project and provided as-a-service by the MIDAS SaaS [16]. Dedalus has incorporated the functional test automation services in its integration process: this experience is presented in the '**Prototype usage in an operational environment**' section. The '**Conclusion**' discusses major advantages and drawbacks of the new solution and outlines future work.

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 labour effort, hardware/software equipment and time-to-market. Since the inception of the service orientation, service testing automation has been a critical challenge for researches and practitioners [2] [1] [11]. In particular, tasks such as: (i) automated optimised generation of test inputs [2], (ii) automated generation of test oracles [1], and (iii) optimised management of test suite for different test activities - such as first testing, re-testing, regression testing [11], has not yet found automation solutions that can be applied to real complex services architecture such as those that are implemented in healthcare [3].

Model-based testing (MBT) utilises formal models (structural, functional and behavioural) of the services architecture under test to undertake the automation of the testing tasks [5]. The “first-generation” MBT research is essentially focused on test input generation. More recently, formal methods, especially SAT/SMT-based techniques have been leveraged [6] that allow the exhaustive exploration of the system execution traces, and efficient test input generation satisfying constraints (formal properties expressed in temporal logic). Jehan and colleagues [6] use a constraint solver to compute the expected inputs for each particular execution of the business process as extracted from the control flow graph.

The MIDAS approach to the prioritization of test cases [11] is entirely original [8]: it is based on the usage of probabilistic graphical models [10] [7] in order to dynamically choose the next test case to run on the basis of the preceding verdicts. Moreover, the scheduler is able to establish a dynamic relationship between test case prioritization and the generation of new test cases, by supplying on the fly to the generator evidence-driven directives based on the preceding verdicts.

Automated functional test

Test environments

The service functional test automation is illustrated through an example of a simplified services architecture related to the CCN application (Figure 3). In order to provide its service, eHealth service consumes the Patient record, the Patient identity and the Terminology services. These services that are

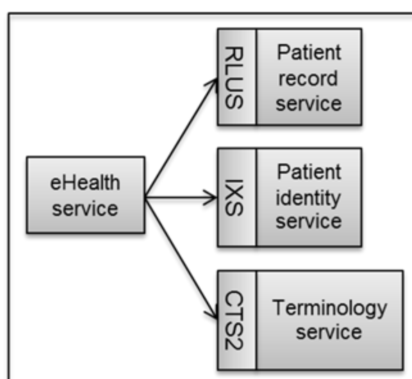

Figure 3. Services architecture under test.

not consumers of other services are called *terminal services*.

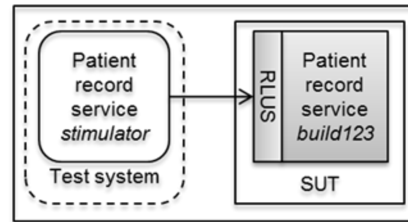

Figure 4. Test environment for terminal service unit test.

Unit test stage

The unit test stage includes the following tasks: (i) produce test inputs (stimuli), (ii) produce test oracles (expected outcomes), (iii) deploy and initialise the build in an appropriate environment (service under test - SUT), (iv) configure and generate the test system, (v) bind the test system with the SUT, (vi) run test cases (transmit stimuli, collect and log outcomes), (vii) arbitrate test outcomes against test oracles, (viii) schedule test case runs (dynamic scheduling), (ix) plan test campaigns (reactive planning) and (x) report test campaigns. For every terminal service, the unit test environment architecture is similar to that sketched in Figure 4.

For non-terminal services, such as the eHealth service, the typical unit test environment is depicted in Figure 5. The test tasks involved in the stage are the same as those for terminal services, but in the test system are generated, in addition to the *stimulator*, three *mocks* that “virtualise” the downstream services. The binding sub-task enables the mocks receipt the requests of the eHealth service, and send back the canned responses. The test system must be able to evaluate against the oracles that the requests that are issued target the appropriate services, are in time, are in the exact sequence and are the right ones.

End-to-end test stage

The services architecture under test (SAUT) distributed environment is deployed with the lastest release builds of the downstream services. In the test system are generated the *interceptors* that catch the exchanges forth and back between the eHealth service and the downstream services (Figure 6). This test environment is put in place in the end-to-end stages of the CD pipelines of all the services involved in the SAUT, including the terminal services. An interesting point is that the end-to-end tests can highlight functional failures of any of the SAUT services - not only of the service of the pipeline

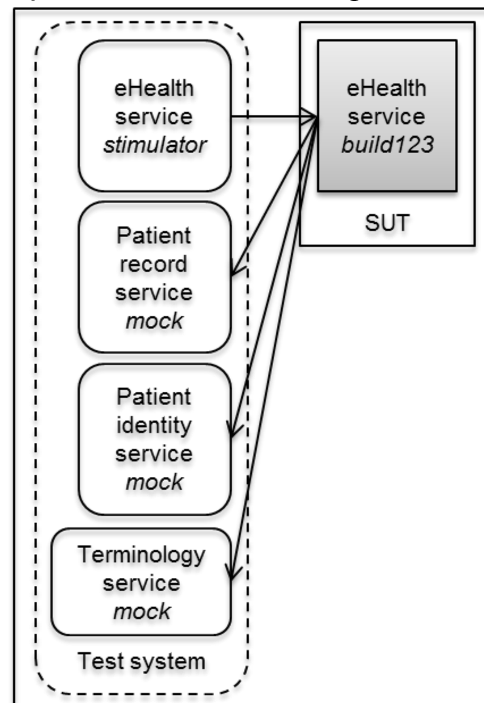

Figure 5. Test environment for non-terminal service unit test.

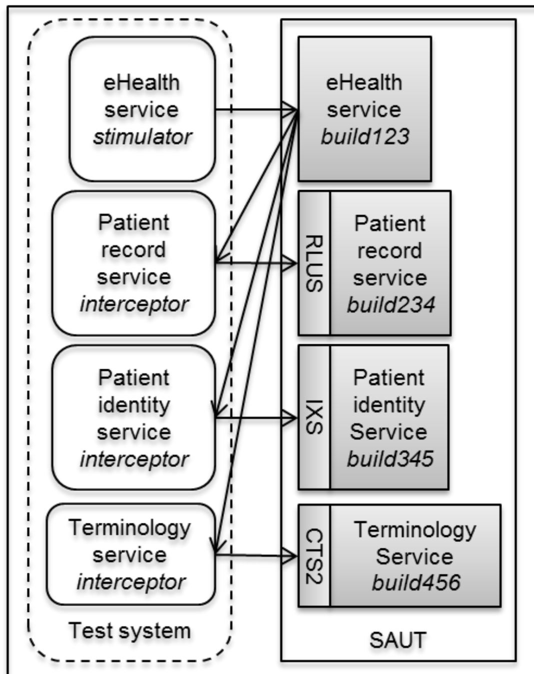

Figure 6. Test environment for end to end test.

in which the stage is accomplished - and so eventually reveal *service tight coupling* - when a change in one service produces an unexpected failure in another service.

End-to-end testing of multi-owner services architecture requires *collaborative testing projects* that involve all the service owners and that explore systematically the cooperation scenarios between all the services. Systematic end-to-end testing campaigns are mandatory for first testing of new distributed applications, but are also recommended as regular activities of re-testing and regression testing. A collaborative testing project involving all the owners of the CCN application services is in progress.

Test automation methods

The MIDAS functional test prototype brings automation solutions (*test automation methods*) for the most critical test tasks: (i) configuration of the test system against distributed services architectures, (ii) test case (input/oracle) generation based on constraint propagation and symbolic execution, (iii) intelligent dynamic test case prioritisation and scheduling, (iv) intelligent reactive planning of test campaign with on-the-fly, evidence-based generation of new test cases. These test automation methods are provided as services by the MIDAS SaaS.

Automated configuration of the test system

The structure of the test system (stimulators, mocks, interceptors) is automatically generated from the SAUT model and the test configuration model. The former model is represented through an XML document depicting the *actual components* of the SAUT and the *actual wires* between them – interaction links that are typed by *service specifications* (e.g. WSDL documents). The latter model is obtained from the former model: (i) by adding *virtual components* (stimulators, mocks) and the corresponding *virtual wires* to actual components and (ii) by designating the *actual wires to be observed* (interceptors).

Automated generation of test cases

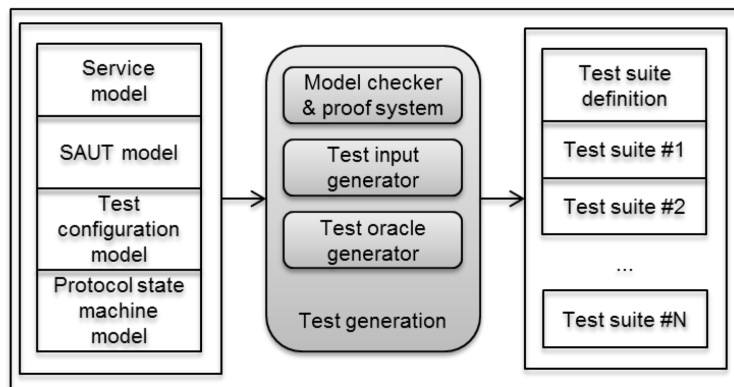

Figure 7. Automated generation of test cases.

Each SAUT component is equipped with a *protocol state machine* (PSM), modelled as a Harel state-chart [4], that represents the *interaction states* of the component and the *transitions* triggered by received messages (*events*), filtered by conditions (*guards*) and producing *effects* described as data-flow transfer functions. The service component PSMs are represented through standard SCXML documents [17] and the conditions and transfer functions are expressed in Javascript.

Test cases (inputs and oracles) are generated from the set of models (Figure 7). The test cases generation process relies on model-checking the PSM models using TLA+ [18], a well-known formal specification language based on temporal logic. 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. The PSMs and the generation parameters are translated into a TLA+ companion algorithm language (PlusCal) that is afterwards compiled into TLA+. Through assertions, *execution traces* of the system 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. Input data are then extracted from the execution traces and fed to the SCXML engine, which executes the PSMs for the scenarios triggered by the input data and produces the related oracles.

Automated dynamic scheduling of test runs

Automated dynamic scheduling takes places in the MIDAS test system that is equipped with automated execution and arbitration of test cases (Figure 8). In this context, the scheduler is able to choose the next test case to run on the basis of the past test verdicts. The cycle schedule/execute/arbitrate continues until there are no more test cases to run or some halting condition is met. The objectives of dynamic scheduling are (i) precocious detection of failures and (ii) localisation of faulty elements (troubleshooting).

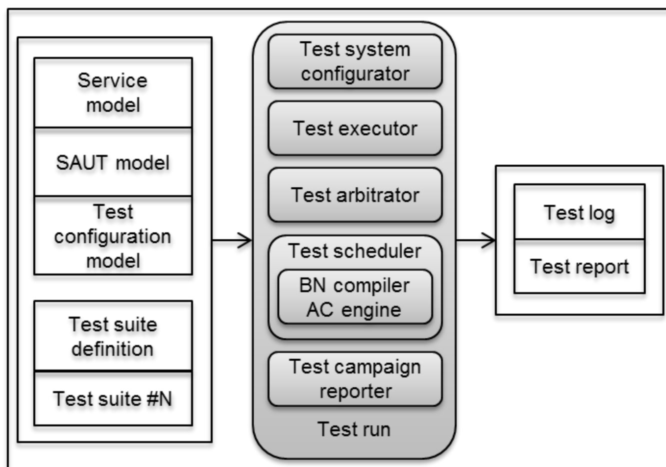

Figure 8. Automated scheduled execution of test cases.

The dynamic scheduler builds a Bayesian Network (BN) model [10] from (i) the SAUT model, (ii) the test suite and (iii) user's beliefs on the SAUT. The BN is compiled into an Arithmetic Circuit (AC) [7]. At each test run the verdicts are inserted as *evidences* in the AC and the subsequent inference calculates a *fitness* probability for each remaining test case that, combined with a scheduling policy (e.g. *max-fitness*, *max-entropy*...), allows the scheduler to choose the next test case.

Automated reactive planning of test campaigns

The idea behind a fully automated workflow for functional testing is to use the scheduler to drive not only the choice among a set of existing test cases but also the generation of new test cases. The test campaign starts with a minimal test suite and, on the basis of evidences (verdicts) brought from the past test runs, the scheduler calculates the degree of ignorance (Shannon entropy) on SAUT elements and recommends the generation of test cases whose execution would diminish this ignorance. This feature is operational and its usage in test campaigns is in progress.

Prototype usage in an operational environment

Dedalus currently utilises a home-made framework for service unit testing that has already significantly shrunk the effort of manually producing and executing test cases and test suites. The major limitations of this solution can be labelled as: (i) "test case overhead", (ii) "unit testing only", (iii) "lack of planning and scheduling", (iv) "manageability". The "test case overhead" issue relates to the necessity of creating a huge amount of test cases since the services to be tested (such as RLUS) are specified as *generic* and the payload structure varies according to the instantiation of the service. In addition, typical content transferred in the healthcare domain is 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, the home-made testing framework is able to support only service unit testing. End-to-end test of service compositions with MIDAS requires only the drafting of the appropriate SAUT, test configuration and PSM models.
With the aforementioned huge amount of test cases, the optimisation of the test campaigns is a must. The home-made test framework doesn't have any support for test cases prioritization and test case generation optimization. MIDAS intelligent scheduler and reactive planning facility propose solutions to the optimisation problem that are technically operational and whose evaluation is in progress.
Last but not least, with the home-made framework every change in the deployed SAUT (IP addresses, ports, URIs, parametrizations) requires a significant effort of reconfiguration by hands of every individual test case, practically preventing any continuous integration approach. With the MIDAS prototype, the SAUT models, the test configuration models, the PSMs and the generated test suites are independent of the SAUT physical locations that are indicated as configuration parameters to be instantiated at test run time.

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-based ones. Furthermore, the test automation methods are provided as *services*, allowing the MIDAS SaaS user both to invoke them individually and to easily combine them in service integration and delivery processes directed by CI/CD servers. These methods are actually integrated as services by a MIDAS partner (Dedalus) in its specific integration and delivery process of healthcare distributed applications and services architectures. Experiences for assessing and mastering advanced features such as dynamic scheduling for re-testing and regression testing and evidence-based test case generation are in progress.
Current drawbacks of the MIDAS prototype are manageability and usability issues and are the matters of future work: (i) taking into account REST/JSON service testing; (ii) automated check of the alignment of the SAUT deployment with the SAUT model; (iii) simplifying the specification of the test configuration; (iv) better handling of passive oracles (generated from incomplete specifications).

References

1. Barr, E., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2015). The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5).
2. Bozkurt, M., Harman, M., & Hassoun, Y. (2010). Testing web services: A survey. Department of Computer Science, King's College London, Tech. Rep. TR-10-01.
3. Conforti, D., Groccia, M. C., Corasaniti, B., Guido, R., & Iannacchero, R. (2014). EHMTI-0172."Calabria cephalalgic network": innovative services and systems for the integrated clinical management of headache patients. *The journal of headache and pain*, 15(1), 1-1.
4. Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8 (3), 231-274.
5. Hierons, R. M., Bogdanov, K., Bowen, J. P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A. J. H., Vilkomir, S., Woodward, M. R., & Zedan, H. (2009). Using formal specifications to support testing. *ACM Comput. Surv.*, *41* (2), 1-76.
6. Jehan, S., Pill, I., & Wotawa, F. (2013, May). Functional SOA testing based on constraints. In *Proceedings of the 8th International Workshop on Automation of Software Test* (pp. 33-39). IEEE Press.
7. Maesano, A. P. (2015). *Bayesian dynamic scheduling for service composition testing*. Ph.D. Dissertation, University Pierre et Marie Curie, Paris.
8. Namin, A. S., & Sridharan, M. (2010). Bayesian reasoning for software testing. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, (pp. 349-354). New York, NY, USA: ACM.
9. Newman, S. (2015). *Building Microservices*. O'Reilly Media, Inc.
10. Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
11. Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritisation: a survey. *Softw. Test. Verif. Reliab.*, *22* (2), 67-120.
12. http://www.dedalus.eu/
13. https://hssp.wikispaces.com/
14. https://jenkins-ci.org/
15. http://martinfowler.com/bliki/DeploymentPipeline.html
16. http://www.midas-project.eu
17. http://www.w3.org/TR/scxml/
18. http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html