Efficient testing based on logical architecture

Liu, Huai; Spichkova, Maria; Schmidt, Heinrich; Ulrich, Andreas; Sauer, Horst; Wieghardt, Jan

Liu, H., Spichkova, M., Schmidt, H., Ulrich, A., Sauer, H., & Wieghardt, J. (2015). Efficient testing based on logical architecture. Proceedings of the 24th Australasian Software Engineering Conference (ASWEC 2015), 49–53. https://doi.org/10.1145/2811681.2811691
Document Version: Accepted Manuscript

Published Version: https://doi.org/10.1145/2811681.2811691

Efficient Testing based on Logical Architecture

Huai Liu, Maria Spichkova, Heinz W. Schmidt
RMIT University
Melbourne, Australia
{huai.liu, maria.spichkova, heinz.schmidt}@rmit.edu.au

Andreas Ulrich, Horst Sauer, Jan Wieghardt
Siemens AG, Corporate Technology
Munich, Germany
{andreas.ulrich, horst.sauer, jan.wieghardt}@siemens.com

ABSTRACT

The rapid increase of software-intensive systems' size and complexity makes it infeasible to exhaustively run testing on the low level of source code. Instead, the testing should be executed on the high level of system architecture, i.e., at a level where component or subsystems relate and interoperate or interact collectively with the system environment. Testing at this level is *system testing*, including hardware and software in union. Moreover, when integrating complex, distributed systems and providing support for conformance, interoperability and interoperation tests, we need to have an explicit test description. In this vision paper, we discuss (1) how to select tests from logical architecture, especially based on the dependencies within the system, and (2) how to represent the selected tests in explicit and readable manner, so that the software systems can be cost-efficiently maintained and evolved over their entire life-cycle. In addition, we further study the relevance between different tests, based on which, we can optimise the test suites for efficient testing, and propose optimal resource allocation strategies for cloud-based testing.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.11 [**Software Engineering**]: Software Architectures

Keywords

logical architecture, dependencies between services, testing

1. INTRODUCTION

With the increase of system size and complexity in software-intensive distributed systems, it has become necessary to coordinate and integrate the work from various technical and human-centered disciplines under the generic context of system engineering. The latest engineered systems should comply with the principles of different fields, such as software engineering, control engineering, project management, etc.

ASWEC ' 15 Vol. II, September 28-October 01, 2015, Adelaide, SA, Australia

DOI: http://dx.doi.org/10.1145/2811681.2811691

Nowadays, engineered systems are becoming more and more software-intensive; as a result, software engineering has become an indispensable and crucial practice in the system engineering process that interconnects the various disciplines.

As an example of engineered systems we consider the class of automation systems that are used in factory automation or process automation solutions as they occur today in many different production systems ranging from electromechanical assembly units for goods such as cars to chemical or biological reaction processes and many more. The heart of an automation system is a programmable logic controller (PLC) that executes a control program in hard real-time to observe signals from the plant and control the industrial process near instantly via sensors and actors.

Today's automation systems resemble highly complex large-scale systems. They consist of 100 and more PLCs that process up to 100,000 I/O signals. The lifespan of an automation system is typically in the range of 10 to 15 years. During this time uninterrupted operation of the plant is expected. Nevertheless the plant has a much higher lifespan of 30 and more years, which implies a modernization of the automation system, sometimes even during continued operation of the plant. These conditions require tremendous efforts for quality assurance for the automation system as the whole and the control programs of the PLCs in particular.

Automation solutions of the future need to cope with higher flexibility, better integration and improved efficiency at highest quality level. To design, develop and operate such upcoming future automation systems, new approaches are required that address the paradigm shift that is currently happening from statically planned, centrally controlled and hierarchically built systems towards flexible, highly meshed and autonomously operating, so called *cyber-physical* systems. Engineering such systems includes a commissioning phase that is itself one of the more complex and costly phases due to specific deployment constraints, architectures and processes. Post commissioning testing has to deal with many combinatorial elements resulting from a myriad of configuration options, different interconnections and communications choices in different deployments despite high levels of reuse across projects and the increase in sheer volume, velocity and variety of data to be processed and reacted to in the deployment environment.

One of the major objective of software engineering is to guarantee the quality of the system under development and in operation. Testing, a mainstream approach to quality assurance through detecting failures, is confronted with new challenges on today's distributed, global, and complex cyber-

physical systems that are increasingly deployed using cloud-enabling with services and/or service compositions. One major challenge is the infeasibility of testing on the low level of source code. In the context of component-based and service-oriented architecture, a service is developed and owned by a third-party organisation and realised in one or more specific self-contained third-party components. The service consumer can only access the service through the service's description. In other words, source code of the services are normally unavailable to users. Even though the source code is accessible, the complexity of today's large-scale systems will make the testing based on source code time-consuming, very expensive, and even infeasible.

As a matter of fact, we do not need the complete information about the system to conduct testing in a fairly effective way. We need to find which local information (at the level of one or more components or subsystems) is sufficient for system-level testing. In our previous work, we elaborated a formal approach that (1) allows modelling and analysis of data and control flow dependencies, and (2) provides a logical system architecture based on the dependencies within/among services [10] and components [8].

Contribution: In this vision paper, we propose a testing framework based on the logical architecture. We convert the traditional white-box control-flow and data-flow based testing into a new technique using the information of data and control flow dependencies among services on the level of system architecture. The new testing technique can help testers generate and select tests that can satisfy certain architecture-level control-flow or data-flow coverage criteria. In addition, we present an approach to further minimising test suite with the purpose of using as few tests as possible to achieve the highest coverage. Given that the tests are generated based on the dependencies between services, we also discuss how to describe the relevance between tests. We further investigate how to improve the testing efficiency by precisely allocating resources for testing in the cloud computing environments. One of the aim of the proposed framework is to incorporate the ideas of cloud-based testing into the Test Description Language (TDL, cf. [13]). The main advantage of TDL is the representation test descriptions as scenarios, i.e., on a higher abstraction level than programming or scripting languages. This allows better readability and understandability of tests, and also conforms with the human-oriented software development, cf. [9, 11].

Outline: The rest of the paper is structured as follows: In Section 2, we discuss the testing-oriented methodology we are implementing in our framework, and introduce the core ideas of analysis of dependencies within systems, with the aim to find the minimal part of model/system needed to run a specific test suite or to verify a particular property. In Section 3, we discuss the related work on test selection and propose a family of architecture-level testing coverage criteria based on the dependencies within systems. We also present some potential research directions for further improving the efficiency of testing. Section 4 concludes the paper by highlighting the main contributions and describing the future work directions.

2. OUR FRAMEWORK

2.1 Testing-oriented optimisation

The development of innovative, flexible and sustainable systems means stronger requirements on the development and testing methodologies, as well as new challenges on the optimisations of the system architecture and the testing approaches. Following the idea of test-driven development (cf., e.g., [5]), we aim at testing-oriented optimisation of the logical architecture of the system, which means that we would improve testing efficiency and the overall system quality, as well as cost-efficiently maintain and evolve the systems over their entire life-cycle. Thus, this would contribute to the *technical sustainability* (i.e., cost-effective longevity and endurance, cf. [7]) of software architecture, and as result to the technical sustainability of the system itself.

In many cases, to run a test suite, the complete information about the system is not required. However, it might be complicated to identify which parts of the system are sufficient for this check. To allow the automation of the analysis, we solve this problem on a formal level first.

In our work we focus on the following questions:

- How we optimise the engineered system making its architecture more suitable for efficient testing? To solve this problem, we suggest to use testing-oriented optimisation of the system architecture. The optimisation is based on the analysis of the dependencies between components (cf. Section 2.2).
- How we select the minimal set of required test suites? We suggest a solution based on logical architecture analysis (cf. Section 3.1).
- How we optimise the testing effectiveness? To solve this problem, we suggest to analyse relevance between tests, based on which we can further improve the testing cost-effectiveness through improving the diversity among tests (cf. Section 3.3). In addition, we suggest to take the cloud computing into consideration and allocate different computing resources for different tests, aiming at enhancing testing efficiency in cloud.

Figure 1 presents the general methodology we implement in our framework. On the one hand, the architecture of the engineered system have to be optimised for the testing: the dependencies within the system have to be analysed also taking into account the kind of properties we have to check for the system (cf. Section 2.2 for more details). On the other hand, we have to generate optimized test suites also taking into account the relevance among tests (cf. Section 3.3) and the test suites coverage (cf. Section 3.1). On this basis, we can perform testing on a cloud-based platform such as Chiminey [14], a bioscience data platform.

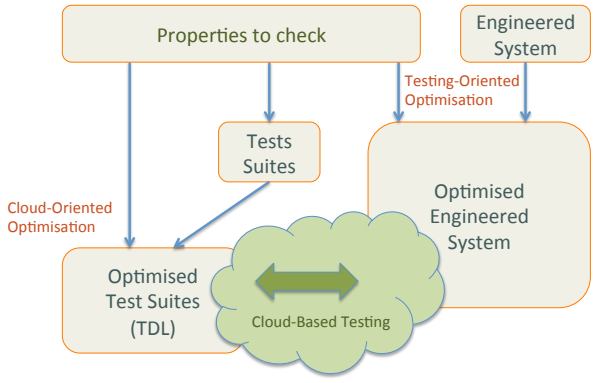

Figure 1: Testing Framework: Methodology

One aim of our frameworks is to incorporate the ideas of sustainable software development and cloud-based testing into the upcoming ETSI standards [4]. We also aim at increasing the readability and understandability of tests, to conform with the ideas of human-oriented software development, cf. [9, 11]. For this purpose, we are going to represent the optimized test suites using the Test Description Language (TDL, cf. [13]). In our approach, we see TDL as an intermediate representation of automatically generated tests. TDL allows to represent test suites (test descriptions) as scenarios, which helps to bridge the gap between high-level test purpose specifications and executable test cases. Thus, TDL can be used as an intermediate representation of tests generated from simulators, test case generators, or logs from previous test runs. It provides a generic language for the formal specification of test suites and makes a clear distinction between concrete syntax and a common abstract syntax.

2.2 Dependencies within the system

Any additional information about the system can make the whole process of verification or testing slower, more expensive or even infeasible, especially if we are speaking about formal verification. Thus, we have to find the minimal part of model needed to run a specific test suite or to verify a particular property. We suggest an approach focusing on data and control flow dependencies between services [10] as well as between components [8]. The dependencies are analysed with the aim to provide a logical decomposition of the system architecture that is especially appropriate for the case of remote monitoring, testing and/or verification. Thus, by adapting this approach to our framework, we can perform a *testing-oriented optimisation* of the logical architecture of the system. It is also crucial to have an efficiency analysis wrt. which test suites should be performed locally, and which should be sent to the Cloud platform. Therefore, we extend our framework by an appropriate model covering all these aspects. Moreover, the resulting architecture will be more sustainable [7].

Figure 2 gives an example to illustrate the dependencies among services in a system consisting of 8 services. The input/output dependencies within a service are represented by dashed lines. Green ovals denote local variables. The channels with large size of data packages within a data flow and frequency they are produced, are denoted by thick red arrows. The services requiring the use of high-performance computing and Cloud virtual machines, are marked by white colour. All other channel and services are marked blue.

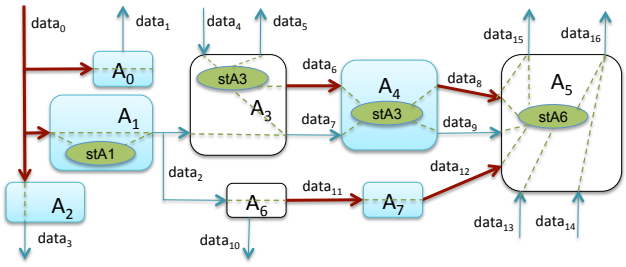

Figure 2: Analysis of dependencies within a system

Following the definition of services introduced in [2], we specify a service S as a partial function from input streams $\mathbb{I}(S)$ to output streams $\mathbb{O}(S)$. Then, on the logical level, a test suite (as well as a system property) can be represented by relations over data and control flows on the system's channels.

For the test suite ts, let I_{ts} and O_{ts} be the sets of input and output channels, respectively, for which data are required to perform the testing. For each channel from O_{ts} we recursively compute all the sets of the dependent input channels. We define by $\mathbb{I}_{ts}^{\mathcal{D}}$ the union of these sets. $\mathbb{I}_{ts}^{\mathcal{D}}$ should be equal to I_r, otherwise we should check whether the test suite was specified correctly. In such a way we can exclude some results of (human) errors [9] in the specification of the test suite. This also helps to make the test suite more precise it or eliminate unnecessary elements.

This allowed us, especially in the case of cloud-supported processing, to reduce the costs of monitoring, testing or verification.

3. TESTING STRATEGIES

3.1 Test selection: Related work

In the testing on the level of source code, coverage-based testing is a mainstream technique which selects tests based on the information relevant to the structure of the software under test [15]. Two main types of code coverage criteria are those based on control flow and data flow, respectively. Control-flow coverage makes use of the control constructs of the software under test. Statement coverage, for instance, requires the selected tests to run each executable statement at least once. In addition, branch coverage requires the selected tests to exercise every feasible branch at least once. Data-flow coverage considers certain patterns of data manipulations, including the definition (normally abbreviated as *def*) and the usage (abbreviated as *use*) of a datum. Many data-flow based techniques have been proposed based on different criteria, such as all-defs, all-uses, etc.

Though the concept of coverage was originally proposed for unit testing on the code level, it has been extended to different contexts. Spillner [12] has proposed some coverage criteria for integration testing. In the context of control-flow based integration testing, the interactions among modules and operations were considered. The simplest criterion is that every module has been executed at least once. More complicated criteria include export operation coverage, import operation coverage, etc. In the context of data-flow based integration testing, the focus was on the parameters among different operations/modules. Parameters are normally defined in the calling operations/modules, and used in the called operations/modules.

Hashim et al. [6] also worked on integration testing, but using the basic concepts in the traditional component-based architecture. They proposed three criteria, namely gate link, gate prefix, and gate path coverage criteria, which are analogous to control-flow criteria. In addition, probabilities were introduced to these criteria such that one could precisely measure how well the components and connectors are covered given a test suite.

Bartolini et al. [1] introduced coverage testing into service-oriented systems. One major challenge of testing in the service-oriented architecture is the infeasibility of white-box testing techniques (including traditional code-level coverage testing). A methodology called SOCT (service-oriented coverage testing) was proposed to measure to what extent various service features are covered. SOCT makes use of the interfaces defined in WSDL (Web Service Description Lan-

guage) to provide a variety of operations for coverage measurement. The SOCT methodology has been applied into the testing of service orchestrations based on BPEL (Business Process Execution Language) to evaluate the coverage of message exchange operations.

3.2 Test selection based on logical architecture

In contrast to all studies that were discussed in the previous section, we attempt to propose coverage criteria on a more abstract level, more specifically, on the level of logic architecture. Based on the logical dependencies among services proposed in our previous study [10] and discussed in Section 2.2, we can have the architecture-level control-flow and data-flow coverage criteria as follows:

Architecture-level service coverage The selected tests should exercise every service at least once. This criterion is the counterpart to the code-level statement coverage.

Architecture-level branch coverage The selected tests should exercise every possible branch between services at least once.

Architecture-level all-defs coverage The selected tests should cover each def of every datum at least once.

Architecture-level all-uses coverage The selected tests should cover each use of every datum at least once.

Among the above five criteria, the first two can be considered as architecture-level control-flow coverage criteria, while the remaining two as architecture-level data-flow coverage criteria.

Consider the example given in Figure 2. A test suite containing the tests in Table 1 is sufficient for achieving 100% of the architecture-level service coverage.

Table 1: Tests for architecture-level service coverage

	Test than can traverse the path:
TC#1	$data_0 \rightarrow A_1 \rightarrow data_2 \rightarrow A_3 \rightarrow data_6 \rightarrow A_4 \rightarrow data_8 \rightarrow A_5 \rightarrow data_{15}$
TC#2	$data_0 \rightarrow A_1 \rightarrow data_2 \rightarrow A_6 \rightarrow data_{11} \rightarrow A_7 \rightarrow data_{12} \rightarrow A_5 \rightarrow data_{16}$
TC#3	$data_0 \rightarrow A_0 \rightarrow data_1$
TC#4	$data_0 \rightarrow A_2 \rightarrow data_3$

However, to achieve 100% of the architecture-level branch coverage, another test (given in Table 2) is required besides the above four tests.

Table 2: An extra test for architecture-level branch coverage

	Test than can traverse the path:
TC#5	$data_0 \rightarrow A_1 \rightarrow data_2 \rightarrow A_3 \rightarrow data_7 \rightarrow A_4 \rightarrow data_9 \rightarrow A_5 \rightarrow data_{15}$

As the basis for the architecture-level data-flow coverage, we define the *architecture-level def* as the data output from a service, while *architecture-level use* as the data input into a service. In the system depicted in Figure 2, the architecture-level defs include $data_1$ from A_0, $data_2$ from A_1, $data_3$ from A_2, $data_5$ from A_3, $data_6$ from A_3, $data_7$ from A_3, $data_8$ from A_4, $data_9$ from A_4, $data_{10}$ from A_6, $data_{11}$ from A_6, $data_{12}$ from A_7, $data_{15}$ from A_5, and $data_{16}$ from A_5; while the architecture-level uses include $data_0$ into A_0, $data_0$ into A_1, $data_0$ into A_2, $data_2$ into A_3, $data_2$ into A_6, $data_4$ into A_3, $data_6$ into A_4, $data_7$ into A_4, $data_8$ into A_5, $data_9$ into A_5, $data_{11}$ into A_7, $data_{12}$ into A_5, $data_13$ into A_5, and $data_14$ into A_5.

The test suite consisting of TC#1 to TC#5 and the tests given in Table 3 can achieve 100% of the architecture-level all-defs coverage.

Table 3: Extra tests for architecture-level all-defs coverage

	Test than can traverse the path:
TC#6	$data_0 \rightarrow A_1 \rightarrow data_2 \rightarrow A_3 \rightarrow data_5$
TC#7	$data_0 \rightarrow A_1 \rightarrow data_2 \rightarrow A_6 \rightarrow data_{10}$

In the test suite for architecture-level all-defs coverage, TC#1 covers $data_2$ from A_1, $data_6$ from A_3, $data_8$ from A_4, and $data_{15}$ from A_5; TC#2 additionally covers $data_{11}$ from A_6, $data_{12}$ from A_7, and $data_{16}$ from A_5; TC#3 additionally covers $data_1$ from A_0; TC#4 additionally covers $data_3$ from A_2; TC#5 additionally covers $data_7$ from A_3 and $data_9$ from A_4; TC#6 additionally covers $data_5$ from A_3; and TC#7 additionally covers $data_{10}$ from A_6.

To achieve 100% of the architecture-level all-uses coverage, we can use the test suite containing TC#1 to TC#5 and the tests given in Table 4.

Table 4: Extra tests for architecture-level all-uses coverage

	Test than can traverse the path:
TC#8	$data_4 \rightarrow A_3 \rightarrow data_5$
TC#9	$data_{13} \rightarrow A_5 \rightarrow data_{15}$
TC#10	$data_{14} \rightarrow A_5 \rightarrow data_{16}$

In the test suite for architecture-level all-defs coverage, TC#1 covers $data_0$ into A_1, $data_2$ into A_3, $data_6$ into A_4, and $data_8$ into A_5; TC#2 additionally covers $data_2$ into A_6, $data_{11}$ into A_7, and $data_{12}$ into A_5; TC#3 additionally covers $data_0$ into A_0; TC#4 additionally covers $data_0$ into A_2; TC#5 additionally covers $data_7$ into A_4 and $data_9$ into A_5; TC#8 additionally covers $data_4$ into A_3; TC#9 additionally covers $data_{13}$ into A_5; and TC#10 additionally covers $data_{14}$ into A_5.

3.3 Relevance among tests

The relationship among architecture-level tests is complicated. Recall the tests (TC#1 to TC#5) in the previous Section 3.1, no pair of tests has the simple relation of "one is executed before the other". All of them show certain degrees of relevance to one another. For example, TC#1 and TC#5 are similar to each other in the former part of their paths, but quite different in the latter part. We propose the use of similarity between two tests to reflect their relevance.

One popularly used similarity metric is Jaccard index, which is defined as the ratio between the size of the intersection and the size of the union of two sets. For example, the similarity between TC#1 and TC#5 can be calculated as $\frac{2}{11} = 0.18$, if we treat $data_i$ and A_j as equal elements in a test and do not consider the sequence in the path of each test. Given a test suite, we can then calculate the relevance between each pair of tests that can show how similar/different one test is from others.

Table 5: Another two possible tests for the system shown in Figure 2

	Test than can traverse the path:
TC#11	$data_0 \rightarrow A_1 \rightarrow data_2 \rightarrow A_3 \rightarrow data_6 \rightarrow A_4 \rightarrow data_9 \rightarrow A_5 \rightarrow data_{15}$
TC#12	$data_0 \rightarrow A_1 \rightarrow data_2 \rightarrow A_3 \rightarrow data_7 \rightarrow A_4 \rightarrow data_8 \rightarrow A_5 \rightarrow data_{16}$

Diversity among tests has been widely acknowledged as an important factor to affect the fault-detection effectiveness [3]: Normally, the more diversified the selected tests are, the more likely they can detect more faults. Given two test suites, we can select the suite with higher degree of diversity, which can be indicated by a lower similarity. For example, for two test suites {TC#1, TC#5} and {TC#1, TC#11} (TC#11 is given in Table 5), TC#5 is less similar to (or more diversified from) TC#1 than TC#11, so we would prefer to use the suite {TC#1, TC#5} in the testing.

In addition, the relevance information can also help selecting new tests. Suppose that we already generated the test TC#11, and we have two candidates TC#5 and TC#12 ((TC#12 is given in Table 5)) for the next test, based on the similarity information, we should select TC#12 instead of TC#5 as the next test to be executed. In a word, the coverage criteria can help us achieve a certain degree of adequacy in testing, while the test relevance can further help increase the diversity among tests; both adequacy and diversity can, in turn, significantly enhance the testing effectiveness.

Tests can include more information, for example, where a service should be executed (such as locally or in the cloud). With such information, we can further improve the test suite such that some tests are mainly associated with service(s) that must be tested locally, while other tests with those that can be executed in the cloud. We can also add the information about the required resources for running a service, and thus design various tests that are associated with different resource requirements. Based on such arrangements, we can precisely allocate testing resources in the cloud computing environments, including which tests need to be run on the local machines, which tests can be executed in the cloud, and how many resources can be assigned to certain tests.

4. CONCLUSION

Nowadays, large-scale distributed systems are rapidly engineered and widely deployed in the cloud. The large scale and high complexity of such systems make it almost infeasible to run the comprehensive testing on the level of source code. In this paper, we propose a series of techniques for the testing on the level of system architecture, so that the software systems can be cost-efficiently maintained and evolved over their entire life-cycle. This would increase the technical sustainability of the system architecture and, as result, of the overall system.

We proposed a family of architecture-level coverage criteria based on the data and control flow dependencies within the system. These criteria, in turn, can help generate test cases that can achieve a certain degree of test adequacy, as well as facilitate the minimisation of existing test suites without largely jeopardising the fault-detection effectiveness. We further presented how to evaluate the relevance between two tests, on the basis of which, we can further improve the testing effectiveness by pursuing the diversity among tests. Finally, we discussed how to make use of cloud computing to optimise the testing efficiency by precisely allocating various resources among well-designed tests.

In this paper we focus on the problem of incremental architecture-based testing, ignoring (for now) the specifics of real-time embedded nature of many of the components/subsystems in cyber-physical and post-commissioning testing. In the future work, we aim to cater for cloud-based soft commissioning in which significant portions of the deployment environment are simulated on high-performance cloud servers. For simplicity and focus, the paper is also content with an architecture based notion of testing that does not fully articulate the combinatorial variation resulting from architectural variation points in product lines. However we believe that our approach can be extended to such more general architectures with further investigations.

5. REFERENCES

[1] C. Bartolini, A. Bertolino, and E. Marchetti. Introducing service-oriented coverage testing. In *Proc. of ASE'08*, pages 57–64, 2008.

[2] M. Broy. Service-oriented Systems Engineering: Specification and design of services and layered architectures. The JANUS Approach. *Eng. Theor. Software Intens. Syst.*, pages 47–81, 2005.

[3] T. Y. Chen, F.-C. Kuo, H. Liu, and W. E. Wong. Code coverage of adaptive random testing. *IEEE T. Reliab.*, 62(1):226–237, 2013.

[4] European Telecommunications Standards Institute (ETSI). http://www.etsi.org/.

[5] J. Fröhlich, S. Jell, and A. Ulrich. Applying TDL to describe tests of a distributed RT control system. In *Proc. of UCAAT'14*, 2014.

[6] N. L. Hashim, S. Ramakrishnan, and H. W. Schmidt. Architectural test coverage for component-based integration testing. In *Proc. of QSIC'07*, pages 262–267, 2007.

[7] H. Koziolek. Sustainability evaluation of software architectures: A systematic review. In *Proc. of QoSA-ISARCS'11*, pages 3–12, 2011.

[8] M. Spichkova. Formalisation and analysis of component dependencies. *Archive of Formal Proofs*.

[9] M. Spichkova, H. Liu, M. Laali, and H. Schmidt. Human factors in software reliability engineering. In *Proc. of WAHESE'15*, 2015. in press.

[10] M. Spichkova and H. Schmidt. Towards logical architecture and formal analysis of dependencies between services. In *Proc. of APSCC'14*, 2014. in press.

[11] M. Spichkova, X. Zhu, and D. Mou. Do we really need to write documentation for a system? In *Proc. of MODELSWARD'13*, 2013. arXiv:1404.7265.

[12] A. Spillner. Test criteria and coverage measures for software integration testing. *Software Qual. J.*, 4(4):276–286, 1995.

[13] A. Ulrich, S. Jell, A. Votintseva, and A. Kull. The ETSI Test Description Language TDL and its application. In *Proc. of MODELSWARD'14*, pages 601–608, 2014.

[14] I. Yusuf, I. Thomas, M. Spichkova, S. Androulakis, G. Meyer, D. Drumm, G. Opletal, S. Russo, A. Buckle, and H. Schmidt. Chiminey: Reliable computing and data management platform in the cloud. In *Proc. of ICSE'15*, pages 677–680, 2015.

[15] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comp. Surv.*, 29(4):366–427, 1997.