Leveraging composability in model-based testing for microservices

Bas van den Brink[1,2], Tannaz Zameni[3], Ulyana Tikhonova[4], Lammert Vinke[2] and Ana-Maria Oprescu[1]

[1]University of Amsterdam, Science Park 904, 1098 XH Amsterdam, The Netherlands
[2]Info Support, De Kruisboog 42, 3905 TG Veenendaal, The Netherlands
[3]University of Twente, Drienerlolaan 5, 7522 NB Enschede, The Netherlands
[4]Axini, Azartplein 12, 1019 PD Amsterdam, The Netherlands

Abstract
The microservices architecture is becoming increasingly more popular in contemporary software systems. Microservices communicate with each other over a network using specific communication protocols, such as HTTP or AMQP. Microservices systems can pose various challenges, with testing being one of them. Model-based testing is an established formal method for testing software systems. However, testing microservices using model-based testing is an open research topic. The goal of our research is to explore formally testing microservices systems. This research is divided in two parts: (1) we develop a theoretical framework for formally testing microservices, extending existing work on ioco-theory; (2) we implement the theoretical framework in a proof-of-concept to experiment with microservices systems from both open-source and industry.

Keywords
microservices, model-based testing, compositional testing

1. Introduction

In recent years, distributed architectures such as microservices [18] are becoming popular in industry [9, 21, 30]. In comparison to monolithic systems, microservices offer independent scaling and simpler deployment procedures, amongst other benefits [13, 26]. However, implementing a microservice architecture also presents some challenges. These challenges arise inherently due to the distributed interaction between services. For instance, testing microservices is regarded to be more complex than testing monolithic applications [13].

This complexity comes from the fact that each microservice is deployed as a separate software artifact. With monolithic systems, software components compose using simple function calls. In microservice systems, this composition is done by establishing a network connection between each microservice. Such a composition introduces additional variables during testing, such as: network instability, (de)serialization errors and communication protocol issues.

As an example, microservices systems might contain hundreds of small components that are separately deployed [24]. These microservices work together to form software systems

BENEVOL'22: The 21st Belgium-Netherlands Software Evolution Workshop Mons, 12-13 September 2022
bas.vandenbrink@infosupport.com (B. v. d. Brink); t.zameni@utwente.nl (T. Zameni); tikhonova@axini.com (U. Tikhonova); lammert.vinke@infosupport.com (L. Vinke); a.m.oprescu@uva.nl (A. Oprescu)

that handle a variety of complex tasks. Companies such as Netflix [24], Amazon [23] and Zalando [20] are known to implement the microservices architecture for their software systems.

Testing a single microservice is arguably straightforward. The complexity of testing microservices lies in testing the integration between these small components [13]. Microservices are typically integrated using a networking protocol, such as HTTP or AMQP [5]. While there exists tooling to test software systems from end-to-end, many tools require the software developer to write and maintain these tests. End-to-end tests are described as flaky [28], expensive to write/maintain and expensive to execute [1, 8, 27]. A model alleviates the maintainability problem, and generating test cases from written specifications rather than building and applying test cases reduces costs and effort [3]. If we leverage the composability of models, then we could avoid fully deploying the end-to-end test infrastructure and suite, and contribute considerable savings.

Model-based testing [38, 37] is a formal technique to automatically generate and execute test cases from a specification. The specification models the behavior of the system under test. The test scenarios are generated from such a specification automatically. The system is tested as a black-box and is interacted with through its inputs and outputs. While model-based testing could be used with various formalisms, such as Finite State Machines and Abstract Data Types [22], we use Labelled Transition Systems to leverage research conducted by Jan Tretmans et al. [35] and its industrial applications, such as Axini's Axini Modeling Plaform (AMP) [7]. Ultimately, this formal testing theory helps us to decide whether the system conforms to its specification.

Indeed, model-based testing can be fitting as a means to perform integration tests for microservices. Additionally, model-based testing has the benefit that tests are automatically generated, meaning that there is less overhead in writing the tests themselves. When the specification of the system changes, new tests are generated automatically.

Research proposes compositional model-based testing as a technique to model components separately and combine these models to test the combination of components [11]. The main benefit is that these models are smaller, making them easier to write and maintain.

Compositional testing is still ongoing research [10, 25, 15], while its applicability to microservices remains an open research topic. With this study, we want to focus on the applicability of compositional model-based testing in combination with microservices.

Our vision is that if compositional testing is applicable to test microservices, three main issues with contemporary testing are tackled:

1. Writing and maintaining expensive integration tests ultimately becomes obsolete, as model-based testing allows for the generation and execution of these tests.
2. Models become smaller and easier to maintain. With the inclusion of CI/CD pipeline support and a proper framework, model-based testing can become more accessible to development teams.
3. Model-based testing can be applied in a more incremental manner, allowing it to be part of the development phase of a project.

This research project is still in an early stage. With this paper, we aim to provide an initial description of the project. Section 2 informally describes theoretical preliminaries for this paper. Section 3 discusses related work, Section 4 presents our methodology, Section 5 aims to provide an informal description of the current ideas behind testing microservices and Section 6 provides an outlook for the rest of the project.

2. Background

This section describes two main concepts used in this paper: transition systems and microservice communication. Transition systems are the formalism of choice for testing microservices systems in our research. Microservice communication form the basis of our proposed software testing theory.

2.1. Transition Systems

There exists various formalisms that can be utilized for model-based testing [37]. In our study, we use transition-based models as a formalism for modeling the behavior of microservices. Transition-based models, or state-transition models, focus on describing the transitions between different states of the system [37]. This formalism is commonly used to describe the behavior of reactive or control-oriented systems, where outputs do not only depend on the current input but also on earlier inputs. We consider microservises to fit well in this scope.

In a simplistic form, *Labelled Transition Systems (LTS)* [35] contain a set of states with transitions between them. There exists various extensions of labelled transition systems. *A Labelled Transition System with Inputs and Outputs (IOLTS)* [35] distinguishes transitions with inputs or outputs of the system. Similarly, a *Symbolic Transition System (STS)* [12] extends upon the LTS with the notion of data and variables. Naturally, there exists a version of the STS with a distinction between inputs and outputs: the *IOSTS*.

2.2. Microservice Communication

There are two aspects regarding microservice communication, which are depicted in table 1. The first column regards the relation between sending and receiving entities: *one-to-one*, or *one-to-many*. The second column determines whether the communication is *synchronous* or *asynchronous*. The *Request/Response* communication style can be observed when synchronous one-to-one communication is applied. With this communication style, a sender will send a message (or request) to exactly one receiver. The sender will await the response of the receiver before continuing program flow.

When *one-to-one* communication is applied *asynchronously*, we observe the *Notification* communication style. In this case, the sender will still send a response to a receiver, but will *not* await a response. The sender will continue its program flow regardless of how the request is handled by the receiver. In the case of asynchronous messaging with one-to-many receivers, we observe the *Publish/Subscribe* communication style. Typically, a message-broker is introduced to handle the communication between the sender and receivers. Receivers can subscribe to a specific message topic. A sender can publish a message to such a topic. The message-broker then ensures that the message is delivered to all subscribers of that topic.

Table 1
A summary of microservice communication styles [31]

	One-to-one	**One-to-many**
Synchronous	Request/response	-
Asynchronous	Notification	Publish/subscribe

3. Related work

There currently exists tooling to test microservices from end-to-end, such as Protractor [4] or Cypress [14]. These frameworks utilize the browser to interact with the system through its web interface. However, these testing techniques require the developer to manually write and maintain the testing suite.

Behavior Driven Development (BDD) testing tools can also be used to test microservices systems. Examples of these tools are Cucumber [32] and SpecFlow [36]. This technique allows practitioners to create a specification of their software system's behavior using a human-readable domain specific language. BDD tools then use this specification to automatically generate most of the test code with predefined inputs and assertions. The practitioner then can implement the rest of this test code and test the software system. With model-based testing, the formal model contains enough information to fully automate test generation and execution.

Furthermore, there have been various research efforts regarding microservices testing. For example, Gazzola et al. [19] present a tool that automatically generates test cases based on the run time traces of deployed microservices systems. These test cases are then executed on a microservice in isolation. All external dependencies of the microservice are replaced by a stub. The behavior of this stub is also determined by the collected runtime traces. The main difference with our work is that our test cases are derived from a formal model instead of execution traces.

Another tool proposed by research is EvoMaster [6]. This tool can automatically generate tests for REST and GraphQL API's, which are commonly used technologies in microservices architectures. EvoMaster uses AI-techniques and (optionally) program analysis to find test cases. This approach does not require any human intervention to generate tests, whereas model-based testing techniques require a formal model to be constructed. Here lies the trade-off between these methods: EvoMaster's approach does not require manual labour, at the cost of a few hours of running its search algorithm. Optional code analysis (white-box testing) improves the effectiveness of the test cases several times over. With our approach, we construct a model which does not require code analysis, and is still able to perform automated tests. We see the major difference in how the knowledge is captured: EvoMaster learns it from the code base, we learn it from the model created by the practitioner.

Tools such as TorXakis [34] and Axini Modeling Platform (AMP) [7] already exist to perform model-based testing on software systems. However, our work explores a tailored theory towards the modeling and testing of microservices. We develop a proof-of-concept that implements this theory specifically. This allows us to see if such a theory for testing microservices differs from general model-based testing approaches.

In our work, we explore the notion of a network environment as an additional transition system during the composition of microservices. We are inspired by the works of both de Alfaro and Henzinger [2] and Daca et al. [15]. Both studies investigate how computing a separate environment during composition can alter the properties of the final composed model. We explore in our work how a network environment alters the properties of the final composed model between microservices.

Van der Bijl et al. [11] conclude in their study that the ioco relation does not hold after composition of underspecified models. As a solution, they propose *Demonic Completion* of underspecified specifications. In short, Demonic Completion is an algorithm that maps all underspecified transitions to a *chaos* state that accepts all behaviors. A benefit of Demonic Completion is that the ioco relation is preserved after composition. A downside of Demonic Completion is that the model accepts any behavior during testing, which possibly weakens the quality of the model.

4. Methodology

The goal of this research is to investigate the applicability of compositional model-based testing in the context of microservices. The first research question is focused on the differences between models of regular software components and models of microservices. To formally model microservices we choose the *Symbolic Transition System with Inputs and Outputs (IOSTS).* A transition system is able to capture the behavior of microservices. Moreover, symbolic transition systems contain the notion of data and variables. As microservices process vast amounts of data, symbolic transition systems make for a suitable modeling formalism.

RQ1: *To what extent does a symbolic input-output transition system capture the behavior of microservices?*
To the best of our knowledge, compositional testing has not yet been applied in the context of microservices. This raises the question: "If we could model and test a single microservice, can we also combine the output of these tests as an input of other microservices?", leading to the following research question:

RQ2: *To what extent can we test microservice systems using compositional model-based testing?*
For the second research question, we develop a formal framework for Symbolic Transition Systems and their compositions. Additionally, this theory is implemented in a proof-of-concept that can be used to test microservices systems. With this proof-of-concept, we will test actual microservices systems. We plan on testing two systems during this research project: the open-source eShopOnContainers [17] and a microservices system from industry provided by Info Support. Due to timing constraints, we have scoped the number of systems under test to two. Moreover, we choose the eShopOnContainers system as it is an open-source reference implementation of a microservice architecture. Being an open-source project, the system is accessible as a first candidate for testing non-trivial microservices systems. Additionally, we also include a microservices system from industry to show the applicability of our theory in practice.

Furthermore, we plan to compare our proof-of-concept to traditional model-based testing tooling by using the Axini Modeling Platform (AMP) [7]. AMP is a model-based testing platform used in industry on large-scale software systems. We will perform validation and evaluation experiments on our proof-of-concept and AMP to investigate how a tailored theory towards formally testing microservices impacts testing results.

4.1. Validation

We should be able to model specifications for microservices. Our proof-of-concept should be able to use these models to detect bugs in microservices systems. To validate this, we systematically inject bugs in microservices systems using open-source mutation testing frameworks such as Stryker [33] or PIT [29]. We will then test these microservices systems using our proof-of-concept and report to what extend it can detect these bugs.

4.2. Evaluation

Our proposed method touches upon two area's of software testing: testing microservices in general and model-based testing theory. We will perform evaluation experiments on both of these aspects.

We evaluate how our proof-of-concept compares to other microservices testing techniques. We have selected three testing techniques: automated testing using EvoMaster [6], browser-based end-to-end testing using Cypress [14] and manually written E2E tests without a framework. We will perform experiments by testing the same microservices system using the aforementioned techniques. We will compare these techniques based on efficiency, effectiveness and applicability using the framework proposed by Eldh et al. [16]. Additionally, we will compare the expressiveness of all testing techniques in terms of user friendliness. For example, what kind of information does each testing technique provide when a test has detected a bug? This could give us insights on how easy or intuitively a practitioner can interact with our proposed testing technique.

We also compare our proof-of-concept with the Axini Modeling Platform, as a framework representative for model-based testing. The goal is to see if a tailored theory for testing microservices differs from a general model-based testing approach. For comparison, we plan to employ a similar approach as conducted by van den Bos and Tretmans [12]. We perform multiple test executions using both model-based testing techniques. After test execution, we measure the number of inputs and outputs that were required to find the injected bug in the system under test. This approach gives us insights in how thorough and effective our proof-of-concept is in finding bugs.

5. Towards modeling microservices

Due to their distributed nature, microservices can display various behaviors that might impact modeling. In this section, we informally describe the differences of modeling a microservice.

5.1. Asynchronous Behavior

Asynchronous messaging might have an impact on model-based testing. If an output of a microservice is produced asynchronously, we might observe it at different moment for each test execution.

Effectively, this introduces a form of non-determinism where microservices that use asynchronous actions yield different traces with the same input. It might be interesting to introduce a form of partial order or relative timestamps to model this kind of asynchronous behavior of microservices. We will explore this in the short-term.

5.2. Operating environments of microservices

A microservice can be tested in isolation. Indeed, deploying a microservice separately and stubbing out any external dependency allows us to create a specific environment for testing. However, there is a difference between this testing environment and the actual production environment of microservices. Microservices communicate over a *network*. This effectively means that if we would compose microservice models, we could model this notion of the *network* as well. A network might introduce a form of non-determinism to the actions between microservices. A network connection may or may not be established between both parties. The result is that when two microservices communicate over a network to perform a procedure, this procedure may or may not be executed.

Moreover, a microservice either has successfully established a connection and is executing a procedure, or it has failed and therefore stopped. To an external observer of the microservice, there is no distinguishable behavior between the two; the microservice is *quiescent*.

When we are composing microservice models, it might be required to model the network as a separate transition system. This is inspired by the work of both de Alfaro and Helzinger [2] and Daca et al. [15]. The authors compute an environment as a separate transition system during the composition of two models. In our case, we can use this environment as a third component during composition. For example, instead of composing microservices A and B, we now compose microservices A and B with a third component (the network) N.

The main benefit is that we can use model-based testing to generate scenario's where the network between two microservices is unstable. As an example, this would allow us to test how a microservice would behave when the network is physically down.

5.3. Underspecification of the HTTP protocol

Microservices use some kind of protocol to communicate with each other. As discussed earlier, microservices typically use the HTTP protocol for synchronous communication. An interesting aspect of this protocol is that it specifies semantics for handling erroneous situations.

In the case that a microservice has handled a HTTP request successfully, it returns a HTTP Status Code in the 2xx-class. This status code class indicates success. If any kind of error occurs during the handling of a HTTP request, the microservice should return a status code in the 5xx-class. This indicates that the microservice did not fulfill the request successfully.

We could make use of the semantics of the HTTP protocol to allow the underspecification of erroneous behavior. For example, we could model a flow of a microservice that only considers the "Happy Path", e.g. only the case where the microservice returns a 2xx-class response. In this model, we did not specify any other possible HTTP Status codes, such as the 5xx-class to indicate an error. When we compose microservice specifications, we could detect that the composed specification does not contain any paths with erroneous HTTP status codes. We know that these erroneous HTTP status codes can be returned, as that is what the HTTP protocol specifies.

In this case, we can perform a special kind of *Demonic Completion* for these microservice models. For this completion, we find the states that contain transitions with HTTP responses. We can then add additional transitions from these states that model a generic error handling. This generic error handling can resemble the "Chaos" state that is used in Demonic Completion that allows all behavior.

The benefit of this approach is that we now allow microservice models with some notion of underspecification. We will investigate the compositional properties of this completion in the near-future.

6. Conclusion and Outlook

With this work-in-progress study, we want to explore the applicability of compositional model-based testing in combination with microservices.

In the short-term, we will formalize our theory regarding microservices testing. Next, we will develop a proof-of-concept that implements this tailor-made theory. This proof-of-concept will then be used to test actual microservices systems. One candidate for testing is the aforementioned open-source eShopOnContainers system. Moreover, we will also test a professionally developed microservices system that is provided by Info Support.

Additionally, we plan on performing validation and evaluation experiments on this proof-of-concept. During these experiments, both microservices systems will be tested using the proof-of-concept and the Axini Modeling Platform. This will then allow us to compare both model-based testing methods, and find out whether developing a testing theory tailored towards microservices reaps the benefits that we hypothesize.

We are aware that our endeavour may change the job portfolio of testing practitioners, however, it does not need to change the requirements on their background education. We simply aim to better leverage their knowledge of the system under test, without the burden of setting up tedious end-to-end test setups. The theoretical challenges intrinsic to the employed formalisms would have been solved in the proposed framework.

References

[1] Pekka Aho, Matias Suarez, Teemu Kanstrén, and Atif M. Memon. "Murphy Tools: Utilizing Extracted GUI Models for Industrial Software Testing". In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops.* Mar. 2014, pp. 343–348. DOI: 10.1109/ICSTW.2014.39.

[2] Luca de Alfaro and Thomas A. Henzinger. "Interface automata". In: *ACM SIGSOFT Software Engineering Notes* 26.5 (Sept. 2001), pp. 109–120. ISSN: 0163-5948. DOI: 10.1145/503271.503226. URL: https://doi.org/10.1145/503271.503226 (visited on 06/21/2022).

[3] Paul E Ammann, Paul E Black, and William Majurski. “Using model checking to generate tests from specifications”. In: *Proceedings second international conference on formal engineering methods (Cat. No. 98EX241)*. IEEE. 1998, pp. 46–54.

[4] Angular. *Protractor. End to end testing for Angular.* URL: https://www.protractortest.org/#/ (visited on 06/21/2022).

[5] Nish Anil, Clark Brent, David Coulter, Miguel Veloso, John Parente, and Maira Wenzel. *Communication in a microservice architecture.* 2021. URL: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture.

[6] Andrea Arcuri. “RESTful API Automated Test Case Generation with EvoMaster”. In: *ACM Transactions on Software Engineering and Methodology* 28.1 (Feb. 2019), pp. 1–37. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3293455. (Visited on 08/29/2022).

[7] Axini. *Axini Modeling Platform.* 2022. URL: https://www.axini.com/nl/.

[8] Fachrul Pralienka Bani Muhamad, Riyanarto Sarno, Adhatus Solichah Ahmadiyah, and Siti Rochimah. “Visual GUI testing in continuous integration environment”. In: *2016 International Conference on Information Communication Technology and Systems (ICTS).* ISSN: 2338-185X. Oct. 2016, pp. 214–219. DOI: 10.1109/ICTS.2016.7910301.

[9] L Bass, I Weber, and L Zhu. *DevOps: A Software Architect’s Perspective.* Addison-Wesley, 2015.

[10] Nikola Beneš, Przemysław Daca, Thomas A. Henzinger, Jan Křetínský, and Dejan Ničković. “Complete Composition Operators for IOCO-Testing Theory”. In: *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering*. Montréal QC Canada: ACM, May 2015, pp. 101–110. ISBN: 978-1-4503-3471-6. DOI: 10.1145/2737166.2737175. URL: https://dl.acm.org/doi/10.1145/2737166.2737175 (visited on 05/22/2022).

[11] Machiel van der Bijl, Arend Rensink, and Jan Tretmans. “Compositional Testing with ioco”. In: *Formal Approaches to Software Testing*. Ed. by Alexandre Petrenko and Andreas Ulrich. Lecture Notes in Computer Science. Springer, 2004, pp. 86–100. ISBN: 978-3-540-24617-6. DOI: 10.1007/978-3-540-24617-6_7.

[12] Petra van den Bos and Jan Tretmans. “Coverage-Based Testing with Symbolic Transition Systems”. In: *Tests and Proofs.* Ed. by Dirk Beyer and Chantal Keller. Vol. 11823. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 64–82. URL: http://link.springer.com/10.1007/978-3-030-31157-5_5 (visited on 01/13/2022).

[13] Lianping Chen. “Microservices: Architecting for Continuous Delivery and DevOps”. In: *2018 IEEE International Conference on Software Architecture (ICSA).* Seattle, WA: IEEE, Apr. 2018, pp. 39–397. ISBN: 978-1-5386-6398-1. DOI: 10.1109/ICSA.2018.00013. URL: https://ieeexplore.ieee.org/document/8417115/ (visited on 07/20/2021).

[14] Cypress. *Cypress.* URL: https://www.cypress.io/ (visited on 06/21/2022).

[15] Przemyslaw Daca, Thomas A. Henzinger, Willibald Krenn, and Dejan Nickovic. “Compositional Specifications for ioco Testing”. In: *arXiv:1904.07083 [cs]* (Apr. 2019). arXiv: 1904.07083. URL: http://arxiv.org/abs/1904.07083 (visited on 11/13/2021).

[16] Sigrid Eldh, Hans Hansson, Sasikumar Punnekkat, Anders Pettersson, and Daniel Sundmark. “A Framework for Comparing Efficiency, Effectiveness and Applicability of Software Testing Techniques”. In: Aug. 2006, pp. 159–170. DOI: 10.1109/TAIC-PART.2006.1.

[17] .NET Foundation. *eShopOnContainers*. June 2022. URL: https://github.com/dotnet-architecture/eShopOnContainers.

[18] Martin Fowler. *Microservices*. URL: https://martinfowler.com/articles/microservices.html (visited on 06/21/2022).

[19] Luca Gazzola, Maayan Goldstein, Leonardo Mariani, Itai Segall, and Luca Ussi. “Automatic Ex-Vivo Regression Testing of Microservices”. In: *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*. Seoul Republic of Korea: ACM, Oct. 2020, pp. 11–20. ISBN: 978-1-4503-7957-1. DOI: 10.1145/3387903.3389309. URL: https://dl.acm.org/doi/10.1145/3387903.3389309 (visited on 06/16/2022).

[20] A Giamas. *From Monolith to Microservices, Zalando’s Journey*. 2016. URL: https://www.infoq.com/news/2016/02/Monolith-Microservices-Zalando/.

[21] Robert Heinrich, André van Hoorn, Holger Knoche, Fei Li, Lucy Ellen Lwakatare, Claus Pahl, Stefan Schulte, and Johannes Wettinger. “Performance Engineering for Microservices: Research Challenges and Directions”. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. L’Aquila Italy: ACM, Apr. 2017, pp. 223–226. ISBN: 978-1-4503-4899-7. DOI: 10.1145/3053600.3053653. URL: https://dl.acm.org/doi/10.1145/3053600.3053653 (visited on 07/20/2021).

[22] Robert M Hierons, Kirill Bogdanov, Jonathan P Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, et al. “Using formal specifications to support testing”. In: *ACM Computing Surveys (CSUR)* 41.2 (2009), pp. 1–76.

[23] T Hoff. *Amazon Architecture*. 2007. URL: http://highscalability.com/blog/2007/9/18/amazon-architecture.html (visited on 03/31/2022).

[24] Yury Izrailevsky, Stevan Vlaovic, and Ruslan Meshenberg. *About Netflix - Completing the Netflix Cloud Migration*. 2016. URL: https://about.netflix.com/,%20https://about.netflix.com/en/news/completing-the-netflix-cloud-migration.

[25] Ramon Janssen and Jan Tretmans. “Matching implementations to specifications: the corner cases of ioco”. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. Limassol Cyprus: ACM, Apr. 2019, pp. 2196–2205. ISBN: 978-1-4503-5933-7. DOI: 10.1145/3297280.3297496. URL: https://dl.acm.org/doi/10.1145/3297280.3297496 (visited on 06/21/2022).

[26] Qing Lei, Weidong Liao, Yingtao Jiang, Mei Yang, and Haifeng Li. “Performance and Scalability Testing Strategy Based on Kubemark”. In: *2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*. Chengdu, China: IEEE, Apr. 2019, pp. 511–516. URL: https://ieeexplore.ieee.org/document/8725658/ (visited on 07/20/2021).

[27] Leonardo Mariani, Mauro Pezze, Oliviero Riganelli, and Mauro Santoro. "AutoBlackTest: Automatic Black-Box Testing of Interactive Applications". In: *Verification and Validation 2012 IEEE Fifth International Conference on Software Testing*. ISSN: 2159-4848. Apr. 2012, pp. 81–90. DOI: 10.1109/ICST.2012.88.

[28] A.M. Memon, M.E. Pollack, and M.L. Soffa. "Hierarchical GUI test case generation using automated planning". In: *IEEE Transactions on Software Engineering* 27.2 (Feb. 2001). Conference Name: IEEE Transactions on Software Engineering, pp. 144–155. ISSN: 1939-3520. DOI: 10.1109/32.908959.

[29] PIT. *PIT Mutation Testing*. URL: https://pitest.org/ (visited on 09/05/2022).

[30] Jose G. Quenum and Samir Aknine. "Towards Executable Specifications for Microservices". In: *2018 IEEE International Conference on Services Computing (SCC)*. San Francisco, CA, USA: IEEE, July 2018, pp. 41–48. ISBN: 978-1-5386-7250-1. DOI: 10.1109/SCC.2018.00013. URL: https://ieeexplore.ieee.org/document/8456399/ (visited on 07/20/2021).

[31] Chris Richardson. *Building Microservices: Inter-Process Communication*. July 2015. URL: https://www.nginx.com/blog/building-microservices-inter-process-communication/ (visited on 02/03/2022).

[32] SmartBear. *Cucumber*. URL: https://cucumber.io/ (visited on 09/05/2022).

[33] Stryker. *Stryker Mutator*. URL: https://stryker-mutator.io/.

[34] TorXakis. *TorXakis*. URL: https://github.com/TorXakis/TorXakis (visited on 06/21/2022).

[35] Jan Tretmans. "Model Based Testing with Labelled Transition Systems". In: *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*. Ed. by Robert M. Hierons, Jonathan P. Bowen, and Mark Harman. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 1–38. ISBN: 978-3-540-78917-8. DOI: 10.1007/978-3-540-78917-8_1. URL: https://doi.org/10.1007/978-3-540-78917-8_1 (visited on 07/12/2021).

[36] Tricentis. *SpecFlow*. URL: https://specflow.org/ (visited on 09/05/2022).

[37] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Elsevier, July 2010. ISBN: 978-0-08-046648-4.

[38] Mark Utting, Alexander Pretschner, and Bruno Legeard. "A TAXONOMY OF MODEL-BASED TESTING". In: (Apr. 2006), p. 18.