Compositional ioco using model-based mocking

Jore J. Booy[1], Jeroen J.A. Keiren[1] and Machiel van der Bijl[2]

[1]Eindhoven University of Technology, Groene Loper 3, 5612 AE, Eindhoven, The Netherlands
[2]Axini, Van Boshuizenstraat 12, 1083 BA, Amsterdam, The Netherlands

Abstract

Model-based testing is a compelling method for the end-to-end testing of microservices. However, when testing with a large number of services, state space explosion is a common problem. It is especially a problem since input-output conformance (**ioco**) is not compositional. We developed a novel and theoretically grounded testing method called model-based mocking (MBM) to end-to-end test microservice systems compositionally. We tested the MBM method using the Axini Modeling platform by inserting 20 mutants into an example microservice system. In our set of inserted bugs, MBM found more than half of the bugs faster compared to other methods and was slower for none of the bugs.

Keywords

Mobel-based testing, ioco, microservices, Axini Modeling Platform, model-based mocking

1. Introduction

Software testing takes up a significant amount of time in the development process. In practice, it is still primarily a manual effort. Model-based testing (MBT) is an automated approach that can be used to systematically and automatically test that an implementation of a system conforms to its specification. MBT automatically generates tests from a formal specification of the system under test (SUT), foregoing manually constructing, executing and maintaining test cases.

Input-output conformance (**ioco**) testing [1], is a commonly used MBT technique. This assumes that the SUT can be modeled using *labelled transition systems* (LTSs) with inputs and outputs. MBT tools typically support describing the specification using *symbolic transition systems* instead of LTSs. Implementations of **ioco**-testing range from academic tools such as JTorX [2] and TorXakis[1] to commercially supported tools such as the Axini Modelling Platform (AMP),[2] which was used, e.g., for testing railway signaling systems at Prorail [3] as well as Internet of Things protocol implementations [4]. A proof of concept shows that **ioco**-testing is effective at detecting unspecified behavior in microservices [5].

In practice, testing using **ioco** comes with some challenges. Testing the parallel composition of specifications is not desired, as it can cause exponential increase in testing time due to the increase in state space, and can cause memory issues. This is often the case with the popular architectural style of microservices. As microservices have well-defined APIs, in principle they

BENEVOL 2023: The 22nd Belgium-Netherlands Software Evolution Workshop, Nijmegen, 27–28 November 2023
hi@jore.dev (J. J. Booy); j.j.a.keiren@tue.nl (J. J.A. Keiren); machiel.van.der.bijl@axini.com (M. v. d. Bijl)
0000-0002-5772-9527 (J. J.A. Keiren)

[1]https://github.com/torxakis
[2]https://www.axini.com/nl/

should be amenable to MBT: one can describe the entire system by modeling the individual services, and models can be combined to describe the entire system. However, since there is often a large number of services, state space explosions are common.

To avoid state space explosions, it is desired to test systems individually. However, **ioco**-testing is not compositional: even though individual SUTs may conform to their specification, their parallel composition may not conform to the parallel composition of the specifications. Some solutions have been proposed. Input-enabled specifications are for instance compositional, and input-enabled specifications can be using *demonic completion* [6]. Alternatively, the set of traces used in the definition of conformance can be reduced, leading to **uioco** [6]. Environmental testing tests if top-level services use their components correctly according to their STS specification [7]. Also, if there are no *ambiguous states* in the parallel composition of the model, **ioco** is still compositional [8]. A similar idea, *mutual acceptance*, was studied in **uioco** [9].

Contributions. In this paper, we develop a theoretical approach called model-based mocking (MBM) for testing microservices using **ioco** with on-the-fly ambiguous state detection. In particular, we show that when testing a specific microservice, the services that are used can be replaced by a mock (generated by the model), which simulates the implementation that adheres to the specification. In theory, this reduces communication and processing delays in the testing process. It is not required for the specifications to be input-enabled. Rather, it is sufficient to check for (absence of) ambiguous states at runtime. We evaluate the theory by implementing the technique in AMP, and testing 20 different mutants of a microservice. We show that MBM often outperforms existing testing approaches.

2. Preliminaries

Input-output conformance (**ioco**) is a theory for MBT based on labelled transition systems (LTSs) [1]. In this setting, the labels are separated into input and output labels.

Definition 2.1. *An* input-output labeled transition system *(IOLTS) is a 5-tuple* $\langle Q, L^I, L^U, T, q_0 \rangle$ *where* Q *is a countable, non-empty set of states;* L^I *and* L^U *are countable sets of input and output labels, such that* $L = L^I \cup L^U$*, and* $L^I \cap L^U = \emptyset$*. We write* $L_\tau = L \cup \{\tau\}$*, where* $\tau \notin L$ *is the* internal action. $T \subseteq Q \times L_\tau \times Q$ *is the transition relation, and* $q_0 \in Q$ *is the initial state.*

We use the following notation. Let $q, q' \in Q$, $\mu \in L_\tau$, $a, a_i \in L$ and $\sigma \in L^*$. We write $q \xrightarrow{\mu} q'$ for $(q, \mu, q') \in T$, $q \xrightarrow{\mu}$ if $q \xrightarrow{\mu} q'$ for some q', and $q \not\xrightarrow{\mu}$ if $\neg(q \xrightarrow{\mu})$. We generalize the transition relation to weak transitions in the standard way, that is, we write $q \overset{\epsilon}{\Rightarrow} q'$ if there is a (possibly empty) sequence of τ-transitions from q to q'; we write $q \overset{a}{\Rightarrow} q'$ if $q \overset{\epsilon}{\Rightarrow} q_1 \xrightarrow{a} q_2 \overset{\epsilon}{\Rightarrow} q'$ for some q_1, q_2. We generalize this to weak traces by writing $q \overset{a_1 \cdot \ldots \cdot a_n}{\Longrightarrow} q'$ if $\exists q_1, \cdots, q_n : q = q_1 \overset{a_1}{\Longrightarrow} \cdots \overset{a_{n-1}}{\Longrightarrow} q_n = q'$, and $q \overset{\sigma}{\Rightarrow}$ if there exists q' such that $q \overset{\sigma}{\Rightarrow} q'$.

Definition 2.2. *An IOLTS* $\langle Q, L^I, L^U, T, q_0 \rangle$ *is* input-enabled *iff* $\forall q \in Q, a \in L^I : q \overset{a}{\Rightarrow}$*. An input-enabled IOLTS is referred to as an* input-output transition system *(IOTS).*

A state from which no output can be produced is called *quiescent*.

Definition 2.3. *Let s be an IOLTS $\langle Q, L^I, L^U, T, q_0\rangle$ and $\sigma \in L^*$ a trace of s. A state $q \in Q$ is* quiescent *if for all $\mu \in L^U \cup \{\tau\}$, $q \overset{\mu}{\not\rightarrow}$. We write $\delta(q)$ if q is quiescent. We can make the observation of quiesence explicit by extending our IOLTS. We write $\Delta(s) = \langle Q, L^I, L^U \cup \{\delta\}, \Delta(T), q_0\rangle$ for this IOLTS, where $\Delta(T) = T \cup \{(q, \delta, q) \mid q \in Q \wedge \delta(q)\}$. We write $\rightarrow_\Delta$ and $\Rightarrow_\Delta$ when we explictly refer to this transition relation or the corresponding weak transition relation, and use L_δ for $L \cup \{\delta\}$. The* suspension traces *of s are $Straces(s) = \{\sigma \in L_\delta^* \mid q_0 \overset{\sigma}{\Rightarrow}_\Delta\}$.*

In the definition of **ioco**, we further use the following notation. The set of states in which the system can be after executing σ is denoted q **after** $\sigma = \{q' \mid q \overset{\sigma}{\Rightarrow}_\Delta q'\}$. The outputs that are enabled in state q are $out(q) = \{a \in L^U \mid q \overset{a}{\Rightarrow}_\Delta\}$. We use s **after** σ to denote q_0 **after** σ.

Definition 2.4. *Let i be an IOTS and s an IOLTS with input labels L^I and output labels L^U. Then i* **ioco** *$s = \forall \sigma \in Straces(s) : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$.*

Parallel composition ($\|$) is used to define how actions are ordered between two processes. Given two IOLTS i_1 and i_2, then the set of actions must synchronize (i.e. happen at the same time) is $G = (L_1^I \cap L_2^U) \cup (L_1^U \cap L_2^I)$. Then all other actions $(L_1 \cup L_2) \setminus G$ are interleaved.

Definition 2.5. *For $i = 1, 2$ let s_i be IOLTS $\langle Q_i, L_i^I, L_i^U, T_i, q^i\rangle$. s_1 and s_2 are* composable *iff $L_1^I \cap L_2^I = L_1^U \cap L_2^U = \emptyset$. If s_1 and s_2 are composable, then $s_1 \parallel s_2 = \langle Q, L^I, L^U, T, q^1 \parallel q^2\rangle$ where $Q = \{(q_1, q_2) \mid q_1 \in Q_1, q_2 \in Q_2\}$, $L^I = (L_1^I \setminus L_2^U) \cup (L_2^I \setminus L_1^U)$, $L^U = L_1^U \cup L_2^U$, and T is the minimal set satisfying the following inference rules (where $\mu \in L_\tau$):*

$$\frac{q_1 \overset{\mu}{\rightarrow} q_1', \mu \notin L_2}{q_1 \parallel q_2 \overset{\mu}{\rightarrow} q_1' \parallel q_2} \qquad \frac{q_2 \overset{\mu}{\rightarrow} q_2', \mu \notin L_1}{q_1 \parallel q_2 \overset{\mu}{\rightarrow} q_1 \parallel q_2'} \qquad \frac{q_1 \overset{\mu}{\rightarrow} q_1', q_2 \overset{\mu}{\rightarrow} q_2', \mu \neq \tau}{q_1 \parallel q_2 \overset{\mu}{\rightarrow} q_1' \parallel q_2'}$$

Note that, in this definition, if s_1 and s_2 synchronize on action a, a is an output of $s_1 \parallel s_2$.

3. Compositional ioco-testing

Modern systems are often composed of multiple components running in parallel. Ideally, we want to be able to test the individual components, as this allows for simpler descriptions, and alleviates the state space explosion problem, while still being able to draw conclusions about the system as a whole. However, in general, **ioco** is not compositional: if implementation i_1 **ioco** s_1 and i_2 **ioco** s_2, it is not generally the case that $i_1 \parallel i_2$ **ioco** $s_1 \parallel s_2$ [6].

Different solutions have been proposed to allow for compositional testing. For instance, when all specifications are input-enabled, **ioco** is compositional [6]. Alternatively, the weaker relation **uioco** can be used for compositional testing. This corresponds to first making the specification input-enabled using demonic completion, and subsequently applying the standard **ioco**-relation [6, 9]. As **ioco** is a stronger relation, and AMP is built on it, we instead investigate compositionality in the setting of **ioco**. Daca *et al.* [8] proposed a solution that avoids *ambiguous states*. These are states in the parallel composition where one component wants to do an output, but the other component is not ready to do the corresponding input.

Definition 3.1. *Let s_1, s_2 be composable IOLTSs, and let $s_i = \langle Q_i, L_i^I, L_i^U, T_i, q_0^i \rangle$. A pair $(q_1, q_2) \in Q_1 \times Q_2$ is an* ambiguous state *if there exists a shared action $a \in L_1^I \cap L_2^U$ such that $q_2 \xrightarrow{a}$ and $q_1 \not\xrightarrow{a}$, or $a \in L_2^I \cap L_1^U$ such that $q_1 \xrightarrow{a}$ and $q_2 \not\xrightarrow{a}$.*

For specifications without ambiguous states, **ioco** is compositional [8]. However, detecting ambiguous states requires building the parallel composition of the specifications before testing, and if there are ambiguous states, the specifications need to be updated to remove those.

In this paper, we do not perform direct compositional testing. Instead, we use the results from Daca *et al.* to show that we can replace implementations that we do not want to test with their specification. This is formalized using the following result.

Theorem 3.2 (ioco-substitution for an ambiguous-free interaction**).** *Let i_1, i_2 be IOTSs and s_1, s_2 be IOLTSs, with respective input and output labels (L_1^I, L_1^U) and (L_2^I, L_2^U). Let s_2 be input-enabled for labels $L_2^I \setminus L_1^U$ and $i_1 \parallel s_2$ contain no ambiguous states.*

$$(i_2 \mathbf{ioco} s_2) \wedge (i_1 \parallel s_2 \mathbf{ioco} s_1 \parallel s_2) \Rightarrow i_1 \parallel i_2 \mathbf{ioco} s_1 \parallel s_2$$

Note that this requires input-enabledness only for $L_2^I \setminus L_1^U$, which weakens the requirements needed to get compositionality for **ioco** in general. Now s_2 only needs to be input-enabled for actions that are not part of the communication with i_1. In practice, this is a useful requirement: if we can restrict s_2 such that it only describes the interface-behaviour needed to communicate with s_1, it does not have any other inputs. This is, e.g., typically the case in a client-server or module-submodule context.

4. Model-Based Mocking

Traditionally, in MBT we test whether an implementation conforms to its specification by generating inputs to the SUT based on the specification, and observing the outputs of the SUT and checking these are allowed by the specification. This requires an adapter to translate between actions in the specification and calls in the implementation.

We use the results from the previous section to obtain a testing setup using *model-based mocking* (MBM). The idea is effectively to replace the systems we are not testing with correct implementations, so called mocks, that are inferred from the specifications. However, we want to avoid manually developing the mocks. Instead, we use the specifications and existing adapters of the mocked components to allow the testing platform to act as mock. For this, AMP listens to the communication between the SUT and the mocked services, uses the adapter to intercept the calls from the SUT to the mocked service, translates the call to an action in the specification of the mocked service, generates a response based on the specification, and translates it back to a call that can be forwarded to the SUT. The approach is sketched in Figure 1b. This way, assuming the services that the SUT depends on are correct, MBM allows us to isolate the SUT, and focus the testing effort on it.

Note that our theory requires that $i_1 \parallel s_2$ does not contain ambiguous states. In our implementation, we handle this during testing. In practice, i_1 will not wait for s_2 to be ready to receive the message. If s_2 receives an unexpected message, this is reported to the testing platform, and the test fails: either s_2 is incomplete or we have detected a bug in the SUT.

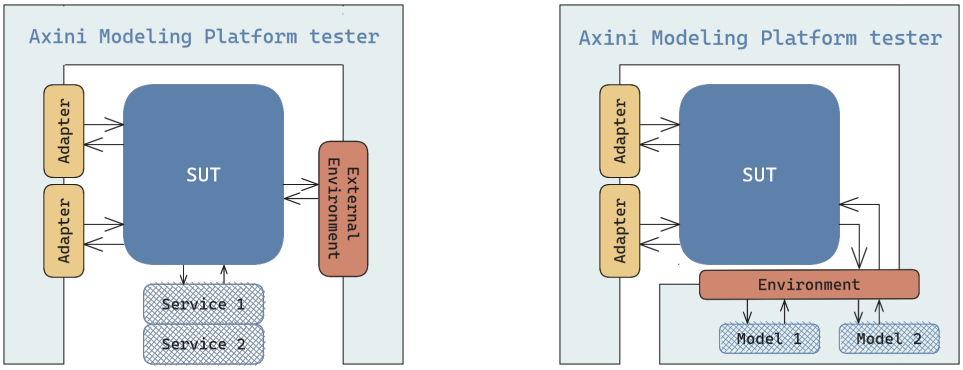


Figure 1: The old way and the new way AMP can test.

The existing approach to isolated SUT testing in AMP is shown in Figure 1a. The tester considers transitions in the specification of the SUT. If the tester triggers an an input in some service, AMP sends the corresponding action label to an adapter, that translates it into either an HTTP or AMQP message for the relevant component. Messages that are received in return, such as an HTTP response, are received, and translated by the adapter into an action in the specification. The tester than takes the corresponding transition in the specification. If the response does not match the specification, the test fails, otherwise testing can continue. Internal communication is not listened to or interacted with.

MBM allows for a different setup shown in Figure 1b. The services that are used by the SUT are modelled in AMP. Calls that are made to these services from SUT are intercepted, and the existing adapter is used to translate messages into actions. These actions are matched to the specification of the service, and an appropriate response is selected based on the service specification. The response is again translated into a message using the adapter, and sent back to the SUT. When a synchronizing transition happens between a mocked service and the SUT, the transition is handled as if it is a external transition from the perspective of the SUT. Internal transitions between mocked services are considered hidden if the original transition was hidden, otherwise it is a regular mocked output. We do not require *a priori* checking of ambiguous states. Instead, ambiguous states between the SUT and the mock can be detected at runtime.

5. Experimental Evaluation

To study the effect of mocking on testing performance, we compare three testing methods. In *naive parallel composition* we first compute the parallel composition of all specifications, and use this specification to test all implementations together. To fully compare the testing methods, the specification for this method is precomputed, and added to AMP. *Simultaneous testing* is AMP's default testing approach. Here all implementations are tested together, however, the parallel composition of the specifications is not precomputed. Instead, the current state of all the specifications is tracked on-the-fly. This should have the same testing effectiveness as naive

composition. *Model-based mocking* is our new approach. It tests a single service; the remaining services are mocked.

As system under test (SUT) we use the eShopOnContainers project[3]. This is an open source web shop built using .NET microservices. We model four services of the project: the basket service, the catalog service, the ordering service and the payment service. When a user checks out in the web shop, all services will be part of the processing of the checkout. The specification model was derived from van den Brink's model [5] and the .NET implementation.

The basket API is the service the end user interacts with the most. The user triggers the checkout from it, after which the rest of the services interact to finish the checkout. The basket API is the service we want to test. For MBM, we mock all services, except the basket. To compare the different testing methods, we consider 20 different mutants of the basket service, and compare the time it takes for each testing method to find the bug in the mutant. The mutants are those originally generated by van den Brink [5]. These mutants for instance change an equals sign to a not equals sign, or return a different status code. For naive parallel composition, the measured time does not include the time needed to build the composition.

All tests have been run on a HP ZBook Studio G4 laptop with Intel Core i7-7700HQ CPU, 16GB of RAM and an NVIDIA Quadro M1200. All implementations of services were run on this same machine. To introduce latency, we use the `tc` tool to simulate a fixed round-trip latency of 100ms. We used a timeout of 500 steps. Each of the test runs is repeated 10 times.

5.1. Results

Each of the testing approaches was able to find 19 out of 20 mutants. Mutant 11 was not detected by any of the approaches. This is consistent with the findings from [5]. Closer inspection of the implementation shows that the mutant is likely to be unreachable. We therefore do not consider it further in the results, and we only report on 19 mutants.

The models we use are complex, and we cannot assume a particular distribution of the data. Also, the data for different experiments are not independent. To statistically compare the testing methods, we therefore used the Kolmogorov-Smirnov test. The test compares the cumulative distribution function (cdf) of two distributions to see if one is significantly bigger or smaller at different points in the distribution. The cdf of our tests is calculated using the percentage of tests that has completed at a specific time, for instance after 30 seconds. If the cdf is significantly greater at any point in time than the other cdf, the testing method is a faster testing method.

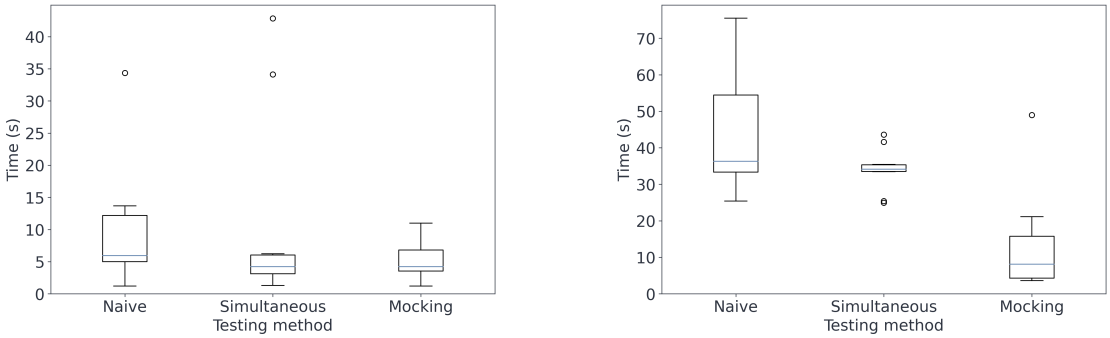
The results are summarized in Table 1. The table shows in how many instances the test on the left was significantly faster than the test on the right, for $p < 0.05$ and $p < 0.10$. For example, the last row in the table shows that mocking was significantly faster than simultaneous testing in 6 out of 19 cases ($p < 0.05$), or 8 out of 19 cases (when $p < 0.10$). The last column shows that mocking was never significantly slower than parallel or simultaneous testing.

If we look at the differences between the methods for individual mutants, we observe that the tests using mocking often has fewer extreme outliers in the test. This can for instance be seen in the results for mutant 1 in Figure 2a. For mutant 13, whose detection requires interaction with the mocked services, mocking is significantly faster than simultaneous testing (both $p = 1.082 \times 10^{-4}$), see Figure 2b.

[3] https://github.com/dotnet-architecture/eShopOnContainers

Table 1
Number of mutants (out of 19) for which one approach is significantly faster than the other. Indicated as $n_1 \mid n_2$, where n_1 is for $p < 0.05$, n_2 is for $p < 0.10$.

	Naive	Simultaneous	Mocking
Naive	×	$0 \mid 0$	$0 \mid 0$
Simultaneous	$2 \mid 9$	×	$0 \mid 0$
Mocking	$10 \mid 12$	$6 \mid 8$	×



(a) Mutant 1

(b) Mutant 13

Figure 2: Some example mutant results.

5.2. Discussion

From the results we observe that mocking is significantly faster than naive parallel testing most of the time, and it is significantly faster than simultaneous testing in about one third of the cases. Simultaneous testing is sometimes faster than naive parallel composition.

We have two reasons to think simultaneous testing was sometimes faster than naive parallel composition. First, higher memory consumption required by explicitly building the parallel composition can cause a performance decrease during testing. This is because garbage collection has to occur more often for the .NET applications, interrupting the regular flow of the program. Second, testing using parallel composition might require visiting different interleavings that represent equivalent paths. Simultaneous testing does not visit these different interleavings.

Regarding mocking, we should note that the mutants were not chosen specifically to require that they are on a path that includes mocked services. In fact, only mutant 13 requires interacting with mocked services to find the bug. Our results show that in particular in case of mutant 13, mocking is much faster than simultaneous testing. We expect that this is mainly due to lower communication and processing delays in the mock compared to the original simulation. We expect a similar performance improvement for other test cases that require mocked services.

All mutants except mutant 13 can be detected by the model following only transitions from the basket service. However, even in those cases, mocking is often faster. We expect that this is again due to the mocked services having lower communication delays. Even though the services are not critical in finding the bug, during testing they might still be used.

6. Concluding remarks

We have developed model-based mocking (MBM). This is a strategy to test microservice architectures in AMP using on **ioco**-testing. We show that if the parallel composition of a mock and implementation does not have ambiguous states, the implementations of services can be replaced by a mock based on their specification. The approach was implemented in AMP. Our experiments show that MBM can speed up **ioco**-testing when targeting individual services, even in cases where the bug could be detected without making use of the mocked services. Thus, MBM provides a faster way to test individual components, guaranteeing that the composition is conformant given that its components are.

For future work, we want to test mocking on additional SUTs to see if our results generalize to different systems. As a consequence of the limitations of composability on input labels of parallel components, our approach does not allow for broadcast messaging. Alternative notions of composability, allowing for the same input labels, and mutual acceptance were defined in the setting for **uioco** [9]. It would be interesting to see if our approach also works in that context.

References

[1] J. Tretmans, Test generation with inputs, outputs, and quiescence, in: Tools and Algorithms for the Construction and Analysis of Systems, volume 1055, Springer, Berlin, Heidelberg, 1996, pp. 127–146. doi:10.1007/3-540-61042-1_42.

[2] A. Belinfante, JTorX: Exploring Model-Based Testing (2014). doi:10.3990/1.9789036537070.

[3] T. Bachmann, D. van der Wal, M. van der Bijl, D. van der Meij, A. Oprescu, Translating EULYNX SysML Models into Symbolic Transition Systems for Model-Based Testing of Railway Signaling Systems, in: 2022 IEEE Conference on Software Testing, Verification and Validation (ICST), 2022, pp. 355–364. doi:10.1109/ICST53961.2022.00044.

[4] X. M. van Dommelen, M. van der Bijl, A. Pimentel, Model-Based Testing of Internet of Things Protocols, in: Formal Methods for Industrial Critical Systems, LNCS, Springer International Publishing, Cham, 2022, pp. 172–189. doi:10.1007/978-3-031-15008-1_12.

[5] B. van den Brink, Towards Model-Based Testing for Microservices, Master's thesis, University of Amsterdam, 2023. URL: https://scripties.uba.uva.nl/search?id=record_53331.

[6] H. M. van der Bijl, On Changing Models in Model-Based Testing, Ph.D. thesis, University of Twente, Enschede, The Netherlands, 2011. doi:10.3990/1.9789036531955.

[7] L. Frantzen, J. Tretmans, Model-based testing of environmental conformance of components, in: Formal Methods for Components and Objects, volume 4709, Springer, Berlin, Heidelberg, 2007, pp. 1–25. doi:10.1007/978-3-540-74792-5_1.

[8] P. Daca, T. A. Henzinger, W. Krenn, D. Nickovic, Compositional specifications for ioco testing, in: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, IEEE, USA, 2014, pp. 373–382. doi:10.1109/ICST.2014.50.

[9] G. van Cuyck, L. van Arragon, J. Tretmans, Compositionality in Model-Based Testing, in: Testing Software and Systems, LNCS, Springer, Cham, 2023, pp. 202–218. doi:10.1007/978-3-031-43240-8_13.