LANGUAGE-BASED INTERACTIVE TESTING

Yishuai Li

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2022

Supervisor of Dissertation

Benjamin C. Pierce
Professor of Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer and Information Science

Dissertation Commitee

Steve Zdancewic, Professor of Computer and Information Science, Chair
Mayur Naik, Professor of Computer and Information Science
Boon Thau Loo, Professor of Computer and Information Science
John Hughes, Professor of Computing Science, Chalmers University of Technology

LANGUAGE-BASED INTERACTIVE TESTING

COPYRIGHT

2022

Yishuai Li

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) License

To view a copy of this license, visit

`https://creativecommons.org/licenses/by-sa/4.0/`

Acknowledgments

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

ABSTRACT

LANGUAGE-BASED INTERACTIVE TESTING

Yishuai Li
Benjamin C. Pierce

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Contents

List of Figures

CHAPTER 1

Introduction

The security and robustness of networked systems rest in large part on the correct behavior of various sorts of servers. This can be validated either by full-blown verification or model checking against formal specifications, or less expensively by rigorous testing.

Rigorous testing requires a rigorous specification of the protocol that we expect the server to obey. Protocol specifications can be written as (i) a *server model* that describes *how* valid servers should handle messages, or (ii) a *property* that defines *what* server behaviors are valid. From these specifications, we can conduct (i) *model-based testing* [4] or (ii) *property-based testing* [7], respectively.

When testing server implementations against protocol specifications, one critical challenge is *nondeterminism*, which arises in two forms—we call them (1) *internal nondeterminism* and (2) *network nondeterminism*:

(1) *Within* the server, correct behavior may be underspecified. For example, to handle HTTP conditional requests [6], a server generates strings called entity tags (ETags), but the RFC specification does not limit what values these ETags should be. Thus, to create test messages containing ETags, the tester must remember and reuse the ETags it has been given in previous messages from the server.

(2) *Beyond* the server, messages and responses between the server and different clients might be delayed and reordered by the network and operating-system buffering. If the tester cannot control how the execution environment reorders messages—*e.g.,* when testing over the Internet—it needs to specify what servers are valid as observed over the network.

These sources of nondeterminism pose challenges in various aspects of testing network protocols: (i) The *validation logic* should accept various implementations, as long as the behavior is included in the specification's space of uncertainties; (ii) To capture bugs effectively, the *test harness* should generate test cases based on runtime observations; (iii) When *shrinking* a counterexample, the test harness should adjust the test cases based on the server's behavior, which might vary from one execution to another.

To address these challenges, I introduce symbolic languages for writing specifications and representing test cases:

(i) The specification is written as a reference implementation—a nondeterministic program that exhibits all possible behavior allowed by the protocol. Inter-implementation and inter-execution uncertainties are represented by symbolic variables, and the space of nondeterministic behavior is defined by all possible assignments of the variables.

The validation logic is derived from the reference implementation, by *dualising* the server-side program into a client-side observer.

(ii) Test generation heuristics are defined as computations from the observed trace (list of sent and received messages) to the next message to send. I introduce a symbolic intermediate representation for specifying the relation between the next message and previous messages.

(iii) The symbolic language for generating test cases also enables effective shrinking of test cases. The test harness minimizes the counterexample by shrinking its symbolic representation. When running the test with a shrunk input, the symbolic representations can be re-instantiated into request messages that reflect the original heuristics.

Thesis claim. Symbolic abstract representation can address challenges in testing networked systems with uncertain behavior. Specifying protocols with symbolic reference implementation enables validating the system's behavior systematically. Representing test input as abstract messages allows generating and shrinking interesting test cases. Combining these methods result in a rigorous tester that can capture protocol violations effectively.

This thesis is structured as follows:

CHAPTER 2

Challenges

The Deep Specification project [2] aims at building a web server and guarantee its functional correctness with respect to formal specification of the network protocol.

HTTP/1.1 requests can be conditional: if the client has a local copy of some resource and the copy on the server has not changed, then the server needn't resend the resource. To achieve this, an HTTP/1.1 server may generate a short string, called an "entity tag" (ETag), identifying the content of some resource, and send it to the client:

```
/* Client: */
GET /target HTTP/1.1

/* Server: */
HTTP/1.1 200 OK
ETag: "tag-foo"
... content of /target ...
```

The next time the client requests the same resource, it can include the ETag in the GET request, informing the server not to send the content if its ETag still matches:

```
/* Client: */
GET /target HTTP/1.1
If-None-Match: "tag-foo"

/* Server: */
HTTP/1.1 304 Not Modified
```

If the tag does not match, the server responds with code 200 and the updated content as usual. Similarly, if a client wants to modify the server's resource atomically by compare-and-swap, it can include the ETag in the PUT request as `If-Match` precondition, which instructs the server to only update the content if its current ETag matches.

Thus, whether a server's response should be judged *valid* or not depends on the ETag it generated when creating the resource. If the tester doesn't know the server's internal state (*e.g.*, before receiving any 200 response including the ETag), and cannot enumerate all of them (as ETags can be arbitrary strings), then it needs to maintain a space of all possible values, narrowing the space upon further interactions with the server.

It is possible, but tricky, to write an ad hoc tester for HTTP/1.1 by manually "dualizing" the behaviors described by the informal specification documents (RFCs). The protocol document describes *how* a valid server should handle requests, while

```
(* update : (K → V) * K * V → (K → V) *)
let check (trace  : stream http_message,
           data   : key → value,
           is     : key → etag,
           is_not : key → list etag) =
  match trace with
  | PUT(k,t,v) :: SUCCESSFUL :: tr' =>
    if t ∈ is_not[k] then reject
    else if   is[k] == unknown
            ∨ strong_match(is[k],t)
         then let d' = update(data,k,v)     in
              let i' = update(is,k,unknown) in
              let n' = update(is_not,k,[])  in
       (* Now the tester knows that
        * the data in [k] is updated to [v],
        * but its new ETag is unknown. *)
              check(tr',d',i',n')
         else reject
  | PUT(k,t,v) :: PRECONDITION_FAILED :: tr' =>
    if strong_match(is[k],t) then reject
    else let n' = update(is_not, k, t::is_not[k])
      (* Now the tester knows that
       * the ETag of [k] is other than [t]. *)
         in check(tr',data,is,n')
  | GET(k,t) :: NOT_MODIFIED :: tr' =>
    if t ∈ is_not[k] then reject
    else if is[k] == unknown ∨ weak_match(is[k],t)
         then let i' = update(is,k,t) in
       (* Now the tester knows that
        * the ETag of [k] is equal to [t]. *)
              check(tr',data,i',is_not)
         else reject
  | GET(k,t0) :: OK(t,v) :: tr' =>
    if weak_match(is[k],t0) then reject
    else if data[k] ≠ unknown ∧ data[k] ≠ v
         then reject
         else let d' = update(data,k,v) in
              let i' = update(is,  k,t) in
       (* Now the tester knows
        * the data and ETag of [k]. *)
              check(tr',d',i',is_not)
  | _ :: _ :: _  => reject
  end
```

FIGURE 2.1. Ad hoc tester for HTTP/1.1 conditional requests, demonstrating how tricky it is to write the logic by hand. The checker determines whether a one-client-at-a-time `trace` is valid or not. The trace is represented as a stream (infinite linked list, constructed by "`::`") of HTTP messages sent and received. `PUT(k,t,v)` represents a PUT request that changes `k`'s value into `v` only if its ETag matches `t`;

the tester needs to determine *what* responses received from the server are valid. For example, “If the server has revealed some resource’s ETag as `"foo"`, then it must not reject requests targetting this resource conditioned over `If-Match: "foo"`, until the resource has been modified”; and “Had the server previously rejected an `If-Match` request, it must reject the same request until its target has been modified.” Figure 2.1 shows a hand-written tester for checking this bit of ETag functionality; we hope the reader will agree that this testing logic is not straightforward to derive from the informal “server’s eye” specifications.

Networked systems are naturally concurrent, as a server can be connected with multiple clients. The network might delay packets indefinitely, so messages sent via different channels may be reordered during transmission. When the tester observes messages sent and received on the client side, it should allow all observations that can be explained by the combination of a valid server + a reasonable network environment between the server and clients.

CHAPTER 3

Prior Practices in Interactive Testing

3.1. Specification Languages

3.1.1. Property-based specification with QuickChick. My first formal specification of HTTP/1.1 was written as QuickChick [10] properties, which takes a trace of requests, and determines whether the traces is valid per protocol specification, like that shown in Figure 2.1. The specification implemented a constraint solving logic by hand, making it hard to scale when the protocol becomes more complex, as discussed in **??**

3.1.2. Model-based specification with ITrees. To write specifications for protocols' rich semantics, I employed "interaction tree" (ITree), a generic data structure for representing interactive programs, introduced by Xia et al. [14]. ITree enables specifying protocols as monadic programs that model valid implementations' possible behavior. The model program can be interpreted into a tester program, to be discussed in Section 3.2.

Figure 3.1 defines the type `itree E R`. The definition is *coinductive*, so that it can represent potentially infinite sequences of interactions, as well as divergent behaviors. The parameter `E` is a type of *external interactions*—it defines the interface by which a computation interacts with its environment. `R` is the *result* of the computation: if the computation halts, it returns a value of type `R`.

```
CoInductive itree (E : Type → Type) (R : Type) :=
| Ret (r : R)
| Vis {X : Type} (e : E X) (k : X → itree E R)
| Tau (t : itree E R).

Inductive event (E : Type → Type) : Type :=
| Event : forall X, E X → X → event E.

Definition trace E := list (event E)

Inductive is_trace E R
  : itree E R → trace E → Prop := ...
  (* straightforward definition omitted *)
```

FIGURE 3.1. Interaction trees and their traces of events.

There are three ways to construct an ITree. The `Ret r` constructor corresponds to the trivial computation that halts and yields the value `r`. The `Tau t` constructor corresponds to a silent step of computation, which does something internal that does not produce any visible effect and then continues as `t`. Representing silent steps explicitly with `Tau` allows us, for example, to represent diverging computation without violating Coq's guardedness condition [5]:

```
CoFixpoint spin {E R} : itree E R := Tau spin.
```

The final, and most interesting, way to build an ITree is with the `Vis X e k` constructor. Here, `e : E X` is a "visible" external effect (including any outputs provided by the computation to its environment) and `X` is the type of data that the environment provides in response to the event. The constructor also specifies a continuation, `k`, which produces the rest of the computation given the response from the environment. `Vis` creates branches in the interaction tree because `k` can behave differently for distinct values of type `X`.

Here is a small example that defines a type `IO` of output or input interactions, each of which works with natural numbers. It is then straightforward to define an ITree computation that loops forever, echoing each input received to the output:

```
Variant IO : Type → Type :=
| Input  : IO nat
| Output : nat → IO ().

CoInductive echo : itree IO () :=
  Vis Input (fun x => Vis (Output x) (fun _ => echo)).
```

3.2. From Specification to Tester

From an ITree specification, I conducted "offline" testing, which takes a trace and determines its validity [9], and "online" testing, where the specification is derived into a client program that validates the system under test interactively [11].

3.2.1. Offline testing of swap server.

I started with testing a simple "swap server" [9], specified in Figure 3.2. The specification says that the server can either accept a connection with a new client (`obs_connect`) or else receive a message from a client over some established connection (`obs_msg_to_server c`), send back the current stored message (`obs_msg_from_server c last_msg`), and then start over with the last received message as the current state.

To test this swap server, I wrote a client program that interacts with the server and produces a trace of requests and responses, and a function that determines whether the trace t is a trace of the linear specification s *i.e.* whether `is_trace s t` in Figure 3.1 holds.

To network nondeterminism, the checker enumerates all possible server-side message orders that can explain the client-side observations, and checks if any of them satisifes the protocol specification.

```
CoFixpoint linear_spec' (conns : list connection_id)
           (last_msg : bytes) : itree specE unit :=
  or ( (* Accept a new connection. *)
       c <- obs_connect;;
       linear_spec' (c :: conns) last_msg )
     ( (* Exchange a pair of messages on a connection. *)
       c <- choose conns;;
       msg <- obs_msg_to_server c;;
       obs_msg_from_server c last_msg;;
       linear_spec' conns msg ).

Definition linear_spec := linear_spec' [] zeros.
```

FIGURE 3.2. Linear specification of the swap server. In the `linear_spec'` loop, the parameter `conns` maintains the list of open connections, while `last_msg` holds the message received from the last client (which will be sent back to the next client). The server repeatedly chooses between accepting a new connection or doing a receive and then a send on some existing connection picked in the list `conns`. The linear specification is initialized with an empty set of connections and a message filled with zeros.

3.2.2. Online testing of HTTP. To test protocols with internal nondeterminism (*e.g.* HTTP) effectively, I introduced a symbolic representation for the server's invisible choices, as shown in Figure 3.3. I then defined a TCP network model in Figure 3.4. Combining the server and network models produces a model program that exhibits all valid observations, considering both internal and network nondeterminism.

From the server and network models, I derived a tester client that interacts with servers over the network, and validates the observations against the protocol specification, as shown in Figure 3.5.

Using this automatially derived tester program, I have found violations against HTTP/1.1 in the latest version of both Apache and Nginx. More details are explained in Li, Pierce, and Zdancewic [11].

3.2.3. Key innovation. To solve the problem of "determinining whether an observation is explainable by a nondeterministic program", I reduced it into a constraint satisfiability: Although the tester doesn't know the server and network's exact choices, it can gain some knowledge of these invisible choices by observing the trace of messages. If the invisible choices are represented as symbolic variables, then an observed trace is valid if there exists some value for the variables that explains this trace, which can be determined by a constraint solver.

```
(* matches : (etag * exp etag) → exp bool *)
(* IF      : (exp bool * T * T) → T       *)
let put (k    : key,
         t    : etag,
         v    : value,
         data : key → value,
         xtag : key → exp etag) =
    IF (matches(t, xtag[k]),
    (* then *)
       xt := fresh_tag();
       let xtag' = update(xtag, k, xt) in
       let data' = update(data, k, v)  in
       return (OK, xtag', data'),
    (* else *)
       return (PreconditionFailed, xtag, data))
```

FIGURE 3.3. Symbolic model handling conditional PUT request. The model maintains two states: `data` that maps keys to their values, and `xtag` that maps keys to symbolic variables that represent their corresponding ETags. Upon receiving a PUT request conditioned over "If-Match: `t`", the server should decide whether the request ETag `matches` that stored in the server. Upon matching, the server processes the PUT request, and represents the updated value's ETag as a fresh variable.

```
let tcp (buffer : list packet) =
    let absorb =
        pkt := recv();
        tcp (buffer ++ [pkt]) in
    let emit =
        let pkts = oldest_in_each_conn(buffer) in
        pkt := pick_one(pkts);
        send(pkt);
        tcp (remove(pkt, buffer)) in
    or (absorb, emit)
```

FIGURE 3.4. Network model for concurrent TCP connections. The model maintains a `buffer` of all packets en route. In each cycle, the model may nondeterministically branch to either absorb `or` emit a packet. Any absorbed packet is appended to the end of buffer. When emitting a packet, the model may choose a connection and send the oldest packet in it.

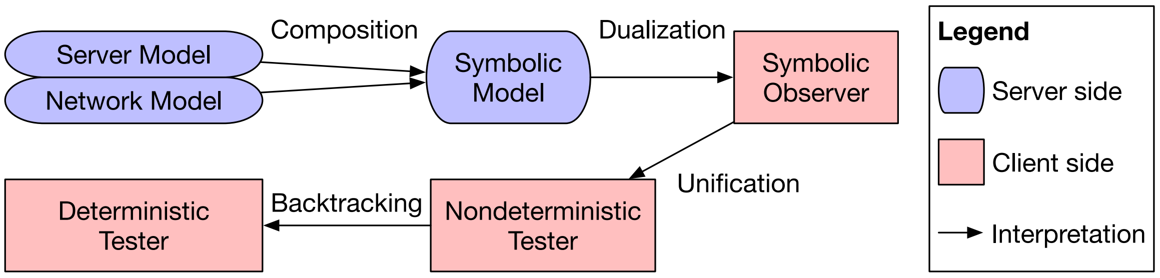

Figure 3.5. Deriving tester program from specification

CHAPTER 4

Theories Developed for Interactive Testing

During the testing practice in chapter 3, the tester's quality was evaluated by mutation testing, *i.e.* running the tester against buggy implementations to see if it rejects. To formally prove that the tester is good, I develop a theory for reasoning on testers' good properties.

Interactive testing is a process that reveals the SUT's interactions and determines whether it satisfies the specification. There are two kinds of interactions: (1) *inputs* that the tester can specify, and (2) *outputs* that are observed from the SUT. In particular, when testing networked systems, the input is a message sent by the tester, and the output is a message received from the SUT.

When viewing the SUT as a function from inputs to outputs, we can test the system by (1) providing an input, (2) get the output, and (3) validating the input-output pair. This process is called *synchronous testing.*

However, the nature of networked systems is that multiple messages might arrive at the system simultaneously, and a high-throughput system should handle the messages concurrently. To check the system's validity upon concurrent inputs, the tester should send multiple messages, rather than executing "one client at a time". This non-blocking process is called *asynchronous testing.*

My goal is to formalise the techniques in Li, Pierce, and Zdancewic [11] into a generic theory for asynchronous testing.

A tester consists of two parts: (i) a test harness that interacts with the SUT and observes the interactions, and (ii) a validator that determines whether the observations satisfy the specification.

The test harness needs to produce counterexamples effectively, and provide good coverage of test cases. The goal is to locate unknown bugs within a fixed budget, which is more practical than theoretical, and will be discussed in **??**. The test theory in this dissertation focuses on guaranteeing the soundness and completeness of the validator logic.

CHAPTER 5

Related Work

5.1. Specifying and Testing Protocols

Modelling languages for specifying protocols can be partitioned into three styles, according to Anand et al. [1]: (1) *Process-oriented* notations that describe the SUT's behavior in a procedural style, using various domain-specific languages like our interaction trees; (2) *State-oriented* notations that specify what behavior the SUT should exhibit in a given state, which includes variants of labelled transition systems (LTS); and (3) *Scenario-oriented* notations that describe the expected behavior from an outside observer's point of view (*i.e.,* "god's-eye view").

The area of model-based testing is well-studied, diverse, and difficult to navigate [1]. Here we focus on techniques that have been practiced in testing real-world programs, which includes notations (1) and (2). Notation (3) is infeasible for protocols with nontrivial nondeterminism, because the specification needs to define observer-side knowledge of the SUT's all possible internal states, making it complex to implement and hard to reason about, as shown in Figure 2.1.

Language of Temporal Ordering Specification (LOTOS) [**Bolognesi1987**] is the ISO standard for specifying OSI protocols. It defines distributed concurrent systems as *processes* that interact via *channels*, and represents internal nondeterminism as choices among processes.

Using a formal language strongly insired by LOTOS, Tretmans and Laar [13] implemented a test generation tool for symbolic transition systems called TorXakis, which has been used for testing Dropbox [13].

TorXakis provides limited support for internal nondeterminism. Unlike our testing framework that incorporates symbolic evalutation, TorXakis enumerates all possible values of internally generated data, until finding a corresponding case that matches the tester's observation. This requires the server model to generate data within a reasonably small range, and thus cannot handle generic choices like HTTP entity tags, which can be arbitrary strings.

Bishop et al. [3] have developed rigorous specifications for transport-layer protocols TCP, UDP, and the Sockets API, and validated the specifications against mainstream implementations in FreeBSD, Linux, and WinXP. Their specification represents internal nondeterminism as symbolic states of the model, which is then evaluated using a special-purpose symbolic model checker. They focused on developing a post-hoc specification that matches existing systems, and wrote a separate tool for generating test cases.

5.2. Reasoning about Network Delays

For property-based testing against distributed applications like Dropbox, Hughes et al. [8] have introduced “conjectured events” to represent uploading and downloading events that nodes may perform at any time invisibly.

Sun, Xu, and Elbaum [12] symbolised the time elapsed to transmit packets from one end to another, and developed a symbolic-execution-based tester that found transmission-related bugs in Linux TFTP upon certain network delays. Their tester used a fixed trace of packets to interact with the server, and the generated test cases were the packets’ delay time.

Bibliography

[1] Saswat Anand et al. "An orchestrated survey of methodologies for automated software test case generation". In: *Journal of Systems and Software* 86.8 (2013), pp. 1978–2001. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2013.02.061. URL: http://www.sciencedirect.com/science/article/pii/S0164121213000563.

[2] Andrew W. Appel et al. "Position paper: the science of deep specification". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375.2104 (Oct. 2017), p. 20160331. ISSN: 1364-503X. DOI: 10.1098/rsta.2016.0331. URL: https://royalsocietypublishing.org/doi/10.1098/rsta.2016.0331.

[3] Steve Bishop et al. "Engineering with Logic: Rigorous Test-Oracle Specification and Validation for TCP/IP and the Sockets API". In: *J. ACM* 66.1 (Dec. 2018). ISSN: 0004-5411. DOI: 10.1145/3243650. URL: https://doi.org/10.1145/3243650.

[4] Manfred Broy et al. "Model-based testing of reactive systems". In: *Volume 3472 of Springer LNCS*. Springer. 2005.

[5] Adam Chlipala. "Infinite Data and Proofs". In: *Certified Programming with Dependent Types*. MIT Press, 2017. URL: http://adam.chlipala.net/cpdt/html/Cpdt.Coinductive.html.

[6] Roy T. Fielding and Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests*. RFC 7232. June 2014. DOI: 10.17487/RFC7232. URL: https://rfc-editor.org/rfc/rfc7232.txt.

[7] George Fink and Matt Bishop. "Property-Based Testing: A New Approach to Testing for Assurance". In: *SIGSOFT Softw. Eng. Notes* 22.4 (July 1997), pp. 74–80. ISSN: 0163-5948. DOI: 10.1145/263244.263267. URL: https://doi.org/10.1145/263244.263267.

[8] John Hughes et al. "Mysteries of DropBox: Property-Based Testing of a Distributed Synchronization Service". In: *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*. 2016, pp. 135–145. DOI: 10.1109/ICST.2016.37. URL: https://doi.org/10.1109/ICST.2016.37.

[9] Nicolas Koh et al. "From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server". In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2019. Cascais, Portugal: ACM, 2019, pp. 234–248. ISBN: 978-1-4503-6222-1. DOI: 10.1145/3293880.3294106. URL: http://doi.acm.org/10.1145/3293880.3294106.

[10] Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*. Software Foundations series, volume 4. Electronic textbook, 2018. URL: `https://softwarefoundations.cis.upenn.edu/qc-current/index.html`.

[11] Yishuai Li, Benjamin C. Pierce, and Steve Zdancewic. "Model-Based Testing of Networked Applications". In: *ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2021.

[12] Wei Sun, Lisong Xu, and Sebastian Elbaum. "Improving the Cost-Effectiveness of Symbolic Testing Techniques for Transport Protocol Implementations under Packet Dynamics". In: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2017. Santa Barbara, CA, USA: Association for Computing Machinery, 2017, pp. 79–89. ISBN: 9781450350761. DOI: `10.1145/3092703.3092706`. URL: `https://doi.org/10.1145/3092703.3092706`.

[13] Jan Tretmans and Piërre van de Laar. "Model-Based Testing with TorXakis: The Mysteries of Dropbox Revisited". In: *Strahonja, V.(ed.), CECIIS: 30th Central European Conference on Information and Intelligent Systems, October 2-4, 2019, Varazdin, Croatia. Proceedings*. Zagreb: Faculty of Organization and Informatics, University of Zagreb. 2019, pp. 247–258.

[14] Li-yao Xia et al. "Interaction Trees: Representing Recursive and Impure Programs in Coq". In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019), 51:1–51:32. ISSN: 2475-1421. DOI: `10.1145/3371119`. URL: `http://doi.acm.org/10.1145/3371119`.