

TESTING BY DUALIZATION

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 Committee

Steve Zdancewic, Professor of Computer and Information Science

Boon Thau Loo, Professor of Computer and Information Science

Rajeev Alur, Professor of Computer and Information Science

Leonidas Lampropoulos, Assistant Professor of Computer Science, University of Maryland

TESTING BY DUALIZATION

COPYRIGHT

2022

Yishuai Li

This work is licensed under the

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/>

ACKNOWLEDGEMENT

Write your acknowledgement text here.

ABSTRACT

TESTING BY DUALIZATION

Yishuai Li

Benjamin C. Pierce

Software engineering requires rigorous testing to guarantee the product’s quality. Semantic testing of functional correctness is challenged by nondeterminism in systems’ behavior, which makes testers difficult to write and reason about.

This thesis presents a language-based technique for testing interactive systems. I propose a theory for specifying and validating systems’ nondeterministic behaviors, with guaranteed soundness and correctness. I then apply the theory to testing practices, and show how to derive specifications into interactive tester programs. I also introduce a language design for producing test inputs that can effectively detect and reproduce systems’ invalid behaviors.

I evaluate the methodology by specifying and testing real-world systems such as web servers and file synchronizers, demonstrating the derived testers’ ability to find disagreements between the specification and the implementation.

TABLE OF CONTENTS

ACKNOWLEDGEMENT	iii
ABSTRACT	iv
CHAPTER 1 : INTRODUCTION	1
1.1 Interactive Testing	1
1.2 Internal and external nondeterminism	3
1.3 Test harness and inter-execution nondeterminism	8
1.4 State of the Art	10
1.5 Contribution	12
CHAPTER 2 : VALIDATOR THEORY	15
2.1 Concepts	15
2.2 QAC language family	16
2.3 Soundness and completeness of validators	23
CHAPTER 3 : SYNCHRONOUS VALIDATOR BY DUALIZATION	28
3.1 Encoding Specifications	28
3.2 Dualizing Specification Programs	32
3.3 Correctness Proof	38
CHAPTER 4 : ASYNCHRONOUS TESTER BY DUALIZATION	46
4.1 From QAC to Interaction Trees	46
4.2 Handling External Nondeterminism	52
4.3 Handling Internal Nondeterminism	57
4.4 Executing Tester Model	64
CHAPTER 5 : TEST HARNESS DESIGN	70

5.1	Overview	70
5.2	Heuristics for Test Generation	72
5.3	Shrinking Interactive Tests	74
CHAPTER 6 : EVALUATION		82
6.1	Testing Web Servers	82
6.2	Testing file synchronizer	88
CHAPTER 7 : RELATED WORK		93
7.1	From Specifications to Validators	93
7.2	Test Harnesses	95
CHAPTER 8 : CONCLUSION AND FUTURE WORK		99
BIBLIOGRAPHY		100

CHAPTER 1

INTRODUCTION

Software engineering requires rigorous testing of rapidly evolving programs, which costs manpower comparable to developing the product itself [37]. To guarantee programs' compliance with their specifications, we need testers that can tell compliant implementations from violating ones.

This thesis studies the testing of interactive systems' semantics: The system under test (SUT) interacts with the tester by sending and receiving messages, and the tester determines whether the messages sent by the SUT are valid with respect to the protocol specification.

This chapter provides a brief view of interactive testing (Section 1.1), explains why nondeterminism makes this problem difficult (Sections 1.2–1.3), discusses the field of existing works (Section 1.4), and summarizes the contributions of this thesis in addressing the challenges caused by nondeterminism (Section 1.5).

1.1. Interactive Testing

Suppose we want to test a web server that supports GET and PUT methods. The server is a stateful program written as a recursive function:¹²

```
CoFixpoint server (data: key → value) :=  
  request ← recv;;  
  match request with  
  | GET k    ⇒ send (data k);; server data  
  | PUT k v  ⇒ send Done   ;; server (data [k ↦ v])  
end.
```

Here the `server` function iterates over a parameter called `data`, which is a key-value store.

In each iteration, the server receives a request and computes its response. It then sends back the response and recurses with the updated data.

¹Syntax “`x ← f;; y`” encodes a monadic program that binds the result of computation `f` as variable `x` in continuation `y`. For example, to receive a request is to bind the result of `recv` as variable `request` in the remaining program that performs pattern matching on it.

²Syntax “`data [k↦v]`” represents a key-value store where `k` is mapped to `v`, and all other keys are mapped by `data`. That is, for all `k'` that are not equal to `k`, `(data [k↦v]) k'` is equal to `(data k')`.

We can write a tester client that interacts with the server and determines whether it behaves correctly:

```
CoFixpoint tester (data: key → value) :=
  request ← random;;
  send request;;
  response ← recv;;
  match request with
  | GET k   ⇒ if response =? data k
               then tester data
               else reject
  | PUT k v ⇒ if response =? Done
               then tester (data [k ↦ v])
               else reject
  end.
```

This tester implements a reference server internally that computes the expected behavior. The behavior is then compared against that produced by the SUT. The tester rejects the SUT upon any difference from the computed expectation.

The above tester can be viewed as two modules: (i) a *test harness* that interacts with the server and produces transactions of sends and receives, and (ii) a *validator* that determines whether the transactions are valid or not:

```
(* Compute the expected response and next state of the server. *)
Definition serverSpec request data :=
  match request with
  | GET k   ⇒ (data k, data)
  | PUT k v ⇒ (Done   , data [k ↦ v])
  end.

(* Validate the transaction against the stateful specification. *)
Definition validate spec request response data :=
  let (expect, next) := spec request data in
  if response =? expect then Success next else Failure.

(* Produce transactions for the validator. *)
CoFixpoint harness validator state :=
  request ← random;;
  send request;;
  response ← recv;;
  if validator request response state is Success next
  then harness validator next
  else reject.

Definition tester := harness (validate serverSpec).
```


This testing method works for deterministic systems, whose behavior can be precisely computed from their inputs. But, many systems are allowed to behave nondeterministically. For example, systems may implement various hash algorithms, or buffer network packets in different ways. The following sections discuss nondeterminism by partitioning it in two ways, and explains how they pose challenges to the validator and the test harness.

1.2. Internal and external nondeterminism

When people talk to each other, voice is transmitted over substances like air or metal. When testers interact with the SUT, messages are transmitted via the runtime environment. The specification might allow SUTs to behave differently from each other, just like people speaking in different accents; we call it *internal nondeterminism*. The runtime environment might affect the transmission of messages, just like solids transmit voice faster than liquids and gases; we call it *external nondeterminism*.

1.2.1. Internal nondeterminism

Within the SUT, correct behavior may be underspecified. Consider web browsing as an example: If a client has cached a local copy of some resource, then when fetching updates for the resource, the client can ask the server not to send the resource's contents if it is the same as the cached copy. To achieve this, an HTTP server may generate a short string, called an “entity tag” (ETag) [14], 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 ETag 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 an `If-Match` precondition, which instructs the server to only update the content if its current ETag matches:

<pre>/* Client: */ PUT /target HTTP/1.1 If-Match: "tag-foo" ... content (A) ...</pre>	<pre>/* Server: */ HTTP/1.1 204 No Content</pre>
<pre>/* Client: */ GET /target HTTP/1.1</pre>	<pre>/* Server: */ HTTP/1.1 200 OK ETag: "tag-bar" ... content (A) ...</pre>

If the ETag does not match, then the server should not perform the requested operation, and should reject with code 412:

<pre>/* Client: */ PUT /target HTTP/1.1 If-Match: "tag-baz" ... content (B) ...</pre>	<pre>/* Server: */ HTTP/1.1 412 Precondition Failed</pre>
<pre>/* Client: */ GET /target HTTP/1.1</pre>	<pre>/* Server: */ HTTP/1.1 200 ok ETag: "tag-bar" ... content (A) ...</pre>

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 that includes an ETag), and cannot enumerate all of them (as ETags can be arbitrary strings), then it needs to maintain a space of all possible values, and narrow the space upon further interactions with the server. For example, "If the server has revealed some resource's ETag as `"tag-foo"`, then it must not reject requests targeting this resource conditioned over `If-Match: "tag-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."

This idea of remembering matched and mismatched ETags is implemented in Figure 1.1.

```

Definition validate request response
  (data      : key → value)
  (tag_is    : key → Maybe etag)
  (tag_is_not: key → list etag) :=
match request, response with
| PUT k t v, NoContent =>
  if t ∈ tag_is_not k then Failure
  else if (tag_is k =? Unknown) || strong_match (tag_is k) t
  then (* Now the tester knows that the data in [k]
        * is updated to [v], but its new ETag is unknown. *)
    Success (data [k ↦ v],
              tag_is [k ↦ Unknown],
              tag_is_not [k ↦ [] ])
  else Failure
| PUT k t v, PreconditionFailed =>
  if strong_match (tag_is k) t then Failure
  else (* Now the tester knows that the ETag of [k]
        * is other than [t]. *)
    Success (data, tag_is, tag_is_not [k ↦ t::(tag_is_not k)])
| GET k t, NotModified =>
  if t ∈ tag_is_not then Failure
  else if (tag_is k =? Unknown) || weak_match (tag_is k) t
  then (* Now the tester knows that the ETag of [k]
        * is equal to [t]. *)
    Success (data, tag_is [k ↦ Known t], tag_is_not)
  else Failure
| GET k t0, OK t v =>
  if weak_match (tag_is k) t0 then Failure
  else if data k =? v
  then (* Now the tester knows the ETag of [k]. *)
    Success (data, tag_is [k ↦ Known t], tag_is_not)
  else Failure
| _, _ => Failure
end.

```

Figure 1.1: Ad hoc tester for HTTP/1.1 conditional requests. PUT $k \ t \ v$ represents a PUT request that changes k 's value into v only if its ETag matches t ; GET $k \ t$ is a GET request for k 's value only if its ETag does not match t ; OK $t \ v$ indicates that the request target's value is v and its ETag is t .

For each key, the validator maintains three internal states: (i) The value stored in `data`, (ii) the corresponding resource’s ETag, if known by the tester, stored in `tag_is`, and (iii) ETags that are known to not match the resource’s, stored in `tag_is_not`. Each pair of request and response contributes to the validator’s knowledge of the target resource. The tester rejects the SUT if the observed behavior does not match the knowledge gained in previous interactions.

Even simple nondeterminism like ETags requires such careful design of the validator, based on thorough comprehension of the specification. We need to construct the validator in a scalable way for more complex protocols.

1.2.2. External nondeterminism

To discuss the nondeterminism caused by the environment, we need to define the environment concept in testing scenario.

Definition 1.1 (Environment, input, output, and observations). *Environment* is the substance that the tester and the SUT interact with. *Input* is the subset of the environment that the tester can manipulate. *Output* is the subset of the environment that the SUT can alter. *Observation* is the tester’s view of the environment.

When testing servers, the environment is the network stack between the client and the server. The input is the sequence of requests sent by the client, and the output is the sequence of responses sent by the server. The responses are transmitted via the network, until reaching the client side as observations.

The tester shown in Section 1.1 runs one client at a time. It waits for the response before sending the next request, as shown in Figure 1.2. Such tester’s observation is guaranteed identical to the SUT’s output, so it only needs to scan the requests and responses with one stateful validator.

To reveal the server’s behavior upon concurrent requests, the tester needs to simulate multiple clients, sending new requests before receiving previous responses. The network delay

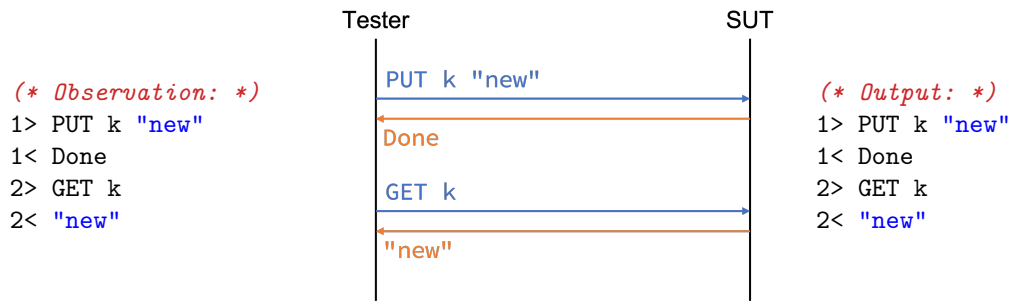


Figure 1.2: With no concurrency, the observation is identical to the output.

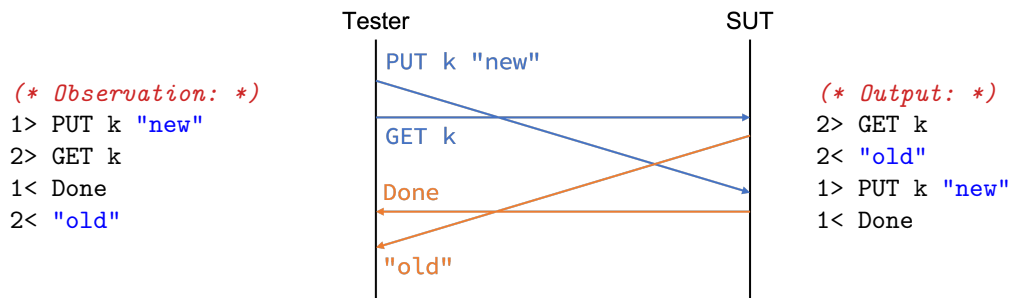


Figure 1.3: Acceptable: The observation can be explained by a valid output reordered by the network environment.

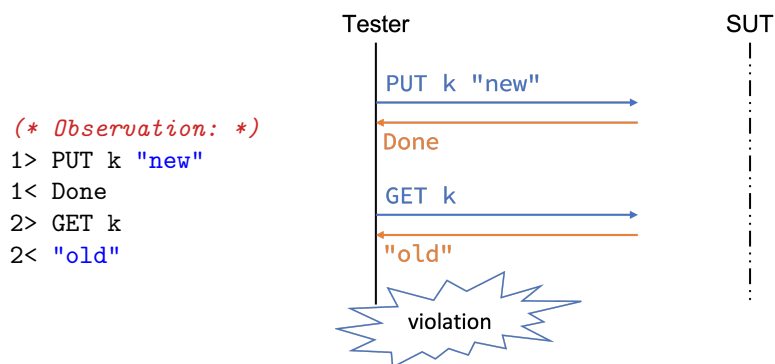


Figure 1.4: Unacceptable: The tester received the Done response before sending the GET request, thus the SUT must have processed the PUT request before the GET request. Therefore, the "old" response is invalid.

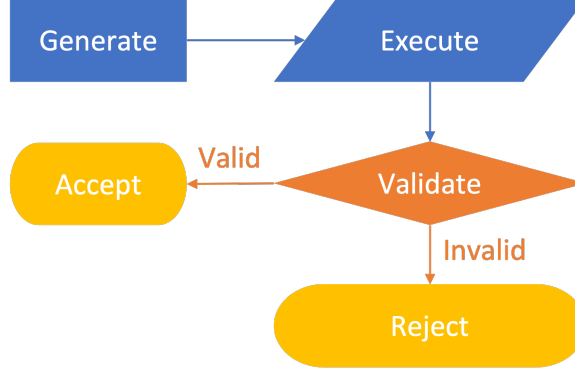


Figure 1.5: Simple tester architecture without shrinking.

might cause the server to receive requests in a different order from that on the tester side. Vice versa, responses sent by the server might be reordered before arriving at the tester, as shown in Figure 1.3. Such tester’s observation can be explained by various outputs on the SUT side. The validator needs to consider all possible outputs that can explain such observation, and see if anyone of them complies with the specification. If no valid output can explain the observation, then the tester should reject the SUT, as shown in Figure 1.4.

We need to construct a tester that can handle external nondeterminism systematically, and provide a generic way for reasoning on the environment.

1.3. Test harness and inter-execution nondeterminism

A good tester consists of (i) a validator that accurately determines whether its observations are valid or not, and (ii) a test harness that can reveal invalid observations effectively. Section 1.2 has explained the challenges in the validator. Here we discuss the test harness.

1.3.1. Test harness

Intuitively, a tester generates test input and launches the test execution. It then validates the observation and accepts/rejects the SUT, as shown in Figure 1.5.

However, to achieve better coverage, a randomized generator might produce huge test input [19]. Suppose the tester has revealed an invalid observation after thousands of interactions; such a report provides limited intuition of where the bug was introduced. To help

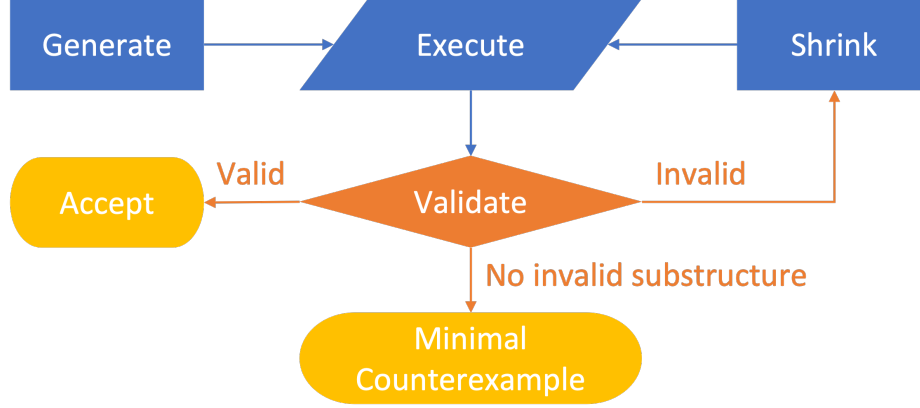


Figure 1.6: Tester architecture with shrinking mechanism.

developers locate the bug more effectively, the tester should present a *minimal counterexample* that can reproduce the violation. This is done by *shrinking* the failing input and rerunning the test with the input’s substructures. As shown in Figure 1.6, if a test input has no substructure that can cause any failure, then we report it as the minimal counterexample.

The test harness consists of generator, shrinker, and executor. This thesis studies the generator and the shrinker that produce the test input.

Interesting test inputs are those that are more likely to reveal invalid observations. Such subset is usually sparse and cannot be enumerated within reasonable budget *e.g.* in Subsection 1.2.1, request ETags that match the target resources’. The tester needs to manipulate the inputs’ distribution, by implementing *heuristics* that emphasize certain input patterns, which is challenged by another form of nondeterminism discussed as follows.

1.3.2. Inter-execution nondeterminism

Consider HTTP/1.1, where requests may be conditioned over timestamps. If a client has cached a version with a certain timestamp, then it can send the timestamp as `If-Modified-Since` precondition. The server should not transmit the request target’s content if its `Last-Modified` timestamp is not newer than the precondition’s:

```
/* Client: */
```

```
GET /index.html HTTP/1.1
If-Modified-Since: Mon, 16 May 2022 07:44:03 GMT

/* Server: */

HTTP/1.1 200 OK
Last-Modified: Tue, 17 May 2022 07:44:03 GMT
... content of target ...

/* Client: */

GET /index.html HTTP/1.1
If-Modified-Since: Tue, 17 May 2022 07:44:03 GMT

/* Server: */

HTTP/1.1 304 Not Modified
```

In this scenario, an interesting candidate for the `If-Modified-Since` precondition in a test case is the `Last-Modified` timestamp of a previous response. To emphasize this request pattern, the tester needs to implement heuristics that generate test inputs based on previous observations.

In case the tester has revealed invalid observations from the server, it needs to rerun the test with shrunk input. The timestamps on the server might be different from the previous execution, so an interesting timestamp in a previous run might become trivial in this run.

The uncertainty that a system may perform differently among executions is called *inter-execution nondeterminism*. Such nondeterminism poses challenges to the input minimization process: To preserve the input pattern, the shrunk HTTP/1.1 request should use the timestamps from the new execution. We need to implement a generic shrinking mechanism that can reproduce the heuristics in the test generator's design.

1.4. State of the Art

This section explains the context for this thesis. I discuss the state of the art in the field of testing, and describe their limitations in addressing the challenges posed by nondeterminism.

1.4.1. Property-based testing: QuickCheck

Property-based testing [16] is a testing methodology for validating semantic properties of programs' behavior. The properties are specified as executable boolean predicates over the behavior. To check whether an SUT satisfies a specification, the tester generates test input and executes the SUT with the generated input. The tester then observes the SUT's behavior, and computes the predicates with the observations.

Practices of property-based testing include QuickCheck for Haskell [10] and its variant QuickChick for Coq [23]. These tools can generate random inputs that satisfy logical conditions [24] and integrate with fuzz testing [25] and combinatorial testing [17], and have tested real-world systems like telecoms software [5] and Dropbox [20].

1.4.2. Model-based testing: TorXakis

Instead of specifying predicates over systems' behavior, model-based testing [9] defines an abstract model that produces valid behavior. When a tester observes an SUT's behavior, it checks whether the behavior is producible by the specification model.

Practical tools for model-based testing include TorXakis [36], whose modelling language is inspired by Language of Temporal Ordering Specification (LOTOS) [8], the ISO standard for specifying distributed systems. The tool can compile process algebra specifications into tester programs, and can be used for testing dropbox [35].

1.4.3. Limitations

In property-based testing, internal and external nondeterminism makes predicates difficult to write, as discussed in Section 1.2. TorXakis provides limited support for internal nondeterminism, allowing the specification to enumerate all possible values of internal choices. This works for scenarios where the space of choices is small *e.g.* within a dozen. When testing ETags in Subsection 1.2.1, it's infeasible to include a list of all strings in the specification.

To handle inter-execution nondeterminism in QuickCheck, Hughes [19] introduced abstract representations for generating and shrinking test inputs that can adapt to different runtime

observations. His technique works for synchronous interactions that blocks the tester to wait for observations, and lacks support for asynchronous testing where the SUT’s output may be indefinitely delayed by the environment.

1.5. Contribution

This thesis addresses the challenges in testing caused by various forms of nondeterminism. I introduce symbolic languages for specifying the protocol and representing test input, and *dualize* the specification into the tester’s (1) validator, (2) generator, and (3) shrinker:

1. The specification is written as a reference implementation—a nondeterministic program that exhibits all possible behaviors allowed by the protocol. Internal and external nondeterminism are represented by symbolic variables, and the space of nondeterministic behavior is defined by all possible assignments to the variables.

For internal nondeterminism, the validator computes the symbolic representation of the SUT’s output. The symbolic output expectation is then *unified* against the tester’s observations, reducing the problem of validating observations to constraint solving.

For external nondeterminism, I introduce a model that specifies the environment. The environment model describes the relation between the SUT’s output and the tester’s observations. By composing the environment model with the reference implementation, we get a tester-side specification that defines the space of valid observations.

2. Test generation heuristics are defined as computations from observations to the next input. To specify such heuristics in a generic way, I introduce intermediate representations for observations and test inputs, which are protocol-independent.

Heuristics in this framework produce symbolic test inputs that are parameterized over observations. During execution, the test harness computes the concrete input by *instantiating* the symbolic input’s arguments with runtime observations.

3. The language for test inputs is designed with inter-execution nondeterminism in mind.

By instantiating the inputs' symbolic intermediate representation with different observations, the test harness gets different test inputs but preserves the pattern.

To minimize counterexamples, the test harness only needs to shrink the inputs' symbolic representation. When rerunning the test, the shrunk input is reinstantiated with the new observations, thus reproduces the heuristics by the test generator.

Thesis claim Symbolic abstract representation can address challenges in testing interactive systems with uncertain behavior. Specifying protocols with symbolic reference implementation enables validating observations of systems with internal and external nondeterminism. Representing test input and observations symbolically allows generating and shrinking interesting test cases despite inter-execution nondeterminism. Combining these methods result in a rigorous tester that can capture protocol violations effectively.

This claim is supported by the following publications:

1. *From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server* [22], with Nicolas Koh, Yao Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic, where I developed a tester program based on a swap server's specification written as ITrees [39], and evaluated the tester's effectiveness by mutation testing.
2. *Verifying an HTTP Key-Value Server with Interaction Trees and VST* [41], with Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Li-yao Xia, Lennart Beringer, William Mansky, Benjamin C. Pierce, and Steve Zdancewic, where I developed the top-level specification for HTTP/1.1, and derived a tester client that revealed liveness and interrupt-handling bugs in our HTTP server, despite it was formally verified.
3. *Model-Based Testing of Networked Applications* [26], which describes my technique of specifying HTTP/1.1 with symbolic reference implementations, and from the specification, automatically deriving a tester program that can find bugs in Apache and Nginx.

Outline This thesis is structured as follows: Chapter 2 presents a theory for synchronous testing, introduces a language family for representing validators, and shows how to reason about their correctness. Chapter 3 applies the validator theory to an integer machine model that exhibits internal nondeterminism, showing how to derive validators automatically from protocol specifications. Chapter 4 transitions from synchronous testing to asynchronous testing, addressing external nondeterminism by specifying the environment. Chapter 5 presents a mechanism for generating and shrinking test inputs that address inter-execution nondeterminism. To evaluate the techniques I proposed in this thesis, Chapter 6 applies them to testing web servers and file synchronizers. I then compare my technique with related works in Chapter 7, and discuss future directions in Chapter 8.

CHAPTER 2

VALIDATOR THEORY

This chapter provides a theoretic view of synchronous testing that involve internal nondeterminism. Section 2.1 defines the basic concepts in testing. Section 2.2 introduces a language family for writing protocol specifications and validators. Section 2.3 shows how to reason on validators' soundness and completeness with respect to the specification.

2.1. Concepts

Testers are programs that determine whether implementations are compliant or not, based on its observations. This section defines the basic concepts and notations in interactive testing.

Definition 2.1 (Implementations and Traces). *Implementations* are programs that can interact with their environment. *Traces* are the implementations' inputs and outputs during execution. "Implementation i can *produce* trace t " is written as " $i \xrightarrow{t}$ ".

This chapter focuses on synchronous testing that assumes no external nondeterminism. Here the tester's observation is identical to the implementation's output, so the tester-side trace is the same as that on the implementation side. Asynchronous testing will be discussed in Chapter 4.

Definition 2.2 (Specification, Validity, and Compliance). A *specification* is a description of valid traces. "Trace t is *valid* per specification s " is written as " $\text{valid}_s t$ ".

An implementation i *complies* with a specification s (written as " $\text{comply}_s i$ ") if it only produces traces that are valid per the specification:

$$\text{comply}_s i \triangleq \forall t, (i \xrightarrow{t}) \implies \text{valid}_s t$$

Definition 2.3 (Tester components and correctness). A tester consists of (i) a *validator* that

accepts or rejects traces (written as “ $\text{accept}_v t$ ” and “ $\neg\text{accept}_v t$ ”), and (ii) a *test harness* that triggers different traces with various inputs.

A tester is *correct* if its acceptances and rejections are sound and complete. A tester is *rejection-sound* if it only rejects non-compliant implementations; it is *rejection-complete* if it can reject all non-compliant implementations, provided sufficient time of execution.¹

The tester’s correctness is based on its components’ properties: A rejection-sound tester requires its validator to be rejection-sound; A rejection-complete tester consists of (i) a rejection-complete validator and (ii) an exhaustive test harness that can eventually trigger invalid traces. The validators’ soundness and completeness are defined as follows:

Definition 2.4 (Correctness of validators). A validator v is *rejection-sound* with respect to specification s (written as “ $v \text{ sound}_s^{\text{Rej}}$ ”) if it only rejects traces that are invalid per s :

$$v \text{ sound}_s^{\text{Rej}} \triangleq \forall t, \neg\text{accept}_v t \implies \neg\text{valid}_s t$$

A validator v is *rejection-complete* with respect to specification s (written as “ $v \text{ complete}_s^{\text{Rej}}$ ”) if it rejects all behaviors that are invalid per s :

$$v \text{ complete}_s^{\text{Rej}} \triangleq \forall t, \neg\text{valid}_s t \implies \neg\text{accept}_v t$$

The rest of this chapter shows how to construct specifications and validators, and how to prove the validators’ correctness with respect to the specifications.

2.2. QAC language family

In this section, I introduce the “query-answer-choice” (QAC) language family for writing specifications and validators for network protocols that involve internal nondeterminism.

¹The semantics of “soundness” and “completeness” vary among contexts. This thesis inherits terminologies from existing literature [34], but explicitly uses “rejection-” prefix for clarity. “Rejection soundness” is equivalent to “acceptance completeness”, and vice versa.

2.2.1. Specifying protocols with server models

Network protocols can be specified with “reference implementations” *i.e.* model programs that exhibit the space of valid behaviors. For client-server systems such as WWW, we can specify networked servers as programs that receive queries and compute the responses. Here I model the server programs with a data structure called state monad.

Definition 2.5 (State monad). Let S be the state type, A be the result type, then type $(S \rightarrow A \times S)$ represents a computation that, given a pre state, yields a result and the post state. This computation is pronounced a “state monad with state type S and result type A ”.

For example, let the state be a key-value mapping $(K \rightarrow V)$, then we can define `get` and `put` computations as follows:

$$\text{get} : K \rightarrow ((K \rightarrow V) \rightarrow V \times (K \rightarrow V)) \quad (1)$$

$$\text{get}(k)(f) \triangleq (f(k), f) \quad (2)$$

$$\text{put} : K \times V \rightarrow ((K \rightarrow V) \rightarrow () \times (K \rightarrow V)) \quad (3)$$

$$\text{put}(k, v)(f) \triangleq ((), f[k \mapsto v]) \quad (4)$$

These function definitions should be read as:

1. The `get` function takes a key as argument, and constructs a state monad with state type $(K \rightarrow V)$ and result type V .
2. Given argument k of type K , `get`(k) takes a mapping f as pre state and yields the mapped value $f(k)$ as result. The post state is the original mapping f unchanged.
3. The `put` function takes a key-value pair as argument, and constructs a state monad with state type $(K \rightarrow V)$ and result type “()” (unit type, which corresponds to `void` return type in C/Java functions).

4. Given argument (k, v) of type $(K \times V)$, $\text{put}(k, v)$ takes a mapping f as pre state and substitutes its value at key k with v . The post state is the substituted mapping $f[k \mapsto v]$.

Now we can define the server model in terms of state monad:

Definition 2.6 (Deterministic server model). A deterministic server is an infinite loop whose loop body takes a query and produces a response. The server definition consists of the loop body and a current state:

$$\text{DeterministicServer} \triangleq \exists S, (Q \rightarrow S \rightarrow A \times S) \times S$$

This type definition is pronounced as: A deterministic server has an initial state of some type S . Its loop body takes a request of type Q and computes a state monad with state type S and result type A , where type A represents the response.

Notice that the server's state type is existentially quantified [29], while its query and response types are not. This is because a protocol specification only defines the space of valid traces, and doesn't require the implementation's internal state to be a specific type.

An instance of server model is written as:

$$\text{pack } S = \sigma \text{ with } (\text{sstep}, \text{state}_0)$$

This expression is pronounced as: The server state is of type σ . Its loop body is function sstep (which has type $Q \rightarrow \sigma \rightarrow A \times \sigma$) and its initial state is state_0 (which has type σ).

For example, consider a compare-and-set (CMP-SET) protocol: The server stores a number n . If the client sends a request that is smaller than S , then the server responds with 0. Otherwise, the server sets n to the request and responds with 1:

```
int n = 0;
while (true) {
```



```

int request = recv();
if (request <= n) send(0);
else { n = request; send(1); }
}

```

Such a server can be modelled as:

$$\text{pack } S = \mathbb{Z} \text{ with } (\lambda(q)(n) \Rightarrow \begin{cases} (0, s) & q \leq n \\ (1, q) & \text{otherwise} \end{cases}, \\ 0)$$

In general, servers' responses and transitions might depend on choices that are invisible to the testers, so called internal nondeterminism, as discussed in Subsection 1.2.1. I represent the space of nondeterministic behaviors by parameterizing it over the server's internal choice.

Definition 2.7 (Nondeterministic server model). A nondeterministic server is an infinite loop whose loop body takes a query and an internal choice to produce a responses. The nondeterministic server definition extends Definition 2.6 with a choice argument of type C :

$$\text{Server} \triangleq \exists S, (Q \times C \rightarrow S \rightarrow A \times S) \times S$$

Consider changing the aforementioned CMP-SET into compare-and-reset (CMP-RST): When the request is greater than S , the server may reset S to any arbitrary number:

```

int arbitrary();
int n = 0;
while (true) {
    int request = recv();
    if (request <= n) send(0);
    else { n = arbitrary(); send(1); }
}

```

Its corresponding server model can be written as:

$$\text{pack } S = \mathbb{Z} \text{ with } (\lambda(q, c)(n) \Rightarrow \begin{cases} (0, n) & q \leq n \\ (1, c) & \text{otherwise} \end{cases}, \\ 0)$$

This model represents the space of uncertain behavior with the internal choice parameter of type integer. For any value $(c : \mathbb{Z})$, the server is allowed to reset S to c .

2.2.2. Valid traces of a server model

By specifying protocols with server models, we can now instantiate the trace validity notation “ $\text{valid}_s t$ ” in Definition 2.2 in terms of operational semantics.

Definition 2.8 (Server transitions). Upon request q and choice c , server model s can step to s' yielding response a (written “ $s \xrightarrow[c]{(q, a)} s'$ ”) if and only if the response and the post model can be computed by the `stepServer` function:

$$\begin{aligned} s \xrightarrow[c]{(q, a)} s' &\triangleq \text{stepServer}(q, c)(s) = (a, s') \\ \text{stepServer} : Q \times C &\rightarrow \text{Server} \rightarrow A \times \text{Server} \\ \text{stepServer}(q, c)(s) &\triangleq \\ &\text{let } (\text{pack } S = \sigma \text{ with } (\text{sstep}, \text{state})) = s \text{ in} \\ &\text{let } (a, \text{state}') = \text{sstep}(q, c)(\text{state}) \text{ in} \\ &(a, \text{pack } S = \sigma \text{ with } (\text{sstep}, \text{state}')) \end{aligned}$$

The `stepServer` function takes a query and a choice and computes a state monad with state type `Server` and result type `A`, by pattern matching on argument $(s : \text{Server})$. Let σ be the server state type of s , `sstep` be the loop body, and $(\text{state} : \sigma)$ be the current state of s , then

$\text{stepServer}(q, c)(s)$ computes the result $(a : A)$ and the post state $(state' : \sigma)$ using the sstep function, and substitutes the server's pre-step $state$ with the post-step $state'$.

Definition 2.9 (Trace validity in QAC). In the QAC language family, a trace is a sequence of $Q \times A$ pairs. When specifying a protocol with a **Server** model, a trace t is valid per specification s if and only if it can be *produced* by the server model:

$$\text{valid}_s t \triangleq \exists s', s \xrightarrow{t} s'$$

Here the producibility relation in Section 2.1 is expanded with an argument s' representing the post-transition state, pronounced “specification s can produce trace t and step to specification s' ”:

1. A server model can produce an empty trace and step to itself:

$$s \xrightarrow{\varepsilon} s$$

2. A server model can produce a non-empty trace if it can produce the head of the trace, and step to some server model that produces the tail of the trace:

$$s \xrightarrow{t+(q,a)} s_2 \triangleq \exists s_1, c, s \xrightarrow{t} s_1 \wedge s_1 \xrightarrow[c]{(q,a)} s_2$$

2.2.3. Validating traces

The validator takes a trace and determines whether it is valid per the protocol specification.

Definition 2.10 (Validator). A validator is an infinite loop whose loop body takes a pair of query and response and determines whether it is valid or not. The validator iterates over a state of some type V . Given a $Q \times A$ pair, the loop body may return a next validator

state or return nothing, written as type “option V ”:

$$\begin{aligned}\text{Validator} &\triangleq \exists V, (Q \times A \rightarrow V \rightarrow \text{option } V) \times V \\ \text{option } X &\triangleq \text{Some } (x : X) \mid \text{None}\end{aligned}$$

For example, a validator for the CMP-SET protocol is written as:

$$\begin{aligned}\text{pack } V = \mathbb{Z} \text{ with } (\lambda(q, a)(v) \Rightarrow &\begin{cases} \text{if } a \text{ is 1 then Some } v \text{ else None} & q \leq v \\ \text{if } a \text{ is 1 then Some } q \text{ else None} & \text{otherwise} \end{cases}, \\ 0) &\end{aligned}$$

Here the validator state the same as the server model's. The loop body computes the expected response and compares it with the observed response. If they are the same, then the next server state is used as the next validator state. Otherwise, the function returns `None`, indicating that the response is invalid.

Having defined the validator type, we can now instantiate the trace acceptance notation “ $\text{accept}_v t$ ” in Definition 2.3 in terms of operational semantics.

Definition 2.11 (Validator transitions). Validator v can consume request q and response a and step to v' (written “ $v \xrightarrow{(q,a)} v'$ ”) if and only if the post validator can be computed by the `stepValidator` function:

$$\begin{aligned}v \xrightarrow{(q,a)} v' &\triangleq \text{stepValidator}(q, a)(v) = \text{Some } v' \\ \text{stepValidator} : Q \times A &\rightarrow \text{Validator} \rightarrow \text{option Validator} \\ \text{stepValidator}(q, a)(\text{pack } V = \beta &\text{ with } (\text{vstep}, \text{state})) \\ &\triangleq \begin{cases} \text{Some } (\text{pack } V = \beta \text{ with } (\text{vstep}, \text{state}')) & \text{vstep}(q, a, \text{state}) = \text{Some } \text{state}' \\ \text{None} & \text{vstep}(q, a, \text{state}) = \text{None} \end{cases}\end{aligned}$$

The `stepValidator` function takes a query and a response, and computes the validator transition by pattern matching on argument $(v : \text{Validator})$. Let β be the validator state type of v , `vstep` be the loop body, and $(state : \beta)$ be the current state of v , then `stepValidator` $(q, a)(v)$ calls the `vstep` function to validate the $Q \times A$ pair. If the pair is valid, then `vstep` returns a post-validation $state'$, which replaces the validator's current $state$. Otherwise, the validator halts with `None`.

Definition 2.12 (Trace acceptance in QAC). A validator accepts a trace if it consumes the entire trace:

$$\text{accept}_v t \triangleq \exists v', v \xrightarrow{t} v'$$

1. A validator can consume an empty trace and step to itself:

$$v \xrightarrow{\varepsilon} v$$

2. A validator consumes a non-empty trace if it can consume the head of the trace, and step to some validator that consumes the tail of the trace:

$$v \xrightarrow{t+(q,a)} v_2 \triangleq \exists v_1, v \xrightarrow{t} v_1 \wedge v_1 \xrightarrow{(q,a)} v_2$$

2.3. Soundness and completeness of validators

We can now phrase the correctness properties in Section 2.1 in terms of the QAC language family:

1. A rejection-sound (*i.e.* acceptance-complete) validator consumes all traces that are producible by the protocol specification:

$$\begin{aligned} v \text{ sound}_s^{\text{Rej}} &\triangleq \forall t, \neg \text{accept}_v t \implies \neg \text{valid}_s t \\ &\triangleq \forall t, (\exists s', s \xrightarrow{t} s') \implies \exists v', v \xrightarrow{t} v' \end{aligned}$$

2. A rejection-complete (*i.e.* acceptance-sound) validator only consumes traces that are producible by the protocol specification:

$$\begin{aligned} v \text{ complete}_s^{\text{Rej}} &\triangleq \forall t, \neg \text{valid}_s t \implies \neg \text{accept}_v t \\ &\triangleq \forall t, (\exists v', v \xrightarrow{t} v') \implies \exists s', s \xrightarrow{t} s' \end{aligned}$$

Both the specification and the validator are infinite loops. To show that the validator consumes the same space of traces as the specification produces, we need to show the correspondence between each server and validator step. This is done by introducing some loop invariant between the server and validator states, and show that it is preserved by the server's and the validator's loop body.

This section shows how to prove that validator **pack** $V = \beta$ **with** (vstep, v_0) is sound and complete with respect to the server model **pack** $S = \sigma$ **with** (sstep, s_0) . The core of the proof is the loop invariant defined as a binary relation between the validator state β and the server state σ . Notation “ $(v \sim s)$ ” is pronounced “validator state v simulates server state s ”.

2.3.1. Proving rejection soundness

To prove that any trace producible by the server is consumable by the validator, I perform forward induction on the server's execution path, and show that every step has a corresponding validator step:

- The initial server state s_0 simulates the initial validator state v_0 :

$$v_0 \sim s_0 \quad (\text{RejSound-Init})$$

- Any server step $s \xrightarrow[c]{(q,a)} s'$ whose pre-execution state s reflects some pre-validation state v can be consumed by the validator yielding a post-validation state v' that reflects the

post-execution state s' :

$$\begin{aligned}
& \forall (q : Q)(c : C)(a : A)(s, s' : \sigma)(v : \beta), & (\text{RejSound-Step}) \\
& s \xrightarrow[c]{(q,a)} s' \wedge v \sim s \\
& \implies \exists (v' : \beta), v \xrightarrow{(q,a)} v' \wedge v' \sim s'
\end{aligned}$$



Here syntax “ $s \xrightarrow[c]{(q,a)} s'$ ” and “ $v \xrightarrow{(q,a)} v'$ ” are simplified from Definition 2.8 and Definition 2.11, representing the server and validator instances by their states. This is because their state types σ, β and step functions `sstep`, `vstep` remain constant over the transitions.

2.3.2. Proving rejection completeness

Rejection completeness says that any trace consumable by the validator is producible by the server model. I construct the server’s execution path $s \xrightarrow{t} s'$ by *backward* induction of the validation path $v \xrightarrow{t} v'$:

- Any accepting validator step $v \xrightarrow{(q,a)} v'$ has some server state s' that reflects the post-validation state v' :

$$\begin{aligned}
& \forall (q : Q)(a : A)(v, v' : \beta), v \xrightarrow{(q,a)} v' & (\text{RejComplete-End}) \\
& \implies \exists s' : \sigma, v' \sim s'
\end{aligned}$$

This gives us a final server state from which we can construct the server’s execution

path inductively.

- Any accepting validator step $v \xrightarrow{(q,a)} v'$ whose post-validation state v' reflects some post-execution server state s' has a corresponding server step from a pre-execution state s that reflects the pre-validation state v :

$$\forall (q : Q)(a : A)(v, v' : \beta)(s' : \sigma), \quad (\text{RejComplete-Step})$$

$$\text{vstep}(q, a, v) = \text{Some } v' \wedge v' \sim s'$$

$$\implies \exists (s : \sigma)(c : C), \text{sstep}(q, c, s) = (a, s') \wedge v \sim s$$



This allows us to inductively construct an execution path starting from some server state that reflects the initial validator state v_0 . To show the existence of an execution path that starts from the initial server state s_0 , we need the following hypothesis:

- The initial validator state v_0 only reflects the initial server state s_0 :

$$\{s \mid v_0 \sim s\} = \{s_0\} \quad (\text{RejComplete-Init})$$

Rejection soundness is proven by forward induction, while rejection completeness is proven by backward induction. This is because the choice C is known from the server step, but unknown from the validator step: Given a validator step, we cannot predict “what choices the server will make in the future”, but can analyze “what choices the server might have made in the past”. This proof strategy is formalized in the Coq proof assistant, and will be

demonstrated with an example in Section 3.3.

CHAPTER 3

SYNCHRONOUS VALIDATOR BY DUALIZATION

As discussed in Section 1.2, nondeterminism makes validators difficult to write. To address this challenge, I construct validators *automatically* from the specification. The key idea is to encode the specification with a programming language, and *dualize* the specification program into the validator.

This chapter demonstrates the dualization technique with a programming language in the QAC family. Section 3.1 introduces the **Prog** language for encoding specifications. Specification written in **Prog** are dualized into validators in Section 3.2, with correctness proven in Section 3.3.

3.1. Encoding Specifications

Constructing the validator automatically requires analyzing the computations of the specification program. The QAC language family in Section 2.2 only provides a state monad interface for server models, which is insufficient for performing structural analysis. This section introduces a programming language for encoding specifications.

For readability, I demonstrate the dualization technique on a subset of QAC server models called integer machine model, featuring random-access memory (RAM) and arithmetic operations. To test real-world systems like web servers, I'll employ a more complex specification language in Chapter 4.

3.1.1. Integer machine model

The server state of an integer machine model is a key-value mapping that resembles a RAM. The addresses are natural numbers, and the data are integers. The initial server state has

zero data in all addresses:

$$\begin{aligned}
s_0 &: \mathbb{N} \rightarrow \mathbb{Z} \\
s_0 &= (_ \mapsto 0) \\
i.e. \quad &\forall (k : \mathbb{N}), s_0!k = 0
\end{aligned}$$

Here syntax “ $s!k$ ” is pronounced “data stored in address $!k$ of memory s ”.

The server’s query, response, and choices (Q, A, C) are of type integer. At the beginning of each server loop, the internal choice is written to address $!0$, and the query is written to address $!1$. The server then performs some computation $f : (\mathbb{N} \rightarrow \mathbb{Z}) \rightarrow (\mathbb{N} \rightarrow \mathbb{Z})$ that manipulates the memory, and then sends back the value stored in address $!1$ as the response:

$$\begin{aligned}
\text{sstep}_f(q, c)(s) &\triangleq \text{let } s_1 = s[0 \mapsto c] \text{ in} \\
&\quad \text{let } s_2 = s_1[1 \mapsto q] \text{ in} \\
&\quad \text{let } s_3 = f(s_2) \text{ in} \\
&\quad (s_3!1, s_3)
\end{aligned}$$

Each memory-manipulating computation f defines an instance of the integer machine model:

$$\text{pack } S = \mathbb{N} \rightarrow \mathbb{Z} \text{ with } (\text{sstep}_f, s_0)$$

Dualizing an integer machine model requires structural analysis of its memory manipulation. Next I’ll introduce a programming language to encode computations $(\mathbb{N} \rightarrow \mathbb{Z}) \rightarrow (\mathbb{N} \rightarrow \mathbb{Z})$.

3.1.2. The Prog modelling language

Syntax A program in the **Prog** language may read and write at any address of the memory, perform arithmetic operations, and make conditional branches:

Prog	\triangleq	return	end computation
		!dst := SExp; Prog	write to address $dst \in \mathbb{N}$
		if SExp \leq SExp then Prog else Prog	conditional branch
SExp	\triangleq	\mathbb{Z}	constant integer
		!src	read from address $src \in \mathbb{N}$
		SExp op SExp	$op \in \{+, -, \times, \div\}$

For example, the following program computes the absolute value of data stored in !1, and stores it in address !1:

```

if    !1  $\leq$  0
then  !1  $\leftarrow$  (0-!1); return
else  return

```

Semantics Each program ($p : \text{Prog}$) specifies a computation on the integer machine:

$$\begin{aligned}
& \text{eval} : \text{Prog} \rightarrow (\mathbb{N} \rightarrow \mathbb{Z}) \rightarrow (\mathbb{N} \rightarrow \mathbb{Z}) \\
& \text{eval}(p)(s) \triangleq \begin{cases} s & p \text{ is return} \\ \text{eval}(p')(s[dst \mapsto e^s]) & p \text{ is } !dst := e; p' \\ \text{eval}(\text{if } e_1^s \leq e_2^s \text{ then } p_1 \text{ else } p_2)(s) & p \text{ is if } e_1 \leq e_2 \text{ then } p_1 \text{ else } p_2 \end{cases} \\
& e^s \triangleq \begin{cases} z & e \text{ is } z : \mathbb{Z} \\ s!src & e \text{ is } !src \\ e_1^s \text{ op } e_2^s & e \text{ is } e_1 \text{ op } e_2 \end{cases}
\end{aligned}$$

Here “ e^s ” is pronounced “evaluating server expression ($e : \text{SExp}$) on memory ($s : \mathbb{N} \rightarrow \mathbb{Z}$)”.

It substitutes all occurrences of “ $!src$ ” with the data stored in address $!src$ of memory s .

Syntax “ $s[k \mapsto v]$ ” is pronounced “writing value v to address $!k$ of memory s ”. It produces a new memory that maps address $!k$ to v , while other addresses remain unchanged from s :

$$s[k \mapsto v] \triangleq k' \mapsto \begin{cases} v & k' = k \\ s!k' & k' \neq k \end{cases}$$

From Prog to server model Every program in the Prog language corresponds to a server model that performs the computations it specifies:

$\text{serverOf} : \text{Prog} \rightarrow \text{Server}$

$\text{serverOf}(p) \triangleq \text{pack } S = \mathbb{N} \rightarrow \mathbb{Z} \text{ with } (\text{sstep}_{\text{eval}(p)}, s_0)$

For example, the CMP-RST protocol in Subsection 2.2.1 can be constructed by applying serverOf to the following program:

if $!1 \leq !2$ then $!1 := 0$; return (1)

else $!1 := 1$; $!2 := !0$; return (2)

The constructed server stores its data n in address $!2$. When the query stored in $!1$ is less than or equal to $!2$ (case 1), the server writes 0 to address $!1$ as response, and leaves the data untouched in address $!2$. For queries greater than $!2$ (case 2), the server writes 1 as response, and updates the data in $!2$ with the internal choice stored in $!0$.

Based on specifications written in the Prog language, we can now construct the validator automatically by dualization.

3.2. Dualizing Specification Programs

This section presents an algorithm that constructs a validator from the specification program:

$$\text{validatorOf} : \text{Prog} \rightarrow \text{Validator}$$

For every program p , $\text{validatorOf}(p)$ determines whether a trace is producible by $\text{serverOf}(p)$:

$$\begin{aligned} &\forall(p : \text{Prog})(t : \text{list } (Q \times A), \\ &(\text{validatorOf}(p) \xrightarrow{t}) \iff (\text{serverOf}(p) \xrightarrow{t}) \end{aligned}$$

More specifically, given a trace of $Q \times A$ pairs, the validator determines whether there exists a sequences of internal choices C that explains how the server produces the trace in Definition 2.9.

The idea is similar to the **tester** in Section 1.1, which **validates** the trace by executing the **serverSpec**, and comparing the expected response against the tester's observation.

However, executing a nondeterministic specification does not produce a specific expectation of response, but a space of responses parameterized over the internal choices. Therefore, upon observing a response A , the validator should determine whether there exists a choice C that leads the specification to produce this response.

This reduces the trace validation problem to constraint solving. The validator maintains a set of constraints that unify the responses produced by the specification against the responses observed from the implementation.

More specifically, the validator executes the **Prog** model and represents internal choices with *symbolic variables*. These variables are carried along the program execution, so the expected responses are computed as *symbolic expressions* that might depend on those variables. The validator then constrains that the symbolic response is equal to the concrete observation.

To achieve this goal, the validator stores a symbolic variables for each address of the server model. It also remembers all the constraints added upon observation. These information are called “validation states”:

$$(\mathbb{N} \rightarrow \text{var}) \times \text{set constraint}$$

Here the **constraints** are relations between validator expressions (**VExps**) that may depend on symbolic **variables**:

$$\begin{array}{lll} \text{constraint} & \triangleq & \text{VExp } cmp \text{ VExp} \quad cmp \in \{<, \leq, =\} \\ \text{VExp} & \triangleq & \mathbb{Z} \quad \text{constant integer} \\ & | & \#x \quad \text{variable } x \in \text{var} \\ & | & \text{VExp } op \text{ VExp} \quad op \in \{+, -, \times, \div\} \end{array}$$

The **VExp** type replaces **SExp**’s address constructor $!src$ with variable constructor $\#x$. This allows the validator to constrain the values of the same address at different times *e.g.* the internal choice stored in $!0$ in different iterations. The validation state maps each address to its current representing variable $(\mathbb{N} \rightarrow \text{var})$, which updates as the validator symbolically executes the server program.

Notice that the server program in Subsection 3.1.2 has conditional branches. When executing the specification program, the branch condition might depend on the internal choices, which is invisible to the validator. As a result, the validator doesn’t know the exact branch taken by the specification, so it maintains multiple validation states, one for each possible execution path:

$$\text{set } ((\mathbb{N} \rightarrow \text{var}) \times \text{set constraint})$$

The initial state of the validator is a single validation state that corresponds to the specification’s initial state:

$$\{(_ \mapsto \#0, \{\#0 = 0\})\}$$

Here the initial validation state says “all addresses are mapped to variable #0, and the value of variable #0 is constrained to be zero”. This reflects the initial server state that maps all addresses to zero value.

The validator’s loop body is derived by dualizing the server model:

1. When the server performs a write operation $!dst := exp$, the validator creates a fresh variable x to represent the new value stored in address $!dst$, and adds a constraint that says x ’s value is equal to that of exp . This rule also applies to writing the request to address $!1$ before executing the program.
2. When the server makes a nondeterministic branch if $e_1 \leq e_2$ then p_1 else p_2 , consider both cases: (a) If p_1 was taken, then the validator should add a constraint $e_1 \leq e_2$; or (b) If p_2 was taken, then the validator should add constraint $e_2 < e_1$.
3. Before executing the program, the server writes the internal choice c to address $!0$. Accordingly, the validator creates a fresh variable to represent the new value stored in address $!0$, without adding any constraint.
4. After executing the program, the server sends back the value stored in $!1$ as response. Accordingly, the validator adds a constraint that says the variable representing address $!1$ is equal to the observed response.
5. When the constraints of a validation state become unsatisfiable, it indicates that the server model cannot explain the observation. This is because either (i) the observation is invalid *i.e.* not producible by the server model, or (ii) the observation is valid, but was produced by a different execution path of the server model.
6. The validator accepts the trace if it can be produced by any execution path of the server model. Since each execution path corresponds to a validation state, the validator only needs to remove the unsatisfiable state from the set of states. If the set of validation states becomes empty, it indicates that the observation cannot be explained by any

$$\begin{aligned}
\text{vstep}_p(q, a)(v) &\triangleq \text{let } v' = v_0 \leftarrow v; \text{vstep}'_p(q, a)(v_0) \text{ in} \\
&\quad \text{if } v' \text{ is } \emptyset \text{ then None else Some } v' \tag{6} \\
\text{vstep}'_p(q, a)(v_0) &\triangleq \text{let } v_1 = \text{havoc}(0, v_0) \text{ in} \\
&\quad \text{let } v_2 = \text{write}(1, q, v_1) \text{ in} \\
&\quad (vs_3, cs_3) \leftarrow \text{exec}(p, v_2); \\
&\quad \text{let } cs_4 = cs_3 \cup \{\#(vs_3!1) = a\} \text{ in} \tag{4} \\
&\quad \text{if solvable } cs_4 \text{ then } \{(vs_3, cs_4)\} \text{ else } \emptyset \tag{5} \\
\text{exec}(p, (vs, cs)) &\triangleq \begin{cases} \{(vs, cs)\} & \text{if } p \text{ is return} \\ \text{exec}(p', \text{write}(d, e, (vs, cs))) & \text{if } p \text{ is } (!d := e; p') \\ \left(\begin{array}{l} \text{let } v_1 = (vs, cs \cup \{e_1^{vs} \leq e_2^{vs}\}) \text{ in} \\ \text{let } v_2 = (vs, cs \cup \{e_2^{vs} < e_1^{vs}\}) \text{ in} \\ \text{exec}(p_1, v_1) \cup \text{exec}(p_2, v_2) \end{array} \right) & \text{if } p \text{ is} \\ & \quad \text{(if } e_1 \leq e_2 \\ & \quad \text{then } p_1 \text{ else } p_2) \end{cases} \tag{2} \\
\text{write}(d, e, (vs, cs)) &\triangleq \text{let } x_e = \text{fresh } (vs, cs) \text{ in} \tag{1} \\
&\quad (vs[d \mapsto x_e], cs \cup \{\#x_e = e^{vs}\}) \\
\text{havoc}(d, (vs, cs)) &\triangleq \text{let } x_c = \text{fresh } (vs, cs) \text{ in } (vs[d \mapsto x_c], cs) \tag{3}
\end{aligned}$$

Figure 3.1: Dualizing server model into validator, with derivation rules annotated.

execution path of the specification, so the validator should reject the trace.

This mechanism is formalized in Figure 3.1. Here the notation “ $v_0 \leftarrow v; \text{vstep}'_p(q, a)(v_0)$ ” is a monadic bind for sets: Let vstep'_p map each element v_0 in v to a set of validation states $(\text{vstep}'_p(q, a)(v_0) : \text{set } ((\mathbb{N} \rightarrow \text{var}) \times \text{set constraint}))$, and return the union of all result sets as v' .

The validator adds constraints in three circumstances: Rule 1 says the write operation updates the destination with the source expression; Rule 2 guards the branch condition to match its corresponding execution path; Rule 4 unifies the server’s symbolic response against the concrete response observed from the implementation.

The constraints added in Rule 1 and Rule 2 are translated from the specification program. Given a server expression $(e : \text{SExp})$ from the source expression or the branch condition, syntax e^{vs} translates it into a validator expression VExp using the validation state vs , by

replacing its addresses with the corresponding variables:

$$e^{vs} \triangleq \begin{cases} n & e \text{ is } z : \mathbb{Z} \\ \#(vs!src) & e \text{ is } !src \\ e_1^{vs} \text{ op } e_2^{vs} & e \text{ is } e_1 \text{ op } e_2 \end{cases}$$

The validator assumes a constraint solver that can determine whether a set of constraints is satisfiable, *i.e.* whether there exists an *assignment* of variables ($\mathbf{var} \rightarrow \mathbb{Z}$) that satisfy all the constraints:

$$\forall cs, \text{solvable } cs \iff \exists (asgn : \mathbf{var} \rightarrow \mathbb{Z}), asgn \text{ satisfies } cs$$

$$asgn \text{ satisfies } cs \triangleq \forall (e_1 \text{ cmp } e_2) \in cs, e_1^{asgn} \text{ cmp } e_2^{asgn}$$

$$e^{asgn} \triangleq \begin{cases} z & e \text{ is } z : \mathbb{Z} \\ asgn!x & e \text{ is } \#x \\ e_1^{asgn} \text{ op } e_2^{asgn} & e \text{ is } e_1 \text{ op } e_2 \end{cases}$$

Here “ e^{asgn} ” is pronounced “evaluating validator expression ($e : \mathbf{VExp}$) with assignment ($asgn : \mathbf{var} \rightarrow \mathbb{Z}$)”. It substitutes all occurrences of “ $\#x$ ” with their assigned value ($asgn!x$).

Now we have the algorithm that constructs the validator from the specification program p :

$$\begin{aligned} \text{validatorOf}(p) &\triangleq \text{pack } V = \text{set } ((\mathbb{N} \rightarrow \mathbf{var}) \times \text{set constraint}) \text{ with} \\ &\quad (\text{vstep}_p, \{(_ \mapsto \#0, \{\#0 = 0\})\}) \end{aligned}$$

For example, to construct a validator for the CMP-RST protocol in Subsection 2.2.1, we

```

pack  $V = \text{set } ((\mathbb{N} \rightarrow \text{var}) \times \text{set constraint})$  with
(
   $\lambda(q, a)(v) \Rightarrow$  let  $v' = (vs_0, cs_0) \leftarrow v;$ 
    let  $vs_1 = vs_0[0 \mapsto \text{fresh } (vs_0, cs_0)]$  in (1)
    let  $x_q = \text{fresh } (vs_1, cs_0)$  in
    let  $vs_2 = vs_1[1 \mapsto x_q]$  in
    let  $cs_2 = cs_0 \cup \{\#x_q = q\}$  in
    let  $cs_{3a0} = cs_2 \cup \{\#(vs_2!1) \leq \#(vs_2!2)\}$  in (2a)
    let  $x_{3a1} = \text{fresh } (vs_2, cs_{3a0})$  in
    let  $vs_{3a1} = vs_2[1 \mapsto x_{3a1}]$  in
    let  $cs_{3a1} = cs_{3a0} \cup \{\#x_{3a1} = 0\}$  in
    let  $cs_{3b0} = cs_2 \cup \{\#(vs_2!2) < \#(vs_2!1)\}$  in (2b)
    let  $x_{3b1} = \text{fresh } (vs_2, cs_{3b0})$  in
    let  $vs_{3b1} = vs_2[1 \mapsto x_{3b1}]$  in
    let  $cs_{3b1} = cs_{3b0} \cup \{\#x_{3b1} = 1\}$  in
    let  $x_{3b2} = \text{fresh } (vs_{3b1}, cs_{3b1})$  in
    let  $vs_{3b2} = vs_{3b1}[2 \mapsto x_{3b2}]$  in
    let  $cs_{3b2} = cs_{3b1} \cup \{\#x_{3b2} = \#(vs_{3b2}!1)\}$  in
     $((vs_4, cs_4) \leftarrow \{(vs_{3a1}, cs_{3a1}), (vs_{3b2}, cs_{3b2})\};$  (3)
    let  $cs_5 = cs_4 \cup \{\#(vs_4!1) = a\}$  in
    if solvable  $cs_5$  then  $\{(vs_4, cs_5)\}$  else  $\emptyset$ 
  in
  if  $v'$  is  $\emptyset$  then None else Some  $v'$ 
,  $\{(\_ \mapsto \#0, \{\#0 = 0\})\}$ 
)

```

Figure 3.2: Validator for CMP-RST automatically derived from its specification in **Prog.** This program consists of three parts: (1) symbolizing the query and internal choice before executing the model, (2) considering both branches in the model program, propagating a validation state for each branch, and (3) filtering the validation states by constraint satisfiability, removing invalid states.

first specify it in **Prog** as:

```

if !1 ≤ !2
then !1 ← 0; return
else !1 ← 0; !2 ← !0; return

```

This program stores the data n in address !2. If the request is less than or equal to n , then the program writes response 0 to address !1, and leaves the data unchanged; Otherwise, it writes 1 as response, and updates address !2 with the internal choice in !0.

We then apply function `validatorOf` to this **Prog**-based specification, resulting in a validator as shown in Figure 3.2. Validators constructed in this way are proven correct in the next section.

3.3. Correctness Proof

So far I have introduced the **Prog** language for writing specifications and shown how to construct a validator from it. This section shows how to prove the soundness and completeness of all validators dualized from **Prog**-based specifications:

$$\begin{aligned}
& \forall p : \text{Prog}, \text{ let } s = \text{serverOf}(p) \text{ in} \\
& \quad \text{let } v = \text{validatorOf}(p) \text{ in} \\
& \quad v \text{ sound}_s^{\text{Rej}} \wedge v \text{ complete}_s^{\text{Rej}} \\
& \quad \text{i.e. } \forall t : \text{list } (Q \times A), \\
& \quad \quad \text{valid}_s t \iff \text{accept}_v t \\
& \quad \quad \text{i.e. } \exists s', s \xrightarrow{t} s' \iff \exists v', v \xrightarrow{t} v'
\end{aligned}$$

To apply the proof techniques in Section 2.3, I design the loop invariant in Subsection 3.3.1. I then use the invariant to prove the hypotheses for soundness and completeness in Subsection 3.3.2 and Subsection 3.3.3, respectively.

3.3.1. Designing loop invariant

Let $\beta = \text{set}((\mathbb{N} \rightarrow \text{var}) \times \text{set constraint})$ be the validator state type, and $\sigma = \mathbb{N} \rightarrow \mathbb{Z}$ be the server state type, then the loop invariant $(v : \beta) \sim (s : \sigma)$ is a binary relation between the validator state v and the server state s . To define this relation, I first discuss the semantics of validator states.

The validator state is a set of validation states that consist of an address-variable mapping and a set of constraints over the variables. Each validation state defines a space of server states. A validator accepts a trace if it has a validation state whose constraints are satisfiable, which indicates the existence of a server that produces the trace. A corresponding server state can be constructed from any assignment that satisfies the validation state's constraints:

$$vs^{asgn} \triangleq \text{addr} \mapsto asgn!(vs!\text{addr})$$

Let $(vs : \mathbb{N} \rightarrow \text{var})$ be a mapping in the validation state, $(asgn : \text{var} \rightarrow \mathbb{Z})$ be an assignment of variables, then $(vs^{asgn} : \mathbb{N} \rightarrow \mathbb{Z})$ is a server state that maps each address $!addr$ to the value assigned by $asgn$ to the address' corresponding variable in vs .

Definition 3.1 (Loop invariant for **Prog**-based validators). Validator state v reflects server state s (written “ $v \sim s$ ”) if the server state lies within the space of servers defined by some validation state $(vs, cs) \in v$. That is, there exists an assignment that relates the server state and the validation state:

$$\begin{aligned} v \sim s &\triangleq \exists (vs, cs) \in v, \exists asgn, \\ &asgn \text{ satisfies } cs \wedge vs^{asgn} = s \end{aligned}$$

Having defined the loop invariant, we only need to instantiate the hypotheses in Subsection 2.3.1–2.3.2 with **Prog**-based definitions. The rest of this section proves the hypotheses for establishing validators' soundness and completeness. The entire proof is formalized in

the Coq proof assistant.

3.3.2. Proving rejection soundness

Lemma 3.1 (RejSound-Init).

If: $vs = (_ \mapsto \#0) \quad cs = \{\#0 = 0\} \quad s = (_ \mapsto 0)$
 Then: $\{(vs, cs)\} \sim s$

Proof. Since (vs, cs) is the only element in the validator state, we only need to show that:

$$\exists(asgn : \text{var} \rightarrow \mathbb{Z}), \quad asgn \text{ satisfies } cs \wedge vs^{asgn} = s$$

By constructing the assignment as:

$$asgn = (_ \mapsto 0)$$

We have:

$$\#0^{asgn} = 0$$

Thus:

$$asgn \text{ satisfies } cs$$

We also know that:

$$\forall k, asgn!(vs!k) = 0 = (s!k)$$

Thus:

$$vs^{asgn} = s$$

□

Lemma 3.2 (RejSound-Step).

$$\begin{aligned}
& \forall(q, c, a : \mathbb{Z})(s, s' : \sigma)(v : \beta), \\
& s \xrightarrow[c]{(q,a)} s' \wedge v \sim s \\
& \implies \exists v' : \beta, v \xrightarrow{(q,a)} v' \wedge v' \sim s'
\end{aligned}$$

Proof. The invariant $v \sim s$ tells us that v contains a pre-validation state that reflects the pre-step server state s :

$$\exists(vs, cs) \in v, \exists asgn, \quad asgn \text{ satisfies } cs \wedge vs^{asgn} = s$$

The \mathbf{vstep}'_p function in Figure 3.1 leads to a set of post-validation states. We need to show that some state in it reflects the post-step server state s' :

$$\exists(vs', cs') \in \mathbf{vstep}'_p(q, a)(vs, cs), \exists asgn', \quad asgn' \text{ satisfies } cs' \wedge vs'^{asgn'} = s'$$

The server's internal choice c was provided, so we know the server's entire execution path. By induction on the **Prog** syntax, we can choose the right post-validation state (vs', cs') by computing each branch condition, and deduct the satisfying assignment $asgn'$ by computing each write operation's source value and destination variable. \square

3.3.3. Proving rejection completeness

Lemma 3.3 (RejComplete-End).

$$\begin{aligned}
& \forall(q, a : \mathbb{Z})(v, v' : \beta), v \xrightarrow{(q,a)} v' \\
& \implies \exists s' : \sigma, v' \sim s'
\end{aligned}$$

Proof. Since the \mathbf{vstep}_p function in Figure 3.1 checks the nonemptiness of the result, we know that v' must be nonempty. Consider validation state $(vs', cs') \in v'$. Since \mathbf{vstep}'_p checks that

(solvable cs'), we know that:

$$\exists asgn, \quad asgn \text{ satisfies } cs'$$

Let:

$$s' = vs' \text{ asgn}$$

Then we have:

$$(vs', cs') \in v' \wedge asgn \text{ satisfies } cs' \wedge vs' \text{ asgn} = s'$$

$$\text{i.e. } v' \sim s'$$

□

Lemma 3.4 (RejComplete-Step).

$$\forall (q, a : \mathbb{Z})(v, v' : \beta)(s' : \sigma),$$

$$v \xrightarrow{(q,a)} v' \wedge v' \sim s'$$

$$\implies \exists (s : \sigma)(c : \mathbb{Z}), s \xrightarrow[c]{(q,a)} s' \wedge v \sim s$$

Proof. We first construct the pre-step server state $(s : \sigma \mid v \sim s)$. We then compute the internal choice c and prove the server step $s \xrightarrow[c]{(q,a)} s'$.

The definition of $v' \sim s'$ says:

$$\exists (vs', cs') \in v', \exists asgn, \quad asgn \text{ satisfies } cs' \wedge vs' \text{ asgn} = s'$$

From the definition of \mathbf{vstep}_p , we know that:

$$\exists (vs, cs) \in v, \quad (vs', cs') \in \mathbf{vstep}'_p(q, a)(vs, cs)$$

Since vstep'_p monotonically increases the set of constraints, we have $cs \subseteq cs'$. Therefore:

$$\text{asgn satisfies } cs$$

Let:

$$s = vs^{\text{asgn}}$$

Then we have:

$$\begin{aligned} (vs, cs) \in v \quad \wedge \quad \text{asgn satisfies } cs \quad \wedge \quad vs^{\text{asgn}} = s \\ \text{i.e. } v \sim s \end{aligned}$$

From the definition of vstep'_p , the validator first creates a fresh variable to represent the server's internal choice, so we can deduct the internal choice from the assignment:

$$x_c = \text{fresh}(vs, cs) \quad c = \text{asgn}!x_c$$

Now we need to show that the server's loop body $\text{sstep}_p(q, c)(s)$ results in response a and post-execution state s' . Since the post-validation state v' simulates s' and guarantees the response to be a , we only need to prove the post-execution state to be s' .

We have known that:

$$s = vs^{\text{asgn}} \quad s' = vs'^{\text{asgn}} \quad (vs, cs) \xrightarrow{(q, a)} (vs', cs')$$

Therefore, we can prove $s \xrightarrow[(c)]{(q, a)} s'$ by induction on the **Prog** syntax, showing that every write or branch operation preserves asgn 's ability to unify the server state with the validation state. This leads the post-execution state to be unifiable against vs' using asgn , thus to be

s' .

□

The core of this proof is to find the internal choice c for server step $s \xrightarrow[c]{(q,a)} s'$. The choice was computed with the assignment deducted from the loop invariant. The assignment contains the value of all symbolic variables, which includes all choices made by the server, past and future. The loop invariant requires the assignment to explain the past choices, but cannot predict future choices. Therefore, we can only infer the server step from the validator step using backward induction.

Lemma 3.5 (RejComplete-Init).

$$\begin{array}{ll} \text{If:} & vs = (_ \mapsto \#0) \quad cs = \{\#0 \equiv 0\} \quad s_0 = (_ \mapsto 0) \\ \text{Then:} & \{s \mid \{(vs, cs)\} \sim s\} = \{s_0\} \end{array}$$

Proof. The requirement for s says:

$$\exists asgn : \mathbf{var} \rightarrow \mathbb{Z}, \quad asgn \text{ satisfies } cs \quad \wedge \quad vs^{asgn} = s$$

The constraint satisfaction tells us that:

$$asgn!0 = 0$$

We then have:

$$\forall k : \mathbb{N}, \quad s!k = asgn!(vs!k) = asgn!0 = 0 = s_0!k$$

Therefore, s_0 is the only server state that (vs, cs) simulates.

□

Now we have proven that all **Prog**-based validators satisfy the hypotheses defined in Subsection 2.3.1–2.3.2, and we can conclude that the validators constructed by dualization are sound and complete. Next I'll show how to apply this dualization technique to test real-world

programs.

CHAPTER 4

ASYNCHRONOUS TESTER BY DUALIZATION

So far I’ve introduced the theory of validating synchronous interactions using the QAC language family, and shown how to construct validators by dualization with a simple Prog language.

However, in real-world testing practices, there are more problems to consider. For example: How to interact with the SUT via multiple channels? How to handle external nondeterminism?

As discussed in Subsection 1.2.2, a networked server’s response may be delayed by the network environment, and an asynchronous tester may send other requests rather than waiting for the response. Therefore, we cannot view the trace as a sequence of $Q \times A$ pairs like we did in Definition 2.9, and the state monad in the QAC language family becomes insufficient for defining the space of asynchronous interactions.

This chapter applies the idea of dualization to testing asynchronous systems. I transition from the QAC language family to the ITree specification language, which allows modelling asynchronous interactions. The ITree-based specifications are dualized into tester programs that can interact with the SUT and reveal potential defects.

Section 4.1 introduces the ITree specification language. Section 4.2 and Section 4.3 address external and internal nondeterminism in the ITree context, using HTTP/1.1 as an example. Section 4.4 then explains how to execute the derived tester model as an interactive program.

4.1. From QAC to Interaction Trees

To write specifications for protocols’ rich semantics, I employed “interaction tree” (ITree), a generic data structure for representing interactive programs in the Coq programming language, introduced by Xia et al. [39]. ITrees allow specifying protocols as monadic programs that model valid implementations’ possible behavior. The model program can be interpreted

```

CoInductive itreeM (E: Type → Type) (R: Type) :=
  | Ret      : R → itreeM E R
  | Trigger : E R → itreeM E R
  | Bind    : ∀ {X : Type}, itreeM E X → (X → itreeM E R) → itreeM E R.

```

Figure 4.1: Mock definition of interaction trees.

into a tester program, as discussed in later sections.

4.1.1. Language definition

Consider an echo program, which keeps reading some data and writing it out verbatim, until reaching EOF. We can represent the program in Coq, with a reference in C as:

<pre> CoFixpoint echo := c ← getchar;; if c is EOF then EXIT else putchar c;; echo. </pre>	<pre> void echo() { const char c = getchar(); if (c == EOF) return; else { putchar(c); echo(); } } </pre>
---	---

Here the behavior after `getchar` depends on the value actually read. The monadic computation in Coq can be desugared into:

```

CoFixpoint echo :=
  Bind getchar
    (λ c ⇒ if c is EOF
          then EXIT
          else Bind (putchar c)
                  (λ _ ⇒ echo)
    ).

```

Such continuation-passing style can be represented as a tree of interactions. To help readers better understand the interaction tree language, I first provide a modified version of it that better shows its tree structure, and then explain the actual type definition used in practice.

Mock interaction trees As shown in Figure 4.1, a mock interaction tree (`itreeM`) has two kinds of leaves, `Ret` and `Trigger`, and has internal nodes constructed by `Bind`:

- (`Ret r`) represents a pure computation that yields a value `r`. In the echo example, `EXIT` halts the program with return value zero:

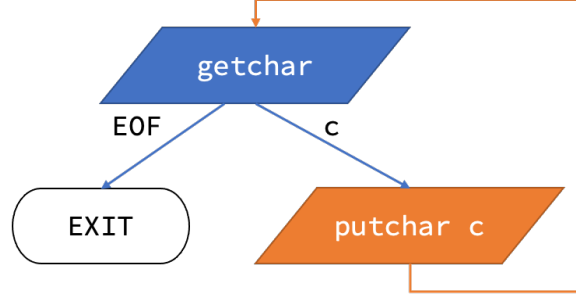


Figure 4.2: Interaction tree for echo program

Definition `EXIT := Ret 0.`

- `(Trigger e)` performs an impure event `e` and returns its result. Here $(e: E \rightarrow R)$ is an event whose result is of type `R`. For example, `getchar` has result type `char`, and `putchar`'s result type is `unit` (which corresponds to `void` in C/C++, or `()` in Haskell). These effective programs are constructed by triggering standard I/O events:

```

Variant stdioE: Type → Type := (* event type *)
  GetChar:      stdioE char
  | PutChar: char → stdioE unit.

```

Definition `getchar : itreeM stdioE char := Trigger GetChar.`

Definition `putchar (c: char) : itreeM stdioE unit := Trigger (PutChar c).`

- `(Bind m k)` binds the return value of `m` to the continuation function `k`. It first runs program `m` until it returns some value of type `X`. The return value $(x: X)$ then instantiates `k` into the following computation $(k\ x: \text{itreeM } E \rightarrow R)$. This corresponds to the monadic `(;;)` syntax:

```

Notation "x ← m1;; m2" := (Bind m1 (λ x ⇒ m2)).
Notation "m1;; m2"    := (Bind m1 (λ _ ⇒ m2)).

```

As illustrated in Figure 4.2, each possible return value `x` is an edge that leads to the child it instantiates, *i.e.*, $(k\ x)$. In this way, the `Ret` and `Trigger` nodes are connected into a tree structure.

The mock interaction tree provides an intuitive continuation-passing structure for representing impure programs. However, to implement dualization effectively, we need to revise the language definition of monadic binding.

```

CoInductive itree (E: Type → Type) (R: Type) :=
  Pure    : R → itree E R
| Impure  : ∀ {X : Type}, E X → (X → itree E R) → itree E R.

```

Figure 4.3: Formal definition of interaction trees (simplified)

Practical interaction trees Instead of binding a program to a continuation, `ITree` binds single events, as shown in Figure 4.3. I use `(Impure e k)` to replace `(Bind (Trigger e) k)` representations in `itreeM`. A `Pure` computation cannot be bound to a continuation, and must be the leaf of an `ITree`.¹

The `Ret`, `Trigger`, and `Bind` constructors introduced in `itreeM` have equivalent representations in `itree`, so we can still write programs in the monadic syntax:

```

Definition ret {E R} : R → itree E R := Pure.

Definition trigger {E R} (e: E R) : itree E R := Impure e Pure.

CoFixpoint bind {X E R} (m: itree E X) (f: X → itree E R) : itree E R :=
  match m with
  | Pure    x    ⇒ f x
  | Impure e k ⇒ Impure e (λ r ⇒ bind (k r) f)
  end.

Notation "x ← m1;; m2" := (bind m1 (λ x ⇒ m2)).
Notation "m1;; m2"     := (bind m1 (λ _ ⇒ m2)).

CoFixpoint translateM {E R} (m: itreeM E R) : itree E R :=
  match m with
  | Ret      r    ⇒ ret r
  | Trigger e    ⇒ trigger e
  | Bind m1 k    ⇒ x ← translateM m1;; translateM (k x)
  end.

```

[LYS: Under construction: End this section by embedding QAC into ITrees. Move interpretation to next section, which handles internal nondeterminism. After that, talk about external nondeterminism and network composition.]

`ITrees` can specify various kinds of programs like servers and testers, by defining different event types. For example, the QAC server in Definition 2.7 exhibits internal nondeterminism.

¹For readability, the “practical” `ITree` definition is a simplified version from Xia et al. [39]. Here `Pure` and `Impure` correspond to the `Ret` and `Vis` constructors. I dropped the `(Tau : itree E R → itree E R)` constructor, which carries no semantics and is used for satisfying Coq’s guardedness condition.

The internal choices made by the server can be represented as `Choice` events whose result can be any value in the space of choices:

```
Variant choiceE: Type → Type :=
  Choice: choiceE C.
```

The server also needs to send requests and receive responses:

```
Variant qaE: Type → Type :=
  Recv: qaE Q      (* receive a request *)
| Send: A → qaE unit. (* send a response *)
```

```
Definition qacE: Type → Type := qaE ⊕ choiceE.
```

Here `qacE` is a sum type of `qaE` and `choiceE` events, meaning that the server may send or receive messages, and may also make internal choices. I split the event types because they'll be handled differently when I derive the tester later in this chapter.

Now we can represent the QAC server with step function `sstep` and initial state σ :

```
CoInductive server (sstep: Q → C →  $\sigma$  → A *  $\sigma$ ) (s:  $\sigma$ )
  : itree qacE void :=
  c ← trigger Choice;;
  q ← trigger Recv;;
  let (a, s') := sstep q c s in
  trigger (Send a);;
  server sstep s'.
```

This subsection has provided a brief taste of the ITree specification language. To construct a tester from the specification, we need to dualize the model's behavior into the tester-side behavior, based on the theory explained in ???. To dualize specifications written in ITrees, we need an *interpretation* mechanism that transforms ITrees into other programs, which will be explained in the next subsection.

4.1.2. Interpreting interaction trees

To interpret a program `p` is to specify a rule that defines “if `p` does this, then do that”. For example, shell syntax (`p < input > output`) executes `p` but redirects its standard I/O. Suppose `p` is the `echo` program in Subsection 4.1.1, then the redirected program should perform file operations specified in `redirect_echo`:

```
Variant fileE: Type → Type :=      (* file operation events *)
```



```

Fgetc: file → fileE char
| Fputc: file → char → fileE unit.

CoInductive redirect_echo (input output: file) : itree fileE unit :=
  c ← trigger (Fgetc input);;
  if c is EOF then ret 0
  else trigger (Fputc output c);;
  redirect_echo input output.

```

When redirecting a program’s standard I/O to files, the interpretation rule is “whenever the program wants to read from or write to standard I/O, perform the read/write operation on the specified file instead”:

```

Definition redirect (input output: file) {R: Type} (e: stdioE R) :=
  match e in stdioE R return itree fileE R with
  | GetChar ⇒ trigger (Fgetc input)
  | PutChar c ⇒ trigger (Fputc output c)
  end.

```

Here the `redirect` function takes a standard I/O event and turns it into an ITree program that performs file events. The result program has the same return type as the original event, so it can “replace” the original `stdioE`. This is done by the `interp` function:

```

CoFixpoint interp {E F R} (f: ∀ {T}, E T → itree F T) (m: itree E R)
  : itree F R :=
  match m with
  | Pure r ⇒ Pure r
  | Impure e k ⇒ x ← f e;;
    interp handler (k x)
  end.

```

```

Definition redirect_echo2 (input output: file) : itree fileE unit :=
  interp (redirect input output) (translateM echo).

```

For each impure event, the interpreter replaces it with the program defined by the handler function `f`. As a result, `redirect_echo2` constructs a redirected echo program that is equivalent with `redirect_echo`.

To derive tester programs from ITree specifications, I’ll introduce multiple interpretation processes, with various event handlers throughout this chapter.

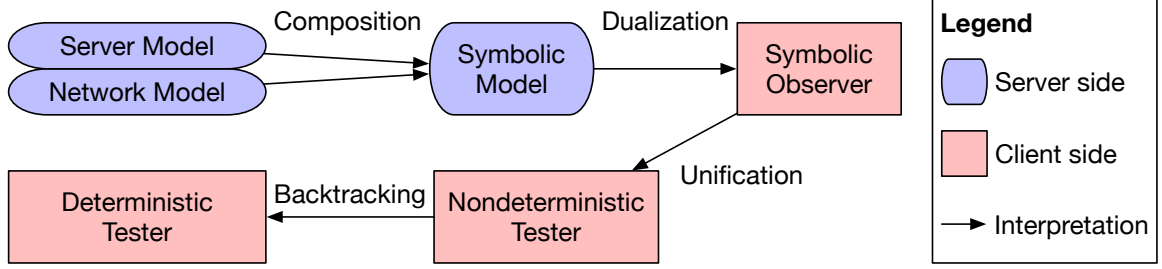


Figure 4.4: Deriving tester program from specification

4.2. Handling External Nondeterminism

As introduced in Subsection 1.2.2, the environment might affect the transmission of messages, so called external nondeterminism. The tester should take the environment into account when validating its observations.

This section explains how to address external nondeterminism by specifying the environment, with the networked server example. It corresponds to the “Composition” arrow in Figure 4.4. Subsection 4.2.1 defines a model for concurrent TCP connections. Subsection 4.2.2 then composes the network model with the server specification, yielding a tester-side specification that defines the space of valid observations.

4.2.1. Modelling the network

When testing servers over the network, request and response packets may be indefinitely delayed. As a result, messages from one end might arrive at the other end in a different order from that they were sent.

The space of network reorderings can be modelled by a *network model*, a conceptual program for the “network wire”. The wire can be viewed as a buffer, which absorbs packets and later emits them:

```

Variant netE: Type → Type := (* network event type *)
  Emit  : packet → netE unit
  | Absorb: netE packet.

```

After absorbing a packet, the wire may emit it immediately or after absorbing/emitting other packets. Such choices are modelled by nondeterministic `Or` branches:

```

Variant nondetE: Type → Type := (* nondeterministic branch *)
  Or: nondetE bool.

Definition or {E R} `nondetE -< E} (x y: itree E R) : itree E R :=
  b ← trigger Or;;
  if b then x else y.

```

Here `(or x y)` creates a nondeterministic program that may behave as either `x` or `y`. The `nondetE` here is a special case of `choiceE` defined in Subsection 4.1.1 with boolean space of choices, but they'll be handled differently during test derivation. The type signature `{E R} `nondetE -< E}` says the `(or)` combinator can apply to ITrees whose event `E` is a super-event of `nondetE`, and with arbitrary return type `R`. For example, arguments `x` and `y` can be of type `(itree (netE \oplus nondetE) void)`.

For example, the network model for concurrent TCP connections is defined in Figure 4.5. The model captures TCP's feature of maintaining the order within each connection, but packets in different connections might be reordered arbitrarily. When the wire chooses a packet to send, the candidate must be the oldest in its connection.

Notice the `pick_one` function, which might return (i) `Some p` or (ii) `None`. The network model then (i) emits packet `p` or (ii) absorbs a packet into `buffer`.

- When the given list `pkts` is empty, `pick_one` always returns `None`, because the wire has no packet in the `buffer`, and must absorb some packet before emitting anything.
- Given a non-empty linked list `(p::l')`, with `p` as head and `l'` as tail, `pick_one` might return `(Some p)`, meaning the wire can emit that packet; or it might return `None`, meaning the wire can still absorb packets into the buffer.

Such network model reflects the TCP environment, where messages are never lost but might be indefinitely delayed. In the next subsection, I'll demonstrate how to compose the server and network models into a client-side observation model.

```

Fixpoint pick_one (l: list packet) : itree nondetE (option packet) :=
  if l is p::l'
  then or (Ret (Some p)) (pick_one l')
  else ret None.

Definition oldest_in_each_conn : list packet → list packet := ...
(* filter the oldest packet in each connection *)

CoFixpoint tcp (buffer: list packet) : itree (netE ⊕ nondetE) void :=
  let absorb := pkt ← trigger Absorb;;
    tcp (buffer ++ [pkt]) in
  let emit p := trigger (Emit p);;
    tcp (remove pkt buffer) in
  let pkts := oldest_in_each_conn buffer in
  opkt ← pick_one pkts;;
  if opkt is Some pkt
  then emit pkt
  else absorb.

```

Figure 4.5: Network model for concurrent TCP connections. The model is an infinite program iterating over a `buffer` of all packets en route. In each iteration, the model either absorbs or emits some packet, depending on the current `buffer` state and the choice made in `pick_one`. 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.

4.2.2. Network composition

The network connects the server on one end to the clients on other ends. When one end sends some message, the network model absorbs it and later emits it to the destination.

To *compose* a server model with a network model is to pair the server's `Send` and `Recv` events with the network's `Absorb` and `Emit` events. Since the network model is nondeterministic, it might not be ready to absorb packets sent by the server. The network might also emit a packet before the server is ready to receive it.

To handle the asynchronicity among the server and network events, I insert message buffers between them. As shown in Figure 4.6, the *incoming buffer* stores the packets emitted by the network but not yet consumed by the server's `Recv` events, and the *outgoing buffer* stores the packets sent by the server but not yet absorbed by the network.

The server and the clients are the opposite ends of the network. Each packet has routing

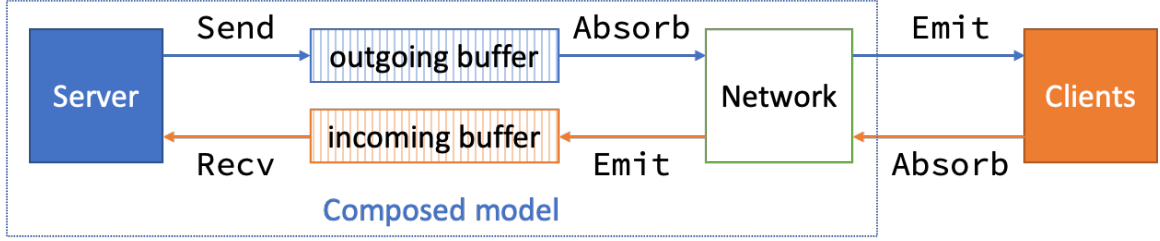


Figure 4.6: Network composition architecture

```

1  CoFixpoint compose {E} (srv: itree (qaE  $\oplus$  E) void)  (* server model *)
2      (net : itree (netE  $\oplus$  nondetE) void)  (* network model *)
3      (bi bo: list packet)  (* incoming and outgoing buffers *)
4      : itree (netE  $\oplus$  nondetE  $\oplus$  E) void :=
5  let step_net :=
6      match net with
7      | Impure (Absorb|) knet  $\Rightarrow$ 
8          match bo with
9          | pkt::bo'  $\Rightarrow$  compose srv (knet pkt) bi bo'
10         | []  $\Rightarrow$  pkt  $\leftarrow$  trigger Absorb;;
11                 compose srv (knet pkt) bi bo
12         end
13     | Impure (Emit pkt|) knet  $\Rightarrow$ 
14         if toServer pkt
15         then compose srv (knet tt) (bi++[pkt]) bo
16         else trigger (Emit pkt);;
17                 compose srv (knet tt) bi bo
18     | Impure (!Or) knet  $\Rightarrow$  b  $\leftarrow$  trigger Or;;
19                 compose srv (knet b) bi bo
20     | Pure vd  $\Rightarrow$  match vd in void with end
21     end
22 in
23 match srv with
24 | Impure (Recv|) ksrv  $\Rightarrow$ 
25     match bi with
26     | pkt::bi'  $\Rightarrow$  compose (ksrv pkt) net bi' bo
27     | []  $\Rightarrow$  step_net
28     end
29 | Impure (Send pkt|) ksrv  $\Rightarrow$ 
30     compose (ksrv tt) net bi (bo++[pkt])
31 | Impure (!e) ksrv  $\Rightarrow$   (* other events performed by the server *)
32     r  $\leftarrow$  trigger e;; compose (ksrv r) net bi bo
33 | Pure vd  $\Rightarrow$  match vd in void with end
34 end.

```

Figure 4.7: Network composition algorithm. When the server wants to send a packet in Line 29, the packet is appended to the outgoing buffer until absorbed by the network in Line 9, and eventually emitted to the client in Line 16. Conversely, packets sent by clients are absorbed by the network in Line 10, emitted to the server's incoming buffer in Line 15, until the server consumes it in Line 26.

fields that indicate its source and destination. When the network emits a packet, we need to determine whether the packet is emitted to the server's incoming buffer or to the clients, by inspecting its destination:

```
Record packet := {
  Source      : connection;
  Destination : connection;
  Data        : data
}.

Definition toServer (p: packet) : bool :=
  if p.(Destination) is server_conn then true else false.
```

Now we can define the composition algorithm formally, as shown in Figure 4.7. In this example, we reuse the `qaE` definition in Subsection 4.1.1, and let the `Q` and `A` both be the `packet` type. The composed `ITree` takes the server and network models as parameters, and makes steps in the two `ITrees` in certain order.

The composed model exhibits to the client three kinds of events: (i) Network operations (`netE`) where packets are emitted to or absorbed from clients, (ii) Nondeterministic branches (`nondetE`) made by the network model, and (iii) Other events `{E}` performed by the server model *e.g.* internal choices (`choiceE`) in Subsection 4.1.1.

Notice that this algorithm schedules the server at a higher priority than the network model. The composed model only steps into the network model when the server is starved in Line 27, by calling the `step_net` process defined in Line 5. Such design is to avoid divergence of the derived tester program, which I'll further explain in Section 4.4.

So far I've shown how to specify systems that exhibit external nondeterminism. By specifying the environment and composing it with the implementation-side specification, we can describe the space of valid observations. The rest of this chapter will show how to derive tester programs from the observer-side specification.

4.3. Handling Internal Nondeterminism

This section applies the dualization theory in Chapter 2 to the ITree context. I’ll show how to perform symbolic evaluation by interpreting ITree programs.

Subsection 4.3.1 explains how to represent systems’ internal choices as ITree’s symbolic events. It fills in the $\{E\}$ hole in Figure 4.7, and constructs the “Symbolic Model” box in Figure 4.4. Subsection 4.3.2 and Subsection 4.3.3 takes the observer-side specification composed in Subsection 4.2.2 and interprets it into a tester model, covering the “Dualization” and “Unification” arrows in the derivation framework.

4.3.1. Symbolizing internal choices

The key idea of language design is to expose symbolic representations to the dualization algorithm. The `Prog` language in ?? encodes data as symbolic expressions `SExp`, so that the responses and branch conditions may depend on internal choices. I do the same for ITree specifications, by symbolizing the choice events and branch conditions. Take my HTTP specification [26] as an example, its choice event has symbolic expression as result type:

```
Variant comparison := Strong | Weak.

Variant exp: Type → Type :=
  Const    : string → exp string
| Var      : var    → exp string
| Compare  : string → exp string → comparison → exp bool.

Variant choiceE: Type → Type :=
  Choice: symE (exp string).
```

Here I instantiate the `choiceE` in Subsection 4.1.1 with symbolic return type (`exp string`), pronounced “expression of type string”. In this example, I use strings to represent entity tags (ETags) that HTTP servers may generate, which was discussed in Subsection 1.2.1. The type interface can be adjusted to other protocols under test.

Symbolic expressions may be constructed as constant values, as variables, or with operators. The `Compare` constructor takes an expression of type string and compares it against a constant string. (`Compare t tx cmp`) represents the ETag comparison between `t` and `tx`, using “strong

```

Notation  $\sigma := (\text{path} \rightarrow \text{resource})$ .

CoFixpoint server_http (state:  $\sigma$ ) :=
  pq  $\leftarrow$  trigger Recv;;
  let respond_with a :=
    trigger (Send { Source      := server_conn;
                    Destination := pq.(Source);
                    Data        := a } ) in
  let q : request      := request_of pq in
  let v : content      := q.(Payload) in
  let k : path         := q.(TargetPath) in
  let t : string       := if_match q in
  let tx: exp string := (state k).(ETag) in
  IFX (Compare t tx Strong)
  THEN
    if q.(Method) is Put
    then
      tx'  $\leftarrow$  trigger Choice;;
      let state' := state [k  $\mapsto$  {Content := v; ETag := tx'}] in
      respond_with OK;;
      server_http state'
    else
      (* handling other kinds of requests *)
      (a, state')  $\leftarrow$  process q state;;
      respond_with a;;
      server_http state'
  ELSE
    respond_with PreconditionFailed;;
    server_http s.

```

Figure 4.8: Server model for HTTP conditional requests

comparison” or “weak comparison” mechanism [14] specified by `cmp`. The constant ETag is provided by the request, and the symbolized one comes from the server state.

Figure 4.8 shows an ITree model for If-Match requests in Subsection 1.2.1. The server first evaluates the request’s If-Match condition by “strong comparison” as required by HTTP. If the request’s ETag matches its target’s, then the server updates the target’s contents with the request payload. The target’s new ETag `tx'` can be of any value, so the model represents it as Choice event.

Notice that the server model exhibits two kinds of branches: (1) The `if` branches are provided by the ITree’s embedding language Coq, which takes a boolean value as condition; (2) The IFX branches are constructors of ITrees with nondeterministic branches, where the condition

is a symbolic expression of type `bool`:

```
Variant branchE: Type → Type :=
  Decide: exp bool → branchE bool.

Notation "IFX condition THEN x ELSE y" :=
  (b ← trigger (Decide condition));
  if b then x else y).
```

These two kinds of branch conditions play different roles in the specification, and will be handled differently during testing:

1. The “pure” `if` condition is used for deterministic branches like `(q.(Method) is Put)` in the example. Here `q` is a “concrete request”² generated by the tester and sent to the server, so its method is known by the tester and needn’t be symbolically evaluated.
2. The “symbolic” `IFX` condition here plays a similar rule as the `if` branches in the `Prog` language: Which branch to take depends on the server’s internal choices, so the tester needs to consider both cases.

Now we can fill the hole `{E}` in Figure 4.7. The server model receives concrete requests and sends symbolic responses, so its event type is defined as:

```
Record packet := {
  Source      : connection;
  Destination : connection;
  Data        : request + symbolic_response
}.

Variant qaE: Type → Type :=
  Recv : qaE packet
| Send : packet → qaE unit.
```

The HTTP server, for example, can be modelled as:

```
Definition server_http: itree (qaE ⊕ choiceE ⊕ branchE) void :=
  server init_state.
```

The server model is then transformed via network composition into a symbolic model for test derivation purposes:

²In this chapter, “concrete” messages are those that don’t involve symbolic variables, as opposed to “symbolic” messages.

```

1 Notation oE := (observeE  $\oplus$  nondetE  $\oplus$  choiceE  $\oplus$  constraintE).
2
3 Definition observe {R} (e: smE R) : itree oE R :=
4   match e with
5     | (Absorb |)       $\Rightarrow$  trigger FromObserver
6     | (Emit px|)       $\Rightarrow$  p  $\leftarrow$  trigger ToObserver;;
7                       trigger (Guard px p)
8     | (|||Branch bx)  $\Rightarrow$  or (trigger (Unify bx true));; ret true)
9                           (trigger (Unify bx false));; ret false)
10    | (|Or|)           $\Rightarrow$  trigger Or
11    | (||Choice|)      $\Rightarrow$  trigger Choice
12  end.
13
14 Definition observer_http: itree oE void :=
15   interp observe sm_http.

```

Figure 4.9: Dualizing symbolic model into symbolic observer.

```

(* Observer-side symbolic model's event type: *)
Notation smE := (netE  $\oplus$  nondetE  $\oplus$  choiceE  $\oplus$  branchE).

Definition sm_http: itree smE void :=
  compose server_http tcp [] [].

```

This corresponds to the “Symbolic Model” in Figure 4.4. The rest of this section will explain the interpretations from this symbolic model.

4.3.2. Dualizing interactions

This subsection takes the symbolic model composed in Subsection 4.3.1 and dualizes its interactions, which corresponds to the “Dualization” arrow in Figure 4.4. It applies the derivation rules (1)–(4) for **Prog** in Section 3.2 to models written as ITrees.

This interpretation phase produces a symbolic observer that models the tester’s observation and validation behavior. The observer sends a request when the server wants to receive one, and receives a response when the server wants to send one. It also creates constraints over the server’s internal choices based on its observations.

Figure 4.9 shows the dualization algorithm. It interprets the symbolic model’s events with the **observe** handler, whose types are explained as follows:

The tester observes a trace of concrete packets, so observer’s interactions return concrete requests and responses, as opposed to the symbolic model whose responses are symbolic.

```

Record concrete_packet := {
  source      : connection;
  destination : connection;
  payload     : request + concrete_response
}.

Variant observeE : Type → Type :=
  FromObserver   : observeE concrete_packet
| ToObserver    : observeE concrete_packet.

```

Notice that the observer’s send and receive events both return the packet sent or received, unlike the server model whose **Send** event takes the sent packet as argument. This is because the tester needs to generate the request packet to send, and the event’s result value represents that generated and sent packet.

As discussed in Section 3.2, when the server sends a symbolic response or branches over a symbolic condition, the tester needs to create symbolic constraints accordingly. The observer introduces “constraint events” for this derivation rule:

```

Variant constraintE : Type → Type :=
  Guard : packet → concrete_packet → constraintE unit
| Unify : exp bool → bool → constraintE unit.

```

Here (Guard *px p*) creates a constraint that the symbolic packet *px* emitted by the specification matches the concrete packet *p* observed during runtime. (Unify *bx b*) creates a constraint that the symbolic branch condition *bx* is unifiable with boolean value *b*. These constraints will be solved in Subsection 4.3.3.

The dualization algorithm in Figure 4.9 does the follows:

1. When the symbolic model absorbs a packet in Line 5, the observer generates a request packet;
2. When the symbolic model emits a symbolic packet *px* in Line 6, the observer receives a concrete packet *p*, and adds a constraint that restricts the symbolic and concrete packets match each other.

3. When the symbolic model branches on a symbolic condition `bx` in Line 8, the tester accepts the observation if it can be explained by any branch. This is done by constructing the observer as a nondeterministic program that has both branches, using the `or` combinator. For each branch, the observer adds a constraint that the symbolic condition matches the chosen branch.
4. Nondeterministic branches in Line 10 are preserved in this interpretation phase, and will be resolved in Section 4.4.
5. Internal choices in Line 11 are addressed by the next phase in Subsection 4.3.3, along with the constraints created in this phase.

The result of dualization is a symbolic observer that models the tester’s behavior like sending requests and receiving responses. The symbolic observer is a nondeterministic program with primitives events like making choices and adding constraints over the choices. The rest of this chapter instantiates the primitive events and resolves the nondeterministic branches, and executes it as an interactive tester.

4.3.3. Symbolic evaluation

This subsection takes the symbolic observer produced in Subsection 4.3.2 and solves the constraints it has created. The constraints unify symbolic packets and branch conditions against the concrete observations. The tester should accept the SUT if the constraints are satisfiable.

As shown in Figure 4.10, the unification algorithm evaluates the primitive symbolic events into a stateful checker program, which reflects the `Prog`-based validator in Section 3.2. The interpreter maintains a validation state `V` which stores the symbolic variables and the constraints over them. The derivation rules are as follows:

1. When the server makes an internal choice in Line 8, the tester creates a fresh variable and adds it to the validation state.

```

1 Notation ntE := (observeE  $\oplus$  nondetE  $\oplus$  exceptE).
2
3 Definition V: Type := list var * list (constraintE unit).
4
5 Definition unify {R} (e: oE R) (v: V) : itree ntE (V * R) :=
6   let (xs, cs) := v in
7   match e with
8   | (||Choice|)       $\Rightarrow$  let x: var := fresh v in
9                           ret (x::xs, cs, Var x)
10  | (|||constraint)  $\Rightarrow$  let cs' := constraint::cs in
11                          if solvable cs'
12                          then ret (xs, cs', tt)
13                          else Trigger (Throw ("Conflict: " ++ print cs'))
14  | (|Or|)  $\Rightarrow$  b  $\leftarrow$  trigger Or;; ret (v, b)
15  | (oe|)  $\Rightarrow$  r  $\leftarrow$  trigger oe;; ret (v, r)
16  end.
17
18 Definition nondet_tester_http: itree ntE void :=
19   (_, vd)  $\leftarrow$  interp_state unify observer_http initV;;
20   match vd in void with end.

```

Figure 4.10: Resolving symbolic constraints.

2. When the observer creates a constraint in Line 10, the tester adds the constraint to the validation state, and solves the new set of constraints. If the constraints become unsatisfiable, then the tester Throws an exception that indicates the current execution branch cannot accept the observations:

```

Variable exceptE: Type  $\rightarrow$  Type :=
  Throw:  $\forall \{X\}$ , string  $\rightarrow$  exceptE X.

```

3. The observer is a nondeterministic program with multiple execution paths, constructed by Or events in Line 14. The tester accepts the observation if any of the branches does not throw an exception. These branches will be handled in the next section, along with the observer's send/receive interactions in Line 15.

Notice that the `unify` function interprets a symbolic observer's event (`oE R`) into a state monad transformer ($V \rightarrow \text{itree } \text{tE } (V * R)$). It makes a step from pre-validation state ($v: V$) to post-validation state ($v': V$), and yields the event's corresponding result ($r: R$). Such stateful interpretation process is handled by `interp_state`:

```

CoFixpoint interp_state {E F V R}
  (handler:  $\forall \{X\}$ , E X  $\rightarrow$  V  $\rightarrow$  itree F (V * X))

```

```

(m: itree E R) (v: V)
: itree F (V * R) :=
match m with
| Pure   r   => ret (v, r)
| Impure e k => '(v', r) <- handler e v;;
            interp_state handler (k r) v'
end.

```

So far I have interpreted the specification into a tester model that observes incoming and outgoing packets, nondeterministically branches, and in some cases throws exceptions. The rest of this chapter will show how to execute this ITree program on a deterministic machine and interact with the SUT.

4.4. Executing Tester Model

This section takes the nondeterministic tester model derived in Subsection 4.3.3 and transforms it into an interactive program. Subsection 4.4.1 handles the nondeterministic branches via backtrack execution, and produces a deterministic tester model. Subsection 4.4.2 then interprets the deterministic tester into IO program that interacts with the SUT.

4.4.1. Backtrack execution

This subsection explains how to run the nondeterministic tester on a deterministic machine. It reflects the derivation rules (5) and (6) for **Prog** in Section 3.2, and constructs the “Backtracking” arrow in Figure 4.4.

The deterministic tester implements a client that sends and receives concrete packets:

```

Variant clientE: Type → Type :=
  ClientSend: concrete_packet → clientE unit
| ClientRecv: clientE (option concrete_packet).

```

Notice that the **ClientRecv** event might return (**Some** *pkt*), indicating that the SUT has sent a packet *pkt* to the tester; or it might return **None**, when the SUT is silent or its sent packet hasn’t arrived at the tester side. This allows the tester to perform non-blocking interactions, instead of waiting for the SUT which might cause starvation.

Figure 4.11 shows the backtracking algorithm. It interacts with the SUT and checks whether

```

1  Notation tE := (clientE  $\oplus$  genE  $\oplus$  exceptE).
2
3  CoFixpoint backtrack (current:      itree ntE void)
4                        (others: list (itree ntE void))
5                        : itree tE void :=
6    match current with
7    | Impure e k  $\Rightarrow$ 
8      match e with
9      | (|Or|)  $\Rightarrow$  b  $\leftarrow$  trigger GenBool;;
10                 backtrack (k b) (k (negb b)::others)
11      | (|Throw msg)  $\Rightarrow$  match others with
12                 | other::ot'  $\Rightarrow$  backtrack other ot'
13                 | []  $\Rightarrow$  trigger (Throw msg)
14                 end
15      | (FromObserver|)  $\Rightarrow$  q  $\leftarrow$  trigger GenPacket;;
16                 trigger (ClientSend q);;
17                 let others' := expect FromObserver q others in
18                 backtrack (k q) others'
19      | (ToObserver|)  $\Rightarrow$ 
20                 oa  $\leftarrow$  trigger ClientRecv;;
21                 match oa with
22                 | Some oa  $\Rightarrow$  let others' := expect ToObserver a others in
23                 backtrack (k a) others'
24                 | None  $\Rightarrow$ 
25                 match others with
26                 | other::ot'  $\Rightarrow$  backtrack other (ot'++[current]) (* postpone *)
27                 | []  $\Rightarrow$  backtrack m [] (* retry *)
28                 end
29                 end
30      end
31    | Pure vd  $\Rightarrow$  match vd in void with end
32  end.
33
34 Definition tester_http: itree tE void :=
35   backtrack nondet_tester_http [].

```

Figure 4.11: Backtrack execution of nondeterministic tester.

```

1  CoFixpoint match_observe {R} (e: observeE R) (r: R)
2      (m: itree ntE (V * void))
3      : itree ntE (V * void) :=
4      match m with
5      | Impure (oe|) k =>
6          match oe, e with
7          | FromObserver, FromObserver
8          | ToObserver , ToObserver => k r
9          | _, _ => trigger (Throw ("Expect " ++ print oe
10                               ++ " but observed " ++ print e))
11      end
12      | Impure (|e0|) k | Impure (||e0) k =>
13          r0 <- trigger e0;;
14          match_observe e r (k r0)
15      | Pure (_, vd) => match vd in void with end
16      end.
17
18  Definition expect {R} (e: observeE R) (r: R)
19      : list (itree ntE (V * void)) -> list (itree ntE (V * void))
20      := map (match_observe e r).

```

Figure 4.12: Matching tester model against existing observation.

the observations can be explained by the nondeterministic tester model. That is, checking whether the tester has an execution path that matches its interactions. This is done by maintaining a list of all possible branches in the tester, and checking if any of them accepts the observation.

The tester exhibits two kinds of randomness: (1) When sending a request packet to the SUT, it generates the packet randomly with `GenPacket`; (2) When the nondeterministic tester model branches, the deterministic tester randomly picks one branch to evaluate, using `GenBool`:

`GenBool`:

```

Variant genE: Type -> Type :=
  GenPacket : genE concrete_packet
  | GenBool   : genE bool.

```

The execution rule is defined as follows:

1. When the tester nondeterministically branches in Line 9, randomly pick a branch ($k \ b$) to evaluate, and push the other branch ($k \ (\text{negb } b)$) to the list of other possible cases.

2. When the **current** tester throws an exception in Line 11, it indicates that the current execution path rejects the observations. The tester should try to explain its observations with other branches of the tester model. If the **others** list is empty, it indicates that the observation is beyond the specification's producible behavior, so the tester should reject the SUT.
3. When the tester wants to observe a packet *from* itself, it generates a packet and sends it to the SUT in Line 16.

Notice that if the current branch is rejected and the tester backtracks to other branches, the sent packet cannot be recalled from the environment. Therefore, all other branches should be matched against this send event as well. This is done by the **expect** function.

As shown in Figure 4.12, (**expect** **e** **r** **l**) matches every tester in list **l** against the observation **e** that has return value **r**. For each element **m** \in **l**, if **m**'s first observer event **oe** matches the observation **e** (Line 7 and Line 8), then **match_observe** instantiates the tester's continuation function **k** with the observed result **r**. Otherwise, the tester throws an exception in Line 9, indicating that this branch cannot explain the observation because they performed different events.

4. When the current tester wants to observe a packet *to* itself, it triggers the **ClientRecv** event in Line 20. If a packet has indeed arrived, then it instantiates the current branch as well as other possible branches, in the same way as discussed in Rule (3).

If the tester hasn't received a packet from the SUT (Line 24), it doesn't reject the SUT, because the expected packet might be delayed in the environment. If there are **other** branches to evaluate (Line 26), then the tester postpones the **current** branch by appending it to the back of the queue. Otherwise, if the current branch is the only one that hasn't rejected, then the tester retries the receive interaction.

Notice that if the SUT keeps silent, then the tester will starve but won't reject, because

(i) such silence is indistinguishable from the SUT sending a packet but delayed by the environment, and (ii) the SUT hasn't *exhibited* any violations against the specification. The starvation issue is addressed in Subsection 4.4.2.

The backtracking algorithm also explains the network composition design in Figure 4.7, where the server model is scheduled at a higher priority than the network model: Suppose the SUT has produced some invalid output, then every branch of the tester should reject its observation by throwing an exception. However, the network model is always ready to absorb packets. Evaluating the network model lazily prevents the composed symbolic model from having infinitely many absorbing branches. This allows the derived tester to converge to rejection upon violation.

Now we have derived the specification into a deterministic tester model in ITree. The tester's events reflect actual computations of a client program. In the next subsection, I'll translate the ITree model into a binary executable that runs on silicon and metal.

4.4.2. From ITree model to IO program

The deterministic tester model derived in Figure 4.11 is an ITree program that never returns (its result type `void` has no elements). It represents a client program that keeps interacting with the SUT until it reveals a violation and throws an exception.

In practice, if the tester hasn't found any violation after performing a certain amount of interactions, then it accepts the SUT. This is done by executing the ITree until reaching a certain depth.

As shown in Figure 4.13, the `execute` function takes an argument `fuel` that indicates the remaining depth to explore in the ITree. If the execution ran out of fuel (Line 3), then the test accepts; If the tester model throws an exception (Line 8), then the test rejects. Otherwise, it translates the ITree's primitive events into IO computations in Coq [LYS: Cite Li-yao's SimpleIO library], which are eventually extracted into OCaml programs that can be compiled into executables that can communicate with the SUT over the operating system's

```

1  Fixpoint execute (fuel: nat) (m: itree tE void) : IO bool :=
2    match fuel with
3    | 0      ⇒ ret true                                (* accept if out of fuel *)
4    | S fuel' ⇒
5      match m with
6      | Impure e k ⇒
7        match e with
8        | (|Throw _)      ⇒ ret false (* reject upon exception *)
9        | (ClientSend q|) ⇒ client_send q;;
10                               execute fuel' (k tt)
11        | (ClientRecv|)   ⇒ oa ← client_recv;;
12                               execute fuel' (k oa)
13        | (|GenPacket|)   ⇒ pkt ← gen_packet;;
14                               execute fuel' (k pkt)
15        | (|GenBool|)     ⇒ b ← ORandom.bool;;
16                               execute fuel' (k b)
17        end
18      | Pure vd ⇒ match vd in void with end
19    end
20  end.
21
22  Definition test_http: IO bool :=
23    execute bigNumber tester_http.

```

Figure 4.13: Interpreting ITree tester to IO monad.

network stack.

This concludes my validation methodology. In this chapter, I have shown how to test real-world systems that exhibit internal and external nondeterminism. I applied the dualization theory in Chapter 2 to address internal nondeterminism, and handled external nondeterminism by specifying the environment’s space of uncertainty. The specification is derived into an executable tester program, by multiple phases of interpretations. The derivation framework is built on the ITree specification language, but the method is applicable to other languages that allow destructing and analyzing the model programs.

So far I have answered “how to tell compliant implementations from violating ones”. The next chapter will answer “how to generate and shrink test input that reveal violations effectively”, and unveil the techniques behind `gen_packet` in Line 13 of Figure 4.13.

CHAPTER 5

TEST HARNESS DESIGN

A tester consists of a validator and a test harness. Chapters 2 and 4 have explained the validator’s theory and practices. This chapter presents a language-based design for test harnesses. I’ll show how to generate and shrink test inputs effectively, addressing inter-execution nondeterminism.

Section 5.1 provides a brief overview of how test harnesses work. Section 5.2 explains how to write heuristics to generate interesting test inputs. Section 5.3 then shows how to keep the test inputs interesting among different executions in the shrinking process.

5.1. Overview

This section introduces the abstract architecture of an interactive tester, using the networked server as an example. I’ll present a naïve implementation of the test harness, which will be improved in the following sections.

The test harness interacts with the environment and provides the observations for the validator. The validator may represent requests and responses as abstract datatypes for the convenience of specification. The test harness translates these abstract representations into bytes transmitted on the underlying channel.

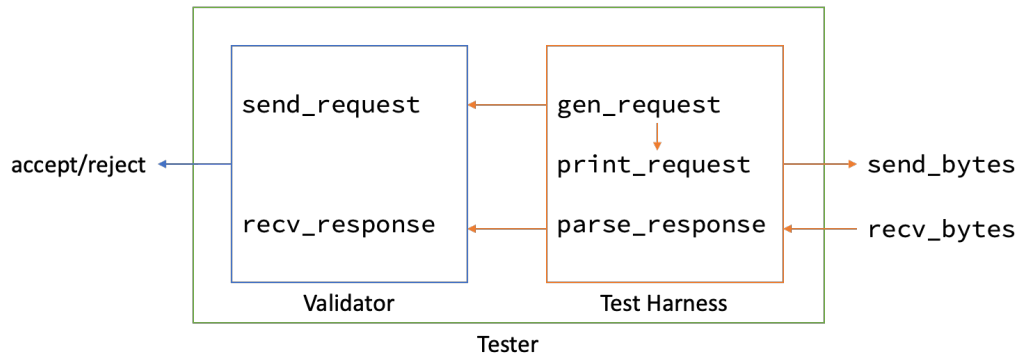


Figure 5.1: Tester architecture outline.

```

1 Definition gen_packet: IO concrete_packet :=
2   src      ← random_conn;;
3   method   ← oneof [Get; Put];;
4   target    ← random_path;;
5   precondition ← oneof [IfMatch, IfNoneMatch];;
6   etag      ← random_etag;;
7   payload   ← random_string;;
8   ret { Source      := src;
9         Destination := server_conn;
10        Data        := inr { Method      := method;
11                             TargetPath  := target;
12                             Headers     := [(precondition, etag)];
13                             Payload     := payload
14                           }
15   }.

```

Figure 5.2: Naïve generator for HTTP requests.

As shown in Figure 5.1, when the validator wants to observe a sent request, the harness generates the request and encodes it into bytes to send. Conversely, when the validator wants to observe a received response, the harness receives bytes from the environment and decodes it into abstract messages.

The generator is a randomized program that produces test inputs. One example is the `gen_packet` function in Figure 4.13. The HTTP packets generator can be naïvely implemented as shown in Figure 5.2. It fills in the request’s fields with arbitrary values, and has limited coverage of the SUT’s behavior. This is because the request target and ETags are both generated randomly, and thus unlikely to hit the server’s resource, resulting in 404 Not Found and 412 Precondition Failed in almost all cases.

To reveal more interesting behavior from the SUT, we should tune the generator’s distribution to emphasize certain patterns of the test input. For example, if the tester knows what paths have the server stored resources, then it can generate more paths that correspond to existent resources; if the tester has observed some ETags generated by the server, then it can include these ETags in future requests. In the next section, I’ll explain how to implement such heuristics in ITree-based testers.

5.2. Heuristics for Test Generation

This section implements heuristics for generating test inputs. I'll use the HTTP tester as an example to show how to make requests more interesting, by parameterizing them over the model state (Subsection 5.2.1) and the trace (Subsection 5.2.2).

5.2.1. State-based heuristics

The model state may instruct the test generator to produce more interesting test inputs. For example, consider the `random_path` generator in Line 4 of Figure 5.2. One way to improve it is to generate more paths that have corresponding resources on the server:

```
Definition gen_path (state: list (path * resource)) : IO path :=  
  let paths: list path := map fst state in  
  freq [(90, oneof paths);  
        (10, random_path)].
```

Here I modify the server model's state type σ from $(\text{path} \rightarrow \text{resource})$ in Figure 4.8 into $(\text{list } (\text{path} * \text{resource}))$, which has the same expressiveness but allows the generator to access the list of all `paths` in the server state. The generator chooses from these existent paths in 90% of the cases, as assigned by the `freq` combinator. The remaining 10% are still generated randomly, to discover how the SUT handles nonexistent paths.

For the `gen_packet` generator in Figure 5.2, replacing its `random_path` with the improved `gen_path` would generate more interesting request targets. This requires the `gen_packet` to carry the server state to instantiate `gen_path`.

As shown in Figure 4.11, the `GenPacket` generator is triggered when the tester wants to observe a packet from itself to the SUT. `fig:symbolic-observer` then shows that such `FromObserver` expectation happens when the symbolic model `Emits` a packet. Such `Emit` event only happens when the server wants to receive a packet in Figure 4.6. The `Recv` events are triggered by the server model in Figure 4.8, which iterates over the server state σ .

Therefore, I extend the server's `Recv` event type to include the server state:

```
Variant qaE: Type → Type :=  
  Recv :  $\sigma$  → qaE packet
```

```
| Send : packet → qaE unit.
```

Now when the server wants to receive a request, it triggers (Recv state), where (state: σ) contains the server's paths and resources at that point. The state argument is then carried to the generator, by adding parameters to the event types along the interpretation:

```
Variant netE: Type → Type :=
  Emit  : packet → netE unit
| Absorb:  $\sigma$       → netE packet.

Variant observeE : Type → Type :=
  FromObserver  :  $\sigma$  → observeE concrete_packet
| ToObserver    : observeE concrete_packet.

Variant genE: Type → Type :=
  GenPacket :  $\sigma$  → genE concrete_packet
| GenBool   : genE bool.

Definition gen_packet:  $\sigma$  → IO concrete_packet.
```

As a result, when instantiating the (GenPacket state) event in Figure 4.13, we can feed the gen_packet function with argument state, so that gen_path can generate interesting paths based on the server state.

5.2.2. Trace-based heuristics

When the SUT makes internal choices *e.g.* generating ETags, the specification represents them as symbolic variables. These variables' concrete value are not stored in the specification state, but may be observed during execution. For example, when an HTTP server responds to a GET request, it might include the resource's ETag as shown in Subsection 1.2.1.

To improve the generator in Figure 5.2, we can generate interesting ETags based on the trace produced during execution. The trace is a list of packets sent and received by the tester, and the packets' payloads may include responses that have an ETag field. The gen_etag function emphasizes ETags that were observed in the trace, which are more likely to match those generated by the SUT:

```
Definition gen_etag (trace: list concrete_packet) : IO string :=
  let etags: list string := tags_of trace in
  freq [(90, oneof etags);
        (10, random_etag)].
```

To utilize this improved generator for ETags, the tester needs to record the trace of packets sent and received. This is done by modifying the `execute` function in Figure 4.13, adding an accumulator as the recursion parameter:

```

Fixpoint execute (fuel: nat) (trace: list concrete_packet)
  (m: itree tE void) : IO bool :=
  match fuel with
  | S fuel' =>
    match m with
    | Impure e k =>
      match e with
      | (ClientSend q|) => client_send q;;
                          execute fuel' (trace ++ [q]) (k tt)
      | (ClientRecv|)  => oa ← client_recv;;
                          let trace' := match oa with
                              | Some a => trace ++ [a]
                              | None   => trace
                          end in
                          execute fuel' trace' (k oa)
      | (|GenPacket state|) => pkt ← gen_packet state trace;;
                          execute fuel' trace (k pkt)
  ... (* similar to Figure 4.13 *)

```

When the tester sends or receives a packet, the packet is appended to the runtime trace. Then the `gen_packet` generator can take the trace accumulated so far, and feed it to the ETag generator:

```

Definition gen_packet (state:  $\sigma$ ) (trace: list concrete_packet) :=
  target ← gen_path state;;
  etag   ← gen_etag trace;;
  ... (* same as Figure 5.2 *)

```

Now I've shown how to generate interesting test inputs by implementing state-based and trace-based heuristics. The next section explains how to shrink the test inputs while keeping them interesting, addressing inter-execution nondeterminism.

5.3. Shrinking Interactive Tests

Suppose we have generated a test input that has caused invalid observations of the SUT. The generated counterexample consists of (1) *signals* that are essential to triggering violations, and (2) *noises* that do not contribute to revealing such violations. We need to shrink the counterexample by removing its noises and keeping its signals.

For interactive testing, the test input is a sequence of request messages. An intuitive way of shrinking is to remove some requests from the original sequence and rerun the test. However, rerunning an interesting request might produce trivial results, due to inter-execution nondeterminism discussed in Subsection 1.3.2.

To prevent turning signals into noises when rerunning the test, I shrink the heuristics instead of shrinking the generated test input. Subsection 5.3.1 introduces the architecture for interactive shrinking, then Subsection 5.3.2 explains the language design beneath that addresses inter-execution nondeterminism.

5.3.1. Architecture

I propose a generic framework for generating and shrinking interactive tests. The key idea is to introduce an abstract representation for test inputs that embed trace-based heuristics. When shrinking the counterexample, the test harness picks a substructure of the abstract representation, and computes the corresponding test input using the new runtime trace.

For example, when generating a timestamp, instead of producing concrete value *e.g.* “Tue, 17 May 2022 07:44:03 GMT”, the generator returns an abstract representation that says “use the timestamp observed in the last response”. When rerunning the test, the timestamp is computed from the new trace *e.g.* “ Wed, 18 May 2022 07:44:03 GMT”.

The test generation and shrinking framework is shown in Figure 5.3. It refines the test harness box in Figure 5.1, and involves four languages, from right to left:

1. Byte representation, in which the tester interacts with the environment. This can be network packets, file contents, or other serialized data produced and observed by the tester.
2. Intermediate representation (IR), a generic language that abstracts the byte representation as structured data. The test harness *parses* byte observations and records its trace in terms of the IR, which allows representing trace-based heuristics as a generic language *i.e.* J-expressions.

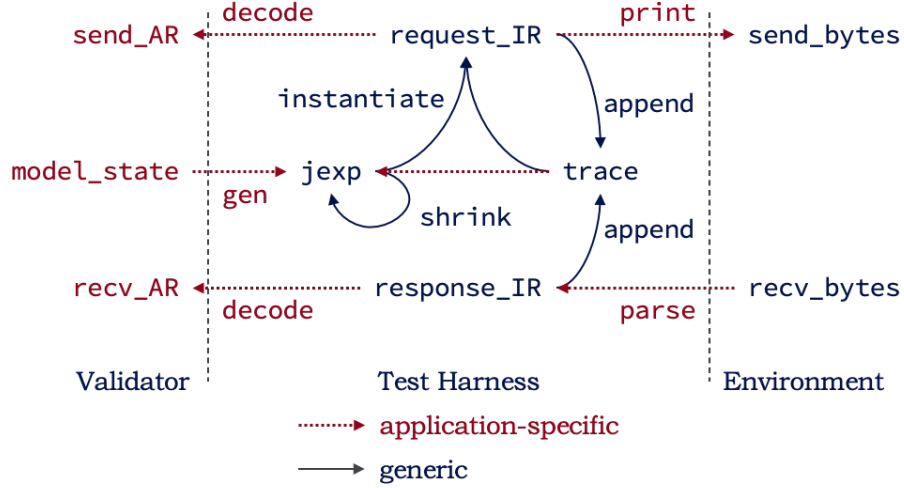


Figure 5.3: Test harness architecture.

3. J-expression (Jexp), a symbolic abstraction of the IR. The IR corresponds to concrete inputs and outputs, whereas Jexp defines a computation from trace to IR. The generator provides test inputs in terms of Jexps; The test harness *instantiates* the generated Jexps into request IR, and *prints* them into byte representation.

When shrinking test inputs, the test harness shrinks the sequence of Jexps. The shrunk Jexps are then instantiated by the new trace during runtime.

The intermediate representation and J-expression will be further explained in Subsection 5.3.2.

4. Application representation (AR), including the request (Q), response (A), and state (S) types used for specifying the protocol. Specification writers can choose the type interface at their convenience, provided the request and response types are embeddable into the IR.

The testing framework implements protocol-independent mechanisms like recording the trace and shrinking counterexamples, based on the generic IR and Jexp languages. It can be used for testing various protocols, provided application-specific translations from IR to AR and between IR and bytes. The test developer needs to tune the generator that produces Jexps,

$$\begin{aligned}
\text{JSON}^T &\triangleq T \mid \{\text{object}^T\} \mid [\text{array}^T] \mid \text{string} \mid \mathbb{Z} \mid \mathbb{B} \mid \text{null} \\
\text{object}^T &\triangleq \varepsilon \mid \text{"string"} : \text{JSON}^T, \text{object}^T \\
\text{array}^T &\triangleq \varepsilon \mid \text{JSON}^T, \text{array}^T \\
\text{IR} &\triangleq \text{JSON}^{\text{IR}} \\
\text{Jexp} &\triangleq \text{JSON}^{\text{label.Jpath.function}} \\
&\quad \text{where } \text{label} \in \mathbb{N}, \text{function} \in \text{IR} \rightarrow \text{IR} \\
\text{Jpath} &\triangleq \text{this} \mid \text{Jpath}\# \text{index} \mid \text{Jpath}@ \text{field} \\
&\quad \text{where } \text{index} \in \mathbb{N}, \text{field} \in \text{string}
\end{aligned}$$

Figure 5.4: Intermediate representation and J-expression.

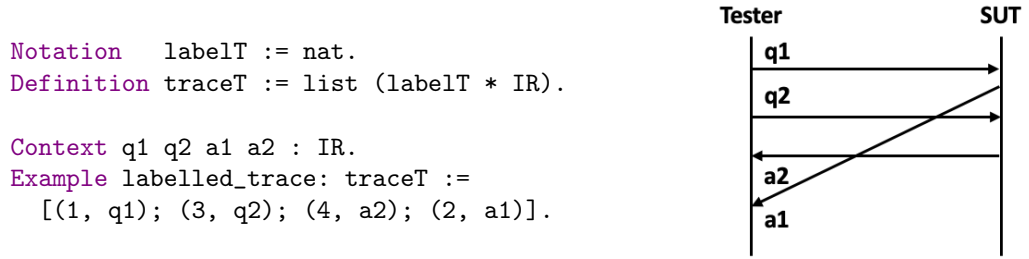


Figure 5.5: Labelled trace example.

encoding their domain knowledge as state-based and trace-based heuristics.

5.3.2. Abstract representation languages

I choose JSON as the IR in this framework, which allows syntax trees to be arbitrarily wide and deep, and provides sufficient expressiveness for encoding message data types in general.

The J-expression is an extension of JSON that can encode trace-based heuristics. As shown in Figure 5.4, a Jexp may include syntax $(\text{label}.\text{Jpath}.\text{function})$ that represents trace-based heuristics, specified as:

1. The *label* refers to an IR in the trace that the heuristics computes with. The test harness records the trace as a list of labelled messages, where requests are labelled odd, and their responses are labelled as the next even number. Labelling messages allows locating them in the trace despite shrinking and inter-execution nondeterminism.

For example, consider the trace in Figure 5.5: If a trace-based heuristics is interested in

```

(* a2 = *)
{
  "files": [
    {
      "name": "foo",
      "mode": 755
    },
    {
      "name": "bar",
      "mode": 500
    }
  ],
  "exitCode": 0
}

Example second_file_mode: jpath :=
  this @ "files" # 2 @ "mode".

Example mode_add_write (j: IR) : IR :=
  match j with
  | JSON_Number n =>
    JSON_Number (mode_bits_or 200 n)
  | _ => j
  end.

Example id (j: IR) : IR := j.

```

Figure 5.6: IR, Jpath, and heuristics function example.

q2's response a2, then it can be encoded as "compute the test input based on message labelled 4":

```

Context get_label: labelT → traceT → IR.

Compute get_label 4 labelled_trace.
(* = a2 : IR *)

```

Suppose the test input is shrunk by removing q1, the label for q2 remains unchanged as 3, so label 4 corresponds to the new response to q2:

```

Example new_trace: traceT :=
  [(3, q2); (4, a2')].

Compute get_label 4 new_trace.
(* = a2' : IR *)

```

As a result, the trace-based heuristics are preserved and adapted to new executions during the shrinking process.

2. The **Jpath** is a path in the IR's syntax tree, and refers to a substructure of the IR that the heuristics uses.

For example, suppose request q2 lists files in a directory using the POSIX `ls` command, and its response a2 is encoded as the IR shown in Figure 5.6. The response IR is a JSON object whose **"files"** field is an array of objects, each has a **"name"**

and a "mode" field. A heuristics can refer to the second file's mode bits by Jpath (this@"files"#2@"mode"), which will guide the test harness to locate its corresponding value:

```
Context get_jpath: jpath → IR → IR.

Compute get_jpath second_file_mode a2.
(* = JSON_Number 500 : IR *)
```

3. The *function* has type $(IR \rightarrow IR)$, and defines the computation based on the sub-IR located by the Jpath.

Consider the mode bits located in the previous example: If the heuristics wants to add write permission to the mode bits, it can do so with the `mode_add_write` function in Figure 5.6, which produces mode 700. Some heuristics might use the sub-IR 500 as-is, using the identity function `id`.

J-expression provides a generic interface for test developers to implement trace-based heuristics. For the aforementioned file system example, the tester can generate a request that changes the mode bits of an observed file, with the following Jexp:

```
(* e5 = *)
{
  "command": "chmod",
  "args": [
    4.(this@"files"#2@"mode").mode_add_write,
    4.(this@"files"#2@"name").id
  ]
}
```

To instantiate Jexps into request IR, the test harness substitutes all occurrences of $(l.p.f)$ in the Jexp with its corresponding IR computed from the runtime trace:

```
Definition eval (l: labelT) (p: jpath) (f: IR → IR) (t: traceT) : IR :=
  let a: IR := get_label l t in
  let j: IR := get_jpath p a in
  f j.
```

For example, given the runtime trace in Figure 5.5, with `a2` is defined in Figure 5.4, the the above Jexp is instantiated into the following request:

```
(* instantiate e5 labelled_trace = *)
```

```
{
  "command": "chmod",
  "args": [ 700, "bar" ]
}
```

However, when rerunning the test, the `new_trace` has a different response associated with label 4. The new response `a2'` might have fewer than 2 files in its payload. Moreover, the response `a2'` might have not appeared in the trace, due to delays in the environment.

To instantiate the original Jexp in such situations, I loosen the `get_jpath` and `get_label` functions when evaluating the heuristics:

1. When evaluating a Jpath starting with `p#n`, if `p` corresponds to an array with fewer than `n` elements, or the array's `n`-th element cannot properly evaluate the remaining path, then try continuing the evaluation with any other element in the array.

For example, consider evaluating `(this@3#"bar")` on the following IR's:

<pre>(* j2 = *) [{ "foo": 21 }, { "bar": 22 }]</pre>	<pre>(* j3 = *) [{ "bar": 31 }, { "baz": 32 }, { "foo": 33 }]</pre>
--	---

Here `j2` doesn't have a third element, and `j3`'s third element doesn't have field `"bar"`. In these cases, `get_jpath` chooses other elements in the two arrays, resulting in value 22 for `j2`, and 31 for `j3`.

2. When evaluating label 1 and Jpath `p` on a trace, if the message labelled 1 does not exist in the trace, or cannot evaluate Jpath `p` properly, then try continuing the evaluation with any other IR in the trace.

For example, consider evaluating J-expression `6.(this#2@"foo").id` on the following traces:

```
Definition t1: traceT :=
  [(1,q1); (2,j2); (5,q2)].
```

Definition $\mathbf{t2: traceT := [(3,q1); (4,j3); (5,q2); (6,a2)]}$.

Here $\mathbf{t1}$ doesn't have a message labelled 6, probably caused by environment delays; $\mathbf{t2}$ has label 6 but its corresponding message is an object rather than an array expected by the Jexp. In these cases, `eval` chooses other messages in the trace to evaluate, resulting in value 21 for $\mathbf{t1}$, and 33 for $\mathbf{t2}$.

By introducing loose evaluation of J-expressions, my test harness allows partial instantiation of heuristics when the runtime trace is less than satisfying.

So far I have shown how to generate and shrink interactive test inputs and address inter-execution nondeterminism. In the next chapter, I'll combine this test harness design with the validator practice in Chapter 4, and evaluate these techniques by testing real-world systems like HTTP servers and file synchronizers.

CHAPTER 6

EVALUATION

This chapter evaluates the testing methodology presented in this thesis, by deriving testers from specifications and running them against systems under test.

I conduct the experiments on two kinds of systems, HTTP/1.1 server (Section 6.1) and file synchronizer (Section 6.2). The research questions are: (1) Qualitative: Is the tester useful to finding invalid behavior in the SUT? and (2) Quantitative: Can the tester reveal those invalid behavior quickly?

6.1. Testing Web Servers

This thesis is motivated by the Deep Specification project [3], whose goal is to build systems with rigorous guarantee of functional correctness, studying HTTP as an example. I formalized a subset of HTTP/1.1 specification, featuring WebDAV requests GET, PUT, and POST [11], ETag preconditions [14], and forward proxying [13].

From the protocol specification written as ITrees, I derived a tester client that sends and receives network packets. Subsection 6.1.1 explains the system under test and the experiment setup. Subsection 6.1.2 and Subsection 6.1.3 then describe the evaluation results qualitatively and quantitatively.

6.1.1. Systems Under Test

I ran the derived tester against three server implementations:

- Apache HTTP Server [12], which is among the most popular servers on the World Wide Web [28, 38]. I used the latest release 2.4.46, and edited its configuration file to enable WebDAV and proxy modules.
- Nginx [33], the other most popular server. The experiment was conducted on the latest release 1.19.10, with only WebDAV module enabled, because Nginx doesn't

fully support forward proxying like Apache does.

- DeepWeb server developed in collaboration with Zhang et al. [41], supporting GET and POST requests. The server’s functional correctness was formally verified, showing our methodology of specifying systems in testable and verifiable ways.

The tests were performed on a laptop computer (with Intel Core i7 CPU at 3.1 GHz, 16GB LPDDR3 memory at 2133MHz, and macOS 10.15.7). The Apache and Nginx servers were deployed as Docker instances, using the same host machine as the tester runs on. Our simple server was compiled into an executable binary, and also ran on localhost.

The tester communicates with the server via POSIX system calls, in the same way as over Internet except using address 127.0.0.1. The round-trip time (RTT) of local loopback is 0.08 ± 0.04 microsecond (at 90% confidence).

6.1.2. Qualitative Result

Apache My tester rejected the Apache HTTP Server, which uses strong comparison for PUT requests conditioned over `If-None-Match`, while RFC 7232 specified that `If-None-Match` preconditions must be evaluated with weak comparison. [BCP: What are strong and weak comparison? [LYS: ETag jargons.]] I reported this bug to the developers, and figured out that Apache was conforming with an obsoleted HTTP/1.1 standard [15]. The latest standard has changed the semantics of `If-None-Match` preconditions, but Apache didn’t update the logic correspondingly.

To further evaluate the tester’s performance in finding other violations, I fixed the precondition bug by deleting 13 lines of source code and recompiling the container.

The tester accepted the fixed implementation, which can be explained in two ways: (1) The server now complies with the specification, or (2) The server has a bug that the tester cannot detect. To provide more confidence that (1) is the case, I ran the tester against servers with

known bugs, created by mutating the Apache source code manually.¹

The tester rejected all 20 bugs, located in various modules of the Apache server: `core`, `http`, `dav`, and `proxy`. They appear both in control flow (*e.g.*, early return of handler function, skipped precondition check) and in data values (*e.g.*, calling functions with wrong parameters, flipping bits in computations, accessing buffer off by one byte). Some mutants took the tester longer time to reveal than others, which will be discussed in Subsection 6.1.3.

Nginx When testing Nginx, I found that its WebDAV module did not check the `If-Match` and `If-None-Match` preconditions of PUT requests. I then browsed the Nginx bug tracker and found a ticket opened by Haverbeke [18], reporting the same issue with `If-Unmodified-Since` preconditions.

This issue has been recognized by the developers in 2016 but never resolved. One possible reason is that the current architecture of Nginx makes it difficult for the WebDAV module to utilize the precondition checker in the HTTP core module.

A fix to this bug would involve either restructuring the modules or copying source code from one module to another, unlike simply removing bad code in Apache, so I only tested mutants for the latter.

DeepWeb My test derivation framework was developed in parallel with the DeepWeb server. After my collaborators finished the formal proof of the server’s functional correctness, I tested the server with my derived tester. The tester has revealed a liveness issue—when a client pipelines more than one requests in a single connection, the server may hang without processing the latter requests.

This liveness bug was out of scope for the server’s functional correctness, which only requires the server not to send invalid messages. Such partial correctness may be trivially satisfied by a silent implementation that never responds. My experiments has shown this limitation

¹I didn’t use automatic mutant generators because (i) Existing tools could not mutate all modules I’m interested in; and (ii) The automatically generated mutants could not cause semantic violations against my protocol specification.

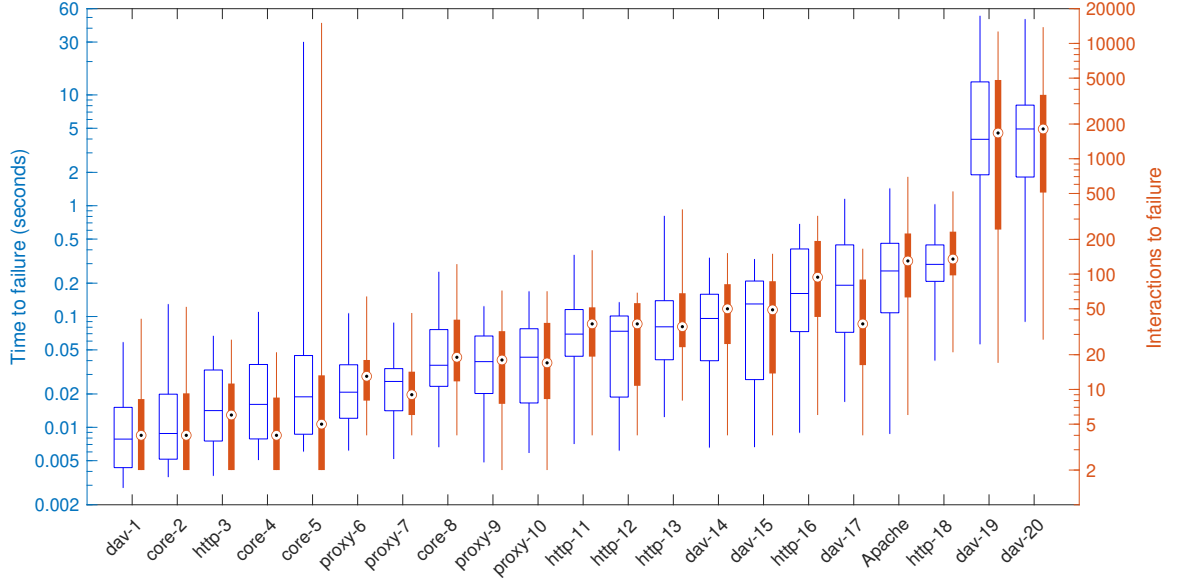


Figure 6.1: Cost of detecting bug in each server implementation. The left box with median line is the tester’s execution time before rejecting the server, which includes interacting with the server and checking its responses. The right bar with median circle is the number of HTTP/1.1 messages sent and received by the tester before finding the bug. Results beyond 25%–75% are covered by whiskers.

in formal verification, and guided the debugging of the server implementation.

These results show that my tester is capable of finding different kinds of bugs in server implementations, within and beyond functional correctness. Next I’ll evaluate how long the tester takes to reveal bugs.

[LYS: Under construction:]

6.1.3. Quantitative Result

To answer Research Question (2) at the beginning of this chapter, I measure the execution time and network interactions taken to reject vanilla Apache and its mutants, as shown in Figure 6.1. The 20 mutants are named after the module where I inserted the bug. The tester rejected all the buggy implementations within 1 minute, and in most cases, within 1 second. This does not include the time for shrinking the counterexample.

Some bugs took longer time to find, and they usually required more interactions to reveal.

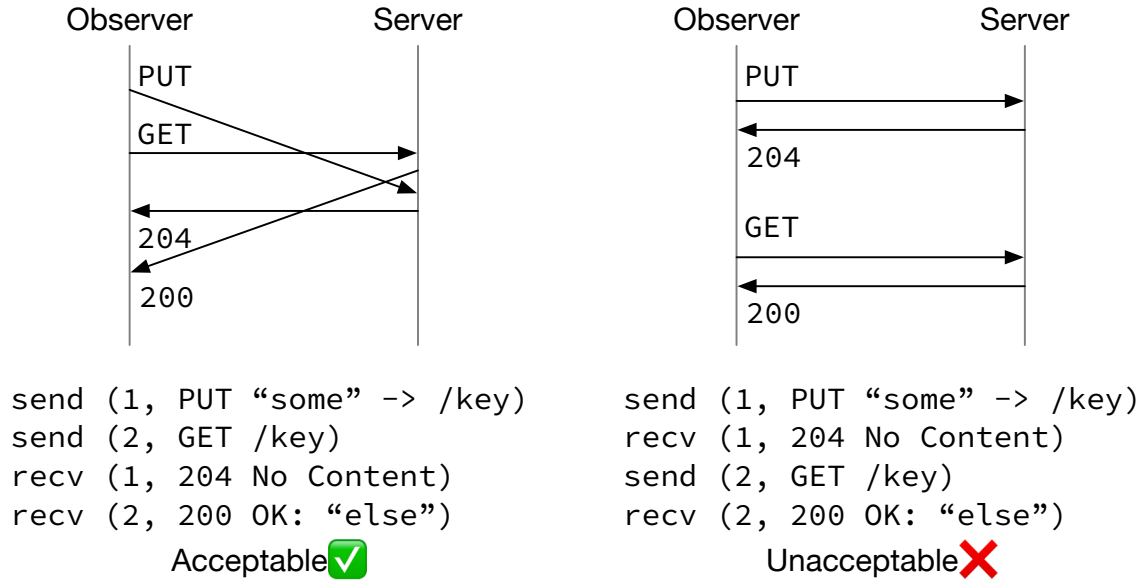


Figure 6.2: GET-after-PUT bug pattern in Apache mutants. The trace on the left does not convince the tester that the server is buggy, because there exists a certain network delay that explains why the PUT request was not reflected in the 200 response. When the trace is ordered as shown on the right, the tester cannot imagine any network reordering that causes such observation, thus must reject the server.

This may be caused by (1) The counter-example has a certain pattern that my generator didn't optimize for, or (2) The tester did produce a counter-example, but failed to reject the wrong behavior. I determine the real cause by analyzing the bugs and their counterexamples:

- Mutants 19 and 20 are related to the WebDAV module, which handles PUT requests that modify the target's contents. The buggy servers wrote to a different target from that requested, but responds a successful status to the client.

The tester cannot tell that the server is faulty until it queries the target's latest contents and observes an unexpected value. To reject the server with full confidence, these observations must be made in a certain order, as shown in Figure 6.2.

- Mutant 18 is similar to the bug in vanilla Apache: the server should have responded with 304 Not Modified, but sent back 200 OK instead. To reveal such violation, a minimal counterexample consists of 4 messages:

1. GET request,
2. 200 OK response with some ETag "x",
3. GET request conditioned over `If-None-Match: "x"`, and
4. 200 OK response, indicating that the ETag "x" did not match itself.

Notice that (2) must be observed before (3), otherwise the tester will not reject the server, with a similar reason as Figure 6.2.

- Mutant 5 causes the server to skip some code in the core module, and send nonsense messages when it should respond with 404 Not Found. The counterexample can be as small as one GET request on a non-existent target, followed by a non-404, non-200 response. However, my tester generates request targets within a small range, so the requests' targets are likely to be created by the tester's previous PUT requests.

Narrowing the range of test case generation might improve the performance in aforementioned Mutants 18–20, but Mutant 5 shows that it could also degrade the performance of finding some bugs.

- The mutants in proxy module caused the server to forward wrong requests or responses.

To test servers' forward proxying functionality, the tester consists of clients and origin servers, both derived by dualization. When the origin server part of the tester accepts a connection from the proxy, it does not know for which client the proxy is forwarding requests. Therefore, the tester needs to check the requests sent by all clients, and make sure none of them matches the forwarded proxy request.

The more client connections the tester has created, the longer it takes the tester to check all connections before rejecting a buggy proxy.

These examples show that the time-consuming issue of some mutants are likely caused by the generators' heuristics. Cases like Mutant 5 can be optimized by state-based heuristics

```

Inductive node :=
  File      : content          → node
| Directory : list (name*node) → node.

Context read : path → node    → option content.
Context write: path → content → node → option node.
Context mkdir: path → node    → option node.
Context ls   : path → node    → list name.
Context rm   : path → node    → option node.

```

Figure 6.3: File system specification.

in Subsection 5.2.1; Proxy-related bugs can be more easily found by trace-based heuristics in Subsection 5.2.2; For Mutants 18–20, the requests should be sent at specific time periods so that the resulting trace is unacceptable per specification, which is discussed in Chapter 8.

6.2. Testing file synchronizer

To demonstrate the generality of my specification-based testing methodology, I applied it to file synchronizers. Subsection 6.2.1 introduces my specification of the file system and synchronization semantics. From these specifications, I derived a tester program for the Unison file synchronizer [30], with results shown in Subsection 6.2.2.

6.2.1. System Under Test

A file synchronizer manipulates the file system to reconcile updates in different replicas [6]. To check a synchronizer’s correctness, the tester needs to update replicas, launch the synchronization process, and observe the propagated updates.

My specification consists of two parts:

1. A file system model represented as a tree, where the leaves are files and the branches are directories. As shown in Figure 6.3, the file system model is a simplified abstraction from the POSIX file interface, ignoring metadata and file permissions. Specifying more aspects of the file system is discussed in Chapter 8.

Based on the file system model, I specified five basic file operations the tester may perform: (i) reading contents from a file, (ii) writing contents to a file, (iii) creating a

new directory, (iv) listing entries under a directory, and (v) removing a file or directory recursively.

Some file operations may fail *e.g.* when reading from a path that refers to a directory. These failures are represented as return value (`None: option _`) in the node functions.

2. A file synchronizer model that syncs updates between two replicas, implementing the reconciliation algorithm by Balasubramaniam and Pierce [6]:

Definition $\sigma := \text{node} * \text{node} * \text{node}.$

Context `reconcile`: $\sigma \rightarrow \sigma.$

Notice that the `reconcile` function manipulates three replicas. This is because the synchronizations might be partial: Upon write-write and write-delete conflicts, the synchronizer does not propagate the conflicting updates, and leaves the dirty files untouched in both replicas.

The third parameter of the `reconcile` function represents the subset of the two replicas that were synchronized: (`reconcile (a,b,g)`) syncs replicas `a` and `b` based on their previous consensus `g`. The consensus is initially empty, and updated when a change in one replica is propagated to another, or the two replicas have made identical changes.

Having specified the file system interface and the reconciliation semantics, I modelled the SUT as a deterministic QA server described in Definition 2.6. As shown in Figure 6.4, the request type `Q` can be file operations or the synchronization command; the response type `A` carries the return value of file system queries, or the transactions' exit code. For example, when synchronizing the two replicas, code 1 indicates partial synchronization with conflicts unresolved, and code 2 means the synchronizer has crashed due to uncaught exceptions or interruptions.

The QA model for the file system+synchronizer was dualized into a tester program that makes system calls to manipulate files, launch the synchronizer, and observe the updates. The system calls are made one at a time, and the file synchronizer is run as a foreground

```

Variant F :=          (* file operations *)
  Fls      (p: path)
| Fread   (p: path)
| Fwrite  (p: path) (c: content)
| Fmkdir  (p: path)
| Frm     (p: path).

Variant R := R1 | R2. (* replicas      *)

Variant Q :=          (* query type    *)
  QFile (r: R) (f: F)
| QSync.

Variant A :=          (* response type *)
  Als   (l: list name)
| Aread (c: content)
| Aexit (z: Z).        (* exit code    *)

```

Figure 6.4: Query and response types for testing file synchronizer.

process that blocks other interactions. Testing the synchronizer as a nonblocking background process is discussed in Chapter 8.

6.2.2. Experiment Results

The tester has revealed two features of the Unison file synchronizer, and I reported them to the developers. By analyzing the program’s behavior, the developers determined that Unison did not violate its specification, which allows a wider set of behavior than my specification defines. The revealed features are as follows:

Synchronizing read-only directories When the tester creates a directory in one replica with read and execution permission (mode 500) and calls the synchronization command, Unison crashed without creating the corresponding directory in the other replica.

The crashing behavior only occurs on macOS, and is caused by Unison’s mechanism for propagating changes: When copying directory `foo` from replica A to replica B, the synchronizer first creates a temporary directory “B/.unison.foo.xxxx.unison.tmp”, and then renames it to “B/foo”. The `rename` implementation in macOS requires write permission to proceed, so the change was not propagated.

This issue is not considered a violation in Unison or macOS, because: (1) Unison is allowed to halt without propagating an expected change, as long as its exit code has indicated an error, and no unexpected change was propagated. (2) The POSIX specification [21] says the `rename` function *may* require write access to the directory.

Despite being compliant to the specification, this feature in Unison is considered a defect, as it disables synchronization of read-only files and directories. A potential fix might be substituting `rename` with other system calls.

This defect was revealed by accident: My file system specification in Figure 6.3 does not mention file permissions, so I defaulted to mode 755. However, when implementing the test executor, I made a typo that wrote the permission as hexadecimal 0x755 while it should be octal 0o755. This caused the created directory to have mode bits 525, which triggered the aforementioned behavior.

This experiment shows that my current abstraction of the file system is worth expanding to include more information like file permissions, which might reveal other features of the file synchronizer.

The experiment also reveals a challenge in specifying programs, that the underlying operating system might also pose uncertainty to the program’s behavior. Such external non-determinism may be handled by parameterizing the program’s specification over the OS’s, in a similar way as composing the server model with the network model in Section 4.2.

Detecting write-delete conflict Suppose replicas A and B have a synchronized file `foo.txt`. If the tester deletes `A/foo.txt` and writes to `B/foo.txt`, then according to the specification, the synchronizer should detect this conflict and halt with exit code 1. During the tests, Unison did not notice that `B/foo.txt` was changed, and decided to propagate the deletion from replica A to replica B.

This behavior is caused by Unison’s “fastcheck” feature that improves the performance at the risk of ignoring conflicts. With fastcheck enabled by default on *nix systems, the syn-

chronizer detects file modifications by checking if their timestamps have changed. When the tester writes to the files with short intervals, the timestamps might remain unchanged, so the synchronizer treats the written file as unmodified. Such behavior can be avoided by updating the file contents after a longer interval.

Notice that although Unison decided to propagate the file deletion, it does check the file contents before deleting it. As a result, the Unison process crashed and complained that `B/foo.txt` was modified during synchronization.

If the fastcheck feature was disabled, then Unison can detect the conflict by comparing the file contents bit-by-bit. It'll propagate no updates and leave the two replicas as-is, which results the same as enabling fastcheck but at lower performance.

This experiment shows that the executor may affect the test result by interacting with the SUT slowly or quickly. It is worth expanding the test harness to allow different configurations for executing the tests.

So far I have demonstrated the effectiveness of my testing methodology, by deriving testers that can reveal violations and interesting caveats in real-world systems like web servers and file synchronizers. In the following chapters, I'll discuss related works in specifying and testing interactive systems, as well as potential improvements that can be made.

CHAPTER 7

RELATED WORK

The area of automated testing is well-studied, diverse, and difficult to navigate [1]. This chapter explores methodologies in specifying and testing interactive systems. I focus on techniques that have been practised in testing real-world programs. Section 7.1 compares different specification techniques and their derived validators. Section 7.2 then discusses practices in developing test harnesses.

7.1. From Specifications to Validators

Different testing scenarios exhibit various challenges that motivate the specification design. This section partitions the validation techniques by the languages used for developing the specifications.

7.1.1. State machine specification: Quviq QuickCheck

Property-based testing with QuickCheck has been well adopted in industrial contexts [19]. The specification is a boolean function over traces *i.e.* the validator. My solutions to addressing internal and external nondeterminism are inspired by practices in QuickCheck.

Internal nondeterminism My HTTP/1.1 specification was initially written as a QuickCheck property. Before handling preconditions like `If-Match` and `If-None-Match`, the validator implements a deterministic server model and compares its behavior with the observations, as shown in the `validate` function in Section 1.1. When expanding the specification to cover conditional requests, I implemented the ad hoc validator by manually translating the trace into tester-side knowledge, as shown in Figure 1.1.

The complexity in describing “what behavior is valid” motivates me to rephrase the specification. I applied the idea of model-based testing [9], and specified the protocol in terms of “how to produce valid behavior”. My specification represented internal nondeterminism with symbolic variables. The validator then checks whether the trace is producible by the

symbolic specification, by reducing the producibility problem to constraint solving.

External nondeterminism Hughes et al. [20] have used QuickCheck to test Dropbox. The specification does not involve internal nondeterminism, but does handle external nondeterminism that local nodes may communicate with the server at any time. This is done by introducing “conjectured events” to represent the possible communications. The validator checks if the conjectured events can be inserted to somewhere in the trace to make it producible by the model.

To specify servers’ allowed observations delayed by the network, Koh et al. [22] introduced the concept of “network refinement”. The network may scramble the traces by delaying some messages after others, with one exception: If the client has received response **A** before it sends request **Q**, then by causality, the server-side trace must have sent response **A** before receiving request **Q**. Upon observing a client-side trace **T**, our QuickCheck validator searches for a server-side trace that (i) can be reasonably scrambled into trace **T**, and (ii) complies with the protocol specification.

My network model design in Section 4.2 was inspired by these idea of conjecturing the environment’s behavior. Instead of inserting conjectured communication events or reordering the trace, I specified the network as an independent module and composed it with the server specification. This provides more flexibility in specifying the environment *e.g.* limiting the buffer size or the number of concurrent connections, and allows reusing the network models for specifying other protocols on top of it.

7.1.2. Process algebra: LOTOS and TorXakis

Language of Temporal Ordering Specification (LOTOS) [8] is the ISO standard for specifying OSI protocols. It defines distributed concurrent systems as *processes* that interact via *channels*. Using a formal language strongly inspired by LOTOS, Tretmans and van de Laar [35] implemented a test generation tool called TorXakis, and used it for testing Dropbox.

TorXakis supports internal nondeterminism by defining a process for each possible value.

This requires the space of invisible values to be finite. In comparison, I represented invisible values as symbolic variables, and employed constraint solving that can handle infinite space of data like strings and integers.

As for external nondeterminism, TorXakis hardcodes all channels between each pair of processes, assuming no new process joins the system. Whereas in my network model, “channels” are the “source” and “destination” fields of network packets, which allows specifying a server that exposes its port to infinitely many clients.

7.1.3. Transition systems: NetSem and Modbat

Using labelled transition systems (LTS), Bishop et al. [7] have developed rigorous specification for TCP, UDP, and the Sockets API. To handle internal nondeterminism in real-world implementations, they symbolized the model states, which is then evaluated with a special-purpose model symbolic model checker. They focused on developing a post-hoc specification that matches mainstream systems like FreeBSD, Linux, and WinXP, while I aim at finding bugs in rapidly evolving implementations, and borrowed the idea of symbolic evaluation in validating observations.

Artho and Rousset [4] have generated test cases for Java network API, which involves blocking and non-blocking communications. Their abstract model was based on extended finite state machines (EFSM), and could capture bugs in the network library `java.nio`. Their validator rejects the SUT upon unexpected exceptions or observations that fail its *encoded* assertions. In comparison, assertions in my validator are *derived* from the abstract model, which covers full functional correctness of the SUT.

7.2. Test Harnesses

This section explores techniques of generating and shrinking test inputs. Subsection 7.2.1 compares different heuristics used by test generators; Subsection 7.2.2 explains existing shrinking techniques for interactive testing scenarios. The executor of test harnesses is beyond the scope of this thesis, and will be discussed in Chapter 8.

7.2.1. Generator Heuristics

In addition to state-based and trace-based heuristics discussed in Section 5.2, other kinds of heuristics can be applied to generating inputs for various testing scenarios.

Constraint-based heuristics The reason for introducing heuristics is to increase the chance of triggering invalid behavior. I specified the heuristics as “how to produce interesting input”, while in some cases, it’s more convenient to specify “what inputs are interesting”. For example, well-typed lambda expressions can be easily defined in terms of typing rules, but are less intuitive to enumerate by a generator.

Narrowing [2] allows generating data that satisfy certain constraints. Lampropoulos et al. [24] have applied the idea of narrowing to the QuickChick testing framework in Coq [23], representing constraints as inductive relations. The relations are compiled into efficient generators that produce satisfying data.

The narrowing generator in QuickChick has been used during my preliminary experiment with HTTP/1.1, where I defined a well-formed relation for HTTP requests to guide the generator. This constraint-based heuristics can be integrated to my testing framework by interpreting QuickChick’s `Gen` (generator) monad into the `IO` monad used by the test harness.

Coverage-based heuristics Another strategy to increase the chance of revealing invalid behavior is to cover more execution paths of the SUT. This idea is applied to fuzz testing [27], with popular implementations AFL [40] and Honggfuzz [32], and combined with property-based testing by Lampropoulos et al. [25].

Coverage-based testers mutate the test inputs and observe the programs’ execution paths. An input is considered interesting if it causes the program to traverse a previously unvisited path. The interesting inputs will be mutated and rerun to cover potentially more paths.

To track the program’s execution paths, coverage-based heuristics need to instrument the SUT during compilation, making it inapplicable for black-box testing. When fuzzing interactive systems like web servers, the SUT is run in a simulator where the requests are provided

as files instead of network packets. The responses are ignored in fuzz testing, which does not check functional correctness. To produce a trace for validation, the SUT needs to be carefully modified to record its send and receive events. Addressing these challenges would enable coverage-based heuristics to be integrated in interactive testing of systems' functional correctness.

7.2.2. Shrinking Interactive Tests

To address inter-execution nondeterminism in Quviq QuickCheck, Hughes [19] introduced the idea of shrinking abstract representations of test input. The tester first generates a *script* using state-based heuristics, and instantiates the script with the tester's runtime state. The scripts can be shrunk and adapted to new runtimes when rerunning the test.

The language design of my test harness is inspired by the QuickCheck approach, and extends it in two dimensions:

1. The QuickCheck framework assumes synchronous interactions, where the requests are function calls that immediately return. When testing asynchronous systems *e.g.* networked servers, the responses might be indefinitely delayed by the environment, which would block the QuickCheck tester's state transition.

To generate test inputs asynchronously, I implemented a non-blocking algorithm for instantiating scripts into requests. When a dependant observation is absent due to delays or loss by the environment, the test harness tries to instantiate the request using other observations, instead of waiting for the observation from the environment.

2. QuickCheck requires the test developer to specify a runtime state to guide the generator, and define the state transition rules for each interaction.

In comparison, I proposed trace-based and state-based heuristics as separate concepts. Generators in my testing framework are based on (i) traces recorded automatically during runtime, and (ii) model states exposed by the underlying specifications. The generic intermediate representation of traces lifts the requirement of specifying tran-

sitions for test generation purposes.

CHAPTER 8

CONCLUSION AND FUTURE WORK

This thesis presents a systematic technique for testing interactive systems with uncertain behavior. I propose a theory of dualizing protocol specification into validators, with formal guarantee of soundness and completeness (Chapter 2). To test systems in real-world practices, I applied the dualization theory to the interaction tree specification language, and derived specifications into interactive testing programs (Chapter 4). I then presented a test harness design to generate and shrink interactive test inputs effectively (Chapter 5). The entire methodology is evaluated by revealing programs' in compliant behavior with derived testers (Chapter 6).

To address challenges posed by internal, external, and inter-execution nondeterminisms, I introduced various flavors of symbolic abstract interpretation: Systems' internal choices are represented as symbolic variables and unified against the tester's observations (??, Section 4.3); Possible impacts made by the environment are represented as nondeterministic branches (Section 4.2), and determined by backtrack searching (Section 4.4); The test inputs are generated as symbolic intermediate representations that can adapt to different traces during runtime (Section 5.3).

The technique in this thesis can be expanded in scenario and by combination:

1. Specifying and testing other kinds of SUTs in various setups *e.g.* background monitoring processes, distributed locks, cyber-physical systems *etc.* These attempts might expose the limitations of my methodology or introduce new challenges in testing.
2. Integrating other testing techniques to the framework *e.g.* tuning the executor to capture time-related violations [31], find bugs with fewer tests by enumerating within a small input space [17], *etc.* This may enhance my testers' effectiveness or reveal potential improvements that existing techniques can make.

BIBLIOGRAPHY

- [1] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978 – 2001, 2013. ISSN 0164-1212. doi: <https://doi.org/10.1016/j.jss.2013.02.061>. URL <http://www.sciencedirect.com/science/article/pii/S0164121213000563>.
- [2] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. *J. ACM*, 47(4):776–822, jul 2000. ISSN 0004-5411. doi: 10.1145/347476.347484. URL <https://doi.org/10.1145/347476.347484>.
- [3] Andrew W. Appel, Lennart Beringer, Adam Chlipala, Benjamin C. Pierce, Zhong Shao, Stephanie Weirich, and Steve Zdancewic. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 375(2104):20160331, oct 2017. ISSN 1364-503X. doi: 10.1098/rsta.2016.0331. URL <https://royalsocietypublishing.org/doi/10.1098/rsta.2016.0331>.
- [4] Cyrille Artho and Guillaume Rousset. Model-based testing of the java network api. *Electronic Proceedings in Theoretical Computer Science*, 245:46–51, Mar 2017. ISSN 2075-2180. doi: 10.4204/eptcs.245.4. URL <http://dx.doi.org/10.4204/EPTCS.245.4>.
- [5] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG ’06, page 2–10, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934901. doi: 10.1145/1159789.1159792. URL <https://doi.org/10.1145/1159789.1159792>.
- [6] S. Balasubramaniam and Benjamin C. Pierce. What is a file synchronizer? In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, MobiCom ’98, page 98–108, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 158113035X. doi: 10.1145/288235.288261. URL <https://doi.org/10.1145/288235.288261>.
- [7] Steve Bishop, Matthew Fairbairn, Hannes Mehnert, Michael Norrish, Tom Ridge, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: Rigorous test-oracle specification and validation for tcp/ip and the sockets api. *J. ACM*, 66(1), December 2018. ISSN 0004-5411. doi: 10.1145/3243650. URL <https://doi.org/10.1145/3243650>.
- [8] Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Computer Networks and ISDN Systems*, 14(1):25 – 59, 1987. ISSN 0169-7552. doi: [https://doi.org/10.1016/0169-7552\(87\)90085-7](https://doi.org/10.1016/0169-7552(87)90085-7). URL <http://www.sciencedirect.com/science/article/pii/0169755287900857>.

- [9] Manfred Broy, Bengt Jonsson, J-P Katoen, Martin Leucker, and Alexander Pretschner. Model-based testing of reactive systems. In *Volume 3472 of Springer LNCS*. Springer, 2005.
- [10] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132026. doi: 10.1145/351240.351266. URL <https://doi.org/10.1145/351240.351266>.
- [11] Lisa M. Dusseault. HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV). RFC 4918, June 2007. URL <https://www.rfc-editor.org/info/rfc4918>.
- [12] Roy T. Fielding and Gail Kaiser. The apache http server project. *IEEE Internet Computing*, 1(4):88–90, July 1997. ISSN 1941-0131. doi: 10.1109/4236.612229.
- [13] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, June 2014. URL <https://www.rfc-editor.org/info/rfc7231>.
- [14] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests. RFC 7232, June 2014. URL <https://www.rfc-editor.org/info/rfc7232>.
- [15] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999. URL <https://www.rfc-editor.org/info/rfc2616>.
- [16] George Fink and Matt Bishop. Property-based testing: A new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80, July 1997. ISSN 0163-5948. doi: 10.1145/263244.263267. URL <https://doi.org/10.1145/263244.263267>.
- [17] Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. Do judge a test by its cover. In Nobuko Yoshida, editor, *Programming Languages and Systems*, pages 264–291, Cham, 2021. Springer International Publishing. ISBN 978-3-030-72019-3.
- [18] Marijn Haverbeke. Dav module does not respect if-unmodified-since, Nov 2012. URL <https://trac.nginx.org/nginx/ticket/242>.
- [19] John Hughes. *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane*, pages 169–186. Springer International Publishing, Cham, 2016. ISBN 978-3-319-30936-1. doi: 10.1007/978-3-319-30936-1_9. URL https://doi.org/10.1007/978-3-319-30936-1_9.
- [20] John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. Mysteries of drop-box: 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*, pages 135–145, 2016. doi: 10.1109/ICST.2016.37. URL <https://doi.org/10.1109/ICST.2016.37>.
- [21] IEEE. IEEE Std 1003.1-2017: POSIX.1a. Web document, 2017. URL <https://pubs.opengroup.org/onlinepubs/9699919799/>. POSIX.1-2017 is a revision of IEEE Std 1003.1-2008, and replaces previous versions of 2008, 2013, and 2016. Also issued as The Open Group Technical Standard Base Specifications, Issue 7.
 - [22] Nicolas Koh, Yao Li, Yishuai Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, William Mansky, Benjamin C. Pierce, and Steve Zdancewic. 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*, pages 234–248, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6222-1. doi: 10.1145/3293880.3294106. URL <http://doi.acm.org/10.1145/3293880.3294106>.
 - [23] 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>.
 - [24] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017. doi: 10.1145/3158133. URL <https://doi.org/10.1145/3158133>.
 - [25] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. Coverage guided, property based testing. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. doi: 10.1145/3360607. URL <https://doi.org/10.1145/3360607>.
 - [26] 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.
 - [27] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, dec 1990. ISSN 0001-0782. doi: 10.1145/96267.96279. URL <https://doi.org/10.1145/96267.96279>.
 - [28] Netcraft. Web server survey, Mar 2022. URL <https://news.netcraft.com/archives/category/web-server-survey/>.
 - [29] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
 - [30] Benjamin C. Pierce and Jérôme Vouillon. Unison: A file synchronizer and its specification. In Naoki Kobayashi and Benjamin C. Pierce, editors, *Theoretical Aspects of Computer Software*, pages 560–560, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-45500-4.

- [31] 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, page 79–89, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350761. doi: 10.1145/3092703.3092706. URL <https://doi.org/10.1145/3092703.3092706>.
- [32] Robert Swiecki. Honggfuzz. URL <https://github.com/google/honggfuzz>.
- [33] Igor Sysoev. Nginx. URL <http://nginx.org/>.
- [34] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49 – 79, 1996. ISSN 0169-7552. doi: [https://doi.org/10.1016/S0169-7552\(96\)00017-7](https://doi.org/10.1016/S0169-7552(96)00017-7). URL <http://www.sciencedirect.com/science/article/pii/S0169755296000177>. Protocol Testing.
- [35] Jan Tretmans and Pi  re van de Laar. Model-based testing with torxakis: The mysteries of dropbox revisited. In *Strahonja, V.(ed.), CECIS: 30th Central European Conference on Information and Intelligent Systems, October 2-4, 2019, Varazdin, Croatia. Proceedings*, pages 247–258. Zagreb: Faculty of Organization and Informatics, University of Zagreb, 2019.
- [36] Jan Tretmans and Pi  re van de Laar. TorXakis. URL <https://github.com/TorXakis/Torxakis>.
- [37] Lionel Sujay Vailshery. Quality assurance and testing budget allocation 2012–2019, Feb 2022. URL <https://www.statista.com/statistics/500641/worldwide-qa-budget-allocation-as-percent-it-spend/>.
- [38] W3Techs. Historical trends in the usage statistics of web servers. URL https://w3techs.com/technologies/history_overview/web_server.
- [39] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: Representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, December 2019. ISSN 2475-1421. doi: 10.1145/3371119. URL <http://doi.acm.org/10.1145/3371119>.
- [40] Michal Zalewski. American fuzzy lop, 2017.
- [41] Hengchu Zhang, Wolf Honor  , Nicolas Koh, Yao Li, Yishuai Li, Li-Yao Xia, Lennart Beringer, William Mansky, Benjamin Pierce, and Steve Zdancewic. Verifying an HTTP Key-Value Server with Interaction Trees and VST. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving (ITP 2021)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages

32:1–32:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-188-7. doi: 10.4230/LIPIcs.ITP.2021.32. URL <https://drops.dagstuhl.de/opus/volltexte/2021/13927>.