# 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

# 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 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 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 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

# CHAPTER 1

## Introduction

Software engineering requires rigorous testing of rapidly evolving programs, which costs manpower comparable to developing the product itself [36]. To guarantee programs' compliance with their specifications, we need testers that can tell compliant implementations from violating ones.

This thesis studies how to test the semantics of interactive systems: 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 kind of testing is applicable in many scenarios, including web servers, distributed file systems, *etc.*

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).

**Convention** In this thesis, I use standard terminologies and conventions from functional programming, such as monads and coinduction. The meta language for data structures and algorithms is Coq, with syntax simplified in places for readability.

### 1.1. Interactive Testing

Suppose we want to test a web server that supports GET and PUT methods. We can represent the server as a stateful program.

```
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 syntax "`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. Syntax "`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')`.

The `server` function iterates with 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.

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 restructured into 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) [13], 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:

```
/* Client: */
PUT /target HTTP/1.1
If-Match: "tag-foo"
... content (A) ...
```

```
/* Server: */
HTTP/1.1 204 No Content
```

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

```
/* Server: */
HTTP/1.1 200 OK
ETag: "tag-bar"
... content (A) ...
```

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

```
/* Client: */
PUT /target HTTP/1.1
If-Match: "tag-baz"
... content (B) ...
```

```
/* Server: */
HTTP/1.1 412 Precondition Failed
```

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

```
/* Server: */
HTTP/1.1 200 ok
ETag: "tag-bar"
... content (A) ...
```

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. 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 such validators in a scalable way for more complex protocols. This is one challenge posed by internal nondeterminism.

### 1.2.2. External nondeterminism

To discuss the nondeterminism caused by the environment, we need to more precisely define the environment concept as it pertains to this testing scenario.

**Definition 1.1** (Environment, input, output, and observations)**.** The *environment* is the substance that the tester and the SUT interact through. The *input* is the subset of the environment that the tester can manipulate. The *output* is the subset of the environment that the SUT can alter. The *observation* is the tester's view of the inputs and the outputs.

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

```
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`.

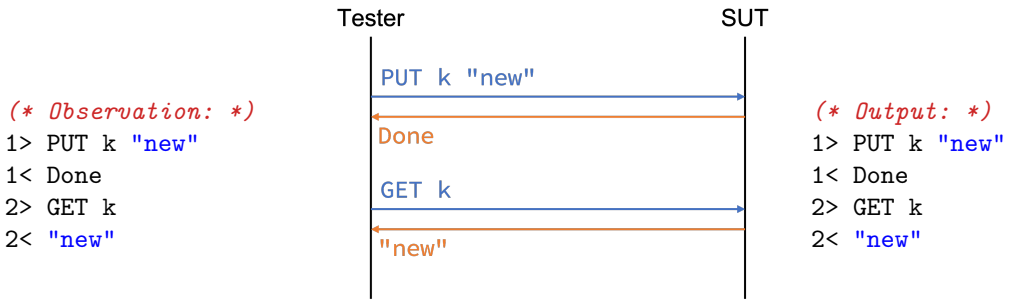



Figure 1.2: With no concurrency, the observation is identical to the output.

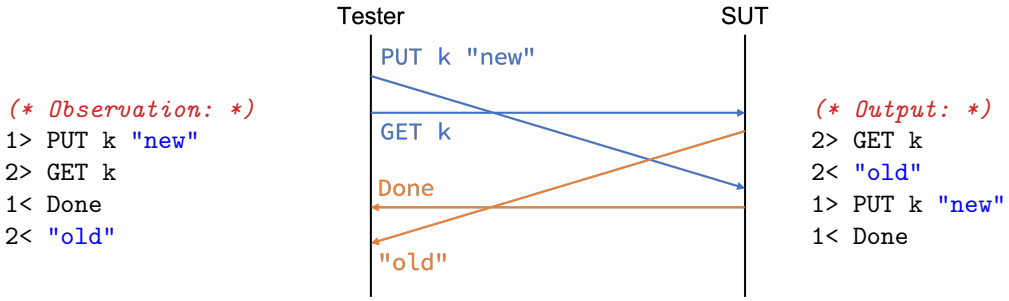



Figure 1.3: Acceptable: The observation can be explained by a valid output reordered by the network environment.

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 observe the server's behavior upon concurrent requests, the tester needs to simulate multiple clients, sending new requests before receiving previous responses. The network delay might cause the server to receive requests in a different order from that on the tester side. Conversely, responses sent by the server might be reordered before arriving at the tester, as shown in Figure 1.3. Such an observation can be explained by various outputs on the SUT side. The validator needs to consider all possible outputs that can explain such an observation, and see if any one of them complies with the specification. If no valid output

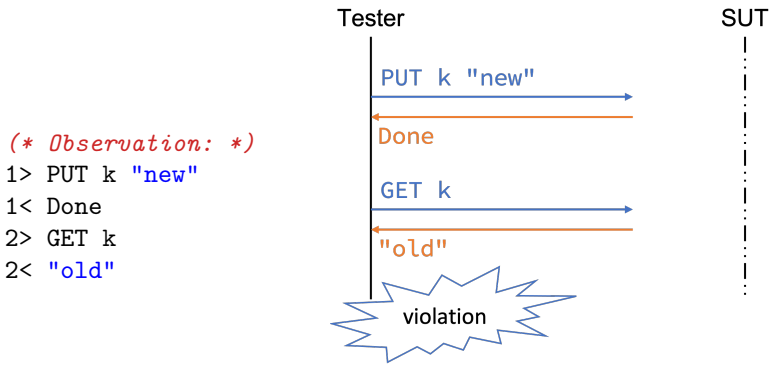



Figure 1.4: Unacceptable: The tester received the `Done` response before sending the `GET` request, thus the SUT must have processed the `PUT` request before the `GET` request. Therefore, the `"old"` response is invalid.

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 a 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 inputs [18]. 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 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.

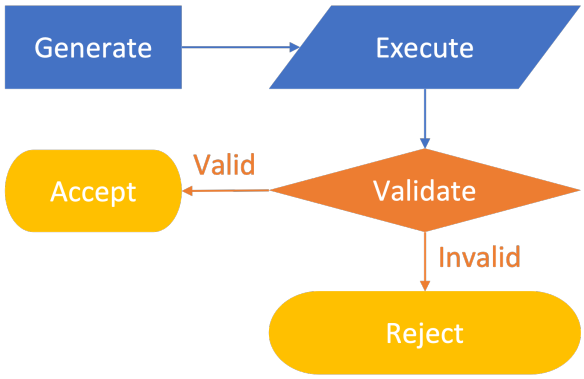



Figure 1.5: Simple tester architecture without shrinking.

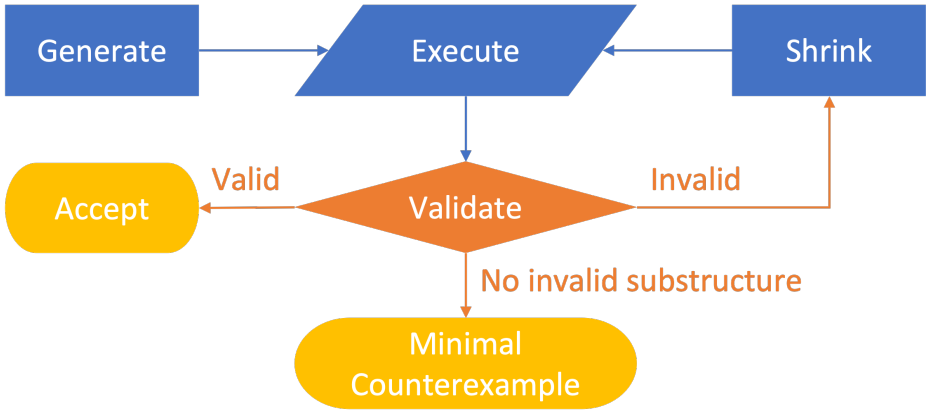



Figure 1.6: Tester architecture with shrinking mechanism.

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 a subset is usually sparse and cannot be enumerated within a reasonable budget, e.g., in Subsection 1.2.1, it would be infeasible to enumerate request ETags to 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: Tue, 7 Jun 2022 07:20:29 GMT
                           /* Server: */
                           HTTP/1.1 200 OK
                           Last-Modified: Wed, 8 Jun 2022 07:20:29 GMT
                           ... content of target ...
/* Client: */
GET /index.html HTTP/1.1
If-Modified-Since: Wed, 8 Jun 2022 07:20:29 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 problem is that the timestamps on the server might be different from the previous execution, so an interesting timestamp in a previous run might become meaningless in this run.

The fact 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 sketch the state of the art in the relevant parts of the field of testing, and describe its limitations in addressing the challenges posed by nondeterminism.

### 1.4.1. Property-based testing: QuickCheck

Property-based testing [15] 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 [9] and its variant QuickChick for Coq [22]. These tools can generate random inputs that satisfy logical conditions [23] and integrate with fuzz testing [24] and combinatorial testing [16], and they have tested real-world systems like telecoms software [4] and Dropbox [19].

### 1.4.2. Model-based testing: TorXakis

Instead of specifying predicates over systems' behavior, model-based testing [8] 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 [35], whose modelling language is inspired by Language of Temporal Ordering Specification (LOTOS) [7], the ISO standard for specifying distributed systems. The tool can compile process algebra specifications into tester programs, and can be used for testing dropbox [34].

### 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 [18] 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 presents "testing by dualization" (TBD), a technique that addresses the challenges in asynchronous testing caused by various forms of nondeterminism. I introduce symbolic languages for specifying the protocol and representing test input, and I 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 implementa-

tion, 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** Testing by dualization can address challenges in testing interactive systems with uncertain behavior. Specifying protocols with a 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 results 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* [21], 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* [25], with Benjamin C. Pierce and Steve Zdancewic, 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 a computation 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 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 theoretical view of synchronous testing that involves internal nondeterminsm. 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 the soundness and completeness of validators with respect to the specification.

### 2.1. Concepts

Testers are programs that determine whether implementations are compliant or not, based on observations. This section defines 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 and 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 "$\mathsf{valid}_s\ t$".

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

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

**Definition 2.3** (Tester components and correctness)**.** A tester consists of (i) a *validator* that accepts or rejects traces (written as "$\mathsf{accept}_v\ t$" and "$\neg\mathsf{accept}_v\ t$"), and (ii) a *test harness*

that triggers different traces with various inputs.

A tester is *rejection-sound* if it rejects only non-compliant implementations; it is *rejection-complete* if it can reject all non-compliant implementations, provided sufficient time of execution. A tester is *correct* if is rejection-sound and -complete.[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 "$v$ $\mathsf{sound}_s^{\mathsf{Rej}}$") if it only rejects traces that are invalid per $s$:

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

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

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

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. Languages in the QAC family can specify protocols of various message types, server states, and internal choices, by instantiating the specification language with different type arguments.

[1] The semantics of "soundness" and "completeness" vary among contexts. This thesis inherits terminologies from existing literature [33], 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, and $A$ be the result type. Then type $(S \to A \times S)$ 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 $(K \to V)$, then we can define `get` and `put` computations as follows:

$$\texttt{get} : K \to ((K \to V) \to V \times (K \to V)) \tag{1}$$

$$\texttt{get}(k)(f) \triangleq (f(k), f) \tag{2}$$

$$\texttt{put} : K \times V \to ((K \to V) \to () \times (K \to V)) \tag{3}$$

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

These function definitions should be read as:

1. The `get` function takes a key as argument, and constructs a state monad with state type $(K \to V)$ and result type $V$.

2. Given argument $k$ of type $K$, $\texttt{get}(k)$ 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 $(K \to V)$ and result type "()" (unit type, which corresponds to `void` return type in C/Java functions).

4. Given argument $(k, v)$ of type $(K \times V)$, $\texttt{put}(k, v)$ takes a mapping $f$ as pre state and substitues 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:

$$\mathsf{DeterministicServer} \triangleq \exists S, (Q \to S \to A \times S) \times S$$

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 [28], 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:

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

This expression is pronounced as: The server state is of type $\sigma$. Its loop body is function $\mathsf{sstep}$ (which has type $Q \to \sigma \to A \times \sigma$) and its initial state is $\mathit{state}_0$ (which has type $\sigma$).

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:

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

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 nondeterminstic 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 $C$:

$$\mathsf{Server} \triangleq \exists S, (Q \times C \to S \to A \times S) \times S$$

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:

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

$$, 0)$$

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 "$\mathsf{valid}_s\ t$" 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 $\mathsf{stepServer}$ function:

$$s \xrightarrow[c]{(q,a)} s' \quad \triangleq \quad \mathsf{stepServer}(q,c)(s) = (a, s')$$

$$\mathsf{stepServer} : Q \times C \to \mathsf{Server} \to A \times \mathsf{Server}$$

$$\mathsf{stepServer}(q,c)(s) \triangleq$$

$$\quad \mathsf{let}\ (\mathsf{pack}\ S = \sigma\ \mathsf{with}\ (\mathsf{sstep}, \mathit{state})) = s\ \mathsf{in}$$

$$\quad \mathsf{let}\ (a, \mathit{state}') = \mathsf{sstep}(q,c)(\mathit{state})\ \mathsf{in}$$

$$\quad (a, \mathsf{pack}\ S = \sigma\ \mathsf{with}\ (\mathsf{sstep}, \mathit{state}'))$$

The $\mathsf{stepServer}$ function takes a query and a choice and computes a state monad with state type $\mathsf{Server}$ and result type $A$, by pattern matching on argument $(s : \mathsf{Server})$. Let $\sigma$ be the server state type of $s$, $\mathsf{sstep}$ the loop body, and $(\mathit{state} : \sigma)$ the current state of $s$. Then $\mathsf{stepServer}(q,c)(s)$ computes the result $(a : A)$ and the post state $(\mathit{state}' : \sigma)$ using the $\mathsf{sstep}$ function, and substitutes the server's pre-step $\mathit{state}$ with the post-step $\mathit{state}'$.

**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:

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

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:

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

### 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 $V$":

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

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

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

$$0)$$

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 acception notation "$\mathsf{accept}_v\ t$" 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:

$$v \xrightarrow{(q,a)} v' \quad \triangleq \quad \mathsf{stepValidator}(q,a)(v) = \mathsf{Some}\ v'$$

$$\mathsf{stepValidator} : Q \times A \to \mathsf{Validator} \to \mathsf{option\ Validator}$$

$$\mathsf{stepValidator}(q,a)(\mathsf{pack}\ V = \beta\ \mathsf{with}\ (\mathsf{vstep}, state))$$

$$\triangleq \begin{cases} \mathsf{Some}\ (\mathsf{pack}\ V = \beta\ \mathsf{with}\ (\mathsf{vstep}, state')) & \mathsf{vstep}(q,a,state) = \mathsf{Some}\ state' \\ \mathsf{None} & \mathsf{vstep}(q,a,state) = \mathsf{None} \end{cases}$$

The stepValidator function takes a query and a response, and computes the validator transition by pattern matching on argument $(v : \mathsf{Validator})$. Let $\beta$ be the validator state type of $v$, vstep be the loop body and $(state : \beta)$ the current state of $v$. Then $\mathsf{stepValidator}(q,a)(v)$ 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)**.** Validator $v$ accepts trace $t$ if and only if it *cosumes* the trace and steps to some validator $v'$, written as "$v \xrightarrow{t} v'$ ":

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

Here the consumability relation is defined as follows:

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:

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

## 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 validator consumes all traces that are producible by the protocol specification:

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

2. A rejection-complete validator only consumes traces that are producible by the protocol specification:

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

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 cor-

respondence between each server and validator step. This is done by introducing some loop invariant between the server and validator states and showing that it is preserved by the server's and the validator's loop body.

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

### 2.3.1. Proving rejection soundness

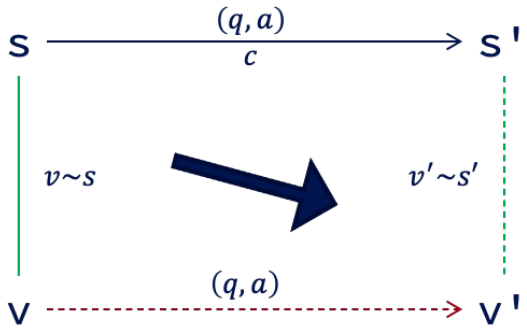
To prove that any trace producible by the server is consumable by the validator, I perform forward induction on the server's execution path and show that every step has a corresponding validator step:

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

$$v_0 \sim s_0 \tag{RejSound-Init}$$

- 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 $s'$:

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




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 $\sigma, \beta$ and step functions $\mathsf{sstep}, \mathsf{vstep}$ 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 $v'$:

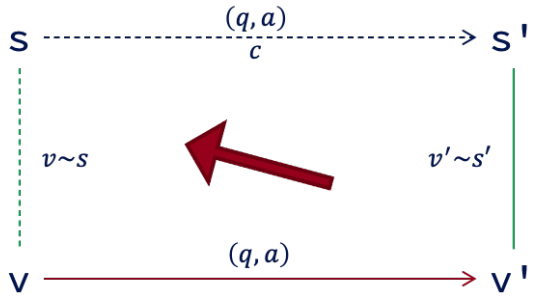
$$\forall (q : Q)(a : A)(v, v' : \beta),\ v \xrightarrow{(q,a)} v' \qquad \text{(RejComplete-End)}$$
$$\implies \exists s' : \sigma, v' \sim s'$$

  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 $v$:

$$\forall (q : Q)(a : A)(v, v' : \beta)(s' : \sigma), \qquad \text{(RejComplete-Step)}$$
$$\mathsf{vstep}(q, a, v) = \mathsf{Some}\ v' \wedge v' \sim s'$$
$$\implies \exists (s : \sigma)(c : C), \mathsf{sstep}(q, c, s) = (a, s') \wedge v \sim s$$




This allows us to inducively 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 $s_0$:

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

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 we 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 their specifications. The key idea is to encode the specification with a programming language, and *dualize* the specification program to derive a 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. Specifications written in Prog are dualized into validators in Section 3.2, with correctness proven in Coq 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 exposes a state monad interface for server models, which is a function that cannot be destructed within the meta language to perform program analysis. This section introduces a programming language for encoding specifications whose structures can be analyzed.

For readability, I demonstrate the dualization technique on a subset of QAC server models called integer machine models, 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:

$$s_0 : \mathbb{N} \to \mathbb{Z}$$

$$s_0 = (\_ \mapsto 0)$$

$$\text{i.e., } \forall (k : \mathbb{N}), s_0!k = 0$$

Here syntax "$s!k$" is pronounced "data stored in address $!k$ of memory $s$". I use "$!k$" to indicate that the natural number $k$ is being thought of as an address.

The server's queries, responses, 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 $f : (\mathbb{N} \to \mathbb{Z}) \to (\mathbb{N} \to \mathbb{Z})$ that manipulates the memory, and then sends back the value stored in address !1 as the response:

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

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

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

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

### 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:

$$
\begin{array}{lcll}
\mathsf{Prog} & \triangleq & \mathsf{return} & \text{end computation} \\
& | & !dst \coloneqq \mathsf{SExp}; \mathsf{Prog} & \text{write to address } dst \in \mathbb{N} \\
& | & \mathsf{if\ SExp} \leq \mathsf{SExp\ then\ Prog\ else\ Prog} & \text{conditional branch} \\
\mathsf{SExp} & \triangleq & \mathbb{Z} & \text{constant integer} \\
& | & !src & \text{read from address } src \in \mathbb{N} \\
& | & \mathsf{SExp}\ op\ \mathsf{SExp} & op \in \{+, -, \times, \div\}
\end{array}
$$

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

$$
\begin{array}{ll}
\mathsf{if} & !1 \leq 0 \\
\mathsf{then} & !1 \leftarrow (0 - !1);\ \mathsf{return} \\
\mathsf{else} & \mathsf{return}
\end{array}
$$

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

$$
\mathsf{eval}\ :\ \mathsf{Prog} \to (\mathbb{N} \to \mathbb{Z}) \to (\mathbb{N} \to \mathbb{Z})
$$

$$
\mathsf{eval}(p)(s) \triangleq \begin{cases}
s & p\ \mathsf{is\ return} \\
\mathsf{eval}(p')(s[dst \mapsto e^s]) & p\ \mathsf{is}\ !dst \coloneqq e; p' \\
\mathsf{eval}(\mathsf{if}\ {e_1}^s \leq {e_2}^s\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2)(s) & p\ \mathsf{is\ if}\ e_1 \leq e_2\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2
\end{cases}
$$

$$
e^s \triangleq \begin{cases}
z & e\ \mathsf{is}\ z : \mathbb{Z} \\
s!src & e\ \mathsf{is}\ !src \\
{e_1}^s\ op\ {e_2}^s & e\ \mathsf{is}\ e_1\ op\ e_2
\end{cases}
$$

Here "$e^s$" is pronounced "evaluating server expression ($e : \mathsf{SExp}$) on memory ($s : \mathbb{N} \to \mathbb{Z}$)". It substitutes all occurences 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 $s$:

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

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

$$\mathsf{serverOf} : \mathsf{Prog} \to \mathsf{Server}$$

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

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

$$\mathsf{if}\ !1 \leq !2\ \mathsf{then}\ !1 := 0; \mathsf{return} \quad (1)$$

$$\mathsf{else}\ !1 := 1; !2 := !0; \mathsf{return} \quad (2)$$

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:

$$\mathsf{validatorOf} : \mathsf{Prog} \to \mathsf{Validator}$$

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

$$\forall(p : \mathsf{Prog})(t : \mathsf{list}\ (Q \times A),$$
$$(\mathsf{validatorOf}(p) \xrightarrow{t}) \iff (\mathsf{serverOf}(p) \xrightarrow{t})$$

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 observations.

However, executing a nondeterministic specification does not produce a specific expectation of response, but a nondeterministic response that depends on the internal choice. Therefore, upon observing a response $A$, the validator should determine whether there is 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 require the responses observed from the implementation to be explainable by the specification.

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 variable for each address of the server model. It also remembers all the constraints added as observations are made during testing. This information is called a "validation state":

$$(\mathbb{N} \to \mathsf{var}) \times \mathsf{set\ constraint}$$

Here the constraints are relations between validator expressions (VExps) that may depend

on symbolic variables:

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

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 ($\mathbb{N} \to \mathsf{var}$), 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:

$$\mathsf{set}\ ((\mathbb{N} \to \mathsf{var}) \times \mathsf{set}\ \mathsf{constraint})$$

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 analyzing the computations of 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.

For example, if the server performs !2 ←!0, then the validator should step from $(vs, cs)$ to $(vs[2 \mapsto x], cs \cup \{\#x = \#(vs!0)\})$, where $\#x$ is a fresh variable. The new validation state says the value stored in address !2 is represented by variable $\#x$. It also constraints that $\#x$ should be equal to $\#(vs!0)$, the variable that represents address !0 in the pre-validation state $vs$.

2. When the server makes a nondeterministic branch $\mathsf{if}\ e_1 \leq e_2\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2$, the validator considers 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. If the set of validation states becomes empty, it indicates that the observation cannot be explained by any execution path of the specification, so the validator should reject the trace.

This mechanism is formalized in Figure 3.1. Here the notation "$v_0 \leftarrow v; \mathsf{vstep}'_p(q, a)(v_0)$" is a monadic bind for sets: Let $\mathsf{vstep}'_p$ map each element $v_0$ in $v$ to a set of validation states

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

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

$(\mathsf{vstep}'_p(q,a)(v_0) : \mathsf{set}\ ((\mathbb{N} \to \mathsf{var}) \times \mathsf{set\ constraint}))$, 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 ($e$ : 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:

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

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 ($\mathsf{var} \to \mathbb{Z}$) that satisfies all the constraints:

$$\forall cs, \mathsf{solvable}\ cs \iff \exists (asgn : \mathsf{var} \to \mathbb{Z}), asgn\ \mathsf{satisfies}\ cs$$

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

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

Here "$e^{asgn}$" is pronounced "evaluating validator expression ($e$ : VExp) with assignment ($asgn : \mathsf{var} \to \mathbb{Z}$)". It substitutes all occurences of "$\#x$" with their assigned value ($asgn!x$).

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

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

For example, to construct a validator for the CMP-RST protocol in Subsection 2.2.1, we first specify it in Prog as:

$$\begin{aligned} & \mathsf{if}\ !1 \leq !2 \\ & \mathsf{then}\ !1 \leftarrow 0; \mathsf{return} \\ & \mathsf{else}\ !1 \leftarrow 1; !2 \leftarrow !0; \mathsf{return} \end{aligned}$$

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.

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

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.

## 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:

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

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. The entire proof is formalized in the Coq proof assistant.

### 3.3.1. Designing the loop invariant

Let $\beta = \mathsf{set}\ ((\mathbb{N} \to \mathsf{var}) \times \mathsf{set\ constraint})$ be the validator state type and $\sigma = \mathbb{N} \to \mathbb{Z}$ 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.

Each validation state in the validator state consists of an address-variable mapping and a set of constraints over the variables. The 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:

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

Let $(vs : \mathbb{N} \to \mathsf{var})$ be a mapping in the validation state, $(asgn : \mathsf{var} \to \mathbb{Z})$ be an assignment of variables, then $(vs^{asgn} : \mathbb{N} \to \mathbb{Z})$ 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:

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

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.

### 3.3.2. Proving rejection soundness

**Lemma 3.1** (RejSound-Init)**.**

$$\begin{aligned} &\text{If:} \quad vs = (\_ \mapsto \#0) \qquad cs = \{\#0 = 0\} \qquad s = (\_ \mapsto 0) \\ &\text{Then:} \quad \{(vs, cs)\} \sim s \end{aligned}$$

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

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

By constructing the assignment as: $asgn = (\_ \mapsto 0)$

We have: $\#0^{asgn} = 0$

Thus: $asgn$ satisfies $cs$

We also know that:

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

Therefore: $vs^{asgn} = s$ □

**Lemma 3.2** (RejSound-Step)**.**

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

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

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

The $\mathsf{vstep}'_p$ 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 $s'$:

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

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 deduce the satisfying assignment $asgn'$ by computing each write operation's source value and destination variable. □

### 3.3.3. Proving rejection completeness

**Lemma 3.3** (RejComplete-End)**.**

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

*Proof.* Since the $\mathsf{vstep}_p$ 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 $\mathsf{vstep}'_p$ checks that

(solvable $cs'$), we know that:

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

Let: $s' = vs'^{\,asgn}$

Then we have:

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

i.e., $v' \sim s'$ □

**Lemma 3.4** (RejComplete-Step)**.**

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

*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:

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

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

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

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

$$asgn \text{ satisfies } cs$$

Let: $s = vs^{asgn}$

Then we have:

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

i.e., $v \sim s$

From the definition of $\mathsf{vstep}'_p$, the validator first creates a fresh variable to represent the server's internal choice, so we can deduce the internal choice from the assignment:

$$x_c = \mathsf{fresh}\ (vs, cs) \qquad c = asgn!x_c$$

Now we need to show that the server's loop body $\mathsf{sstep}_p(q, c)(s)$ 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:

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

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 deduced 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)**.**

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

*Proof.* The requirement for $s$ says:

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

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, a data structure for modelling programs' asynchronous interactions in the Coq proof assistant. ITree provides more expressiveness than QAC, and allows specifying the external nondeterminism caused by the network environment. The ITree-based specifications are derived into tester programs that can interact with the SUT and reveal potential defects.

Figure 4.1 illustrates the derivation framework from ITrees to testers. Section 4.1 introduces the ITree language that encodes each box in the framework. Section 4.2 and Section 4.3 address internal and external nondeterminism in the ITree context, and interprets the "server model" into a "nondeterministic tester". Section 4.4 then explains how to execute the nondterministic tester model as an interactive tester program that runs on deterministic machines.

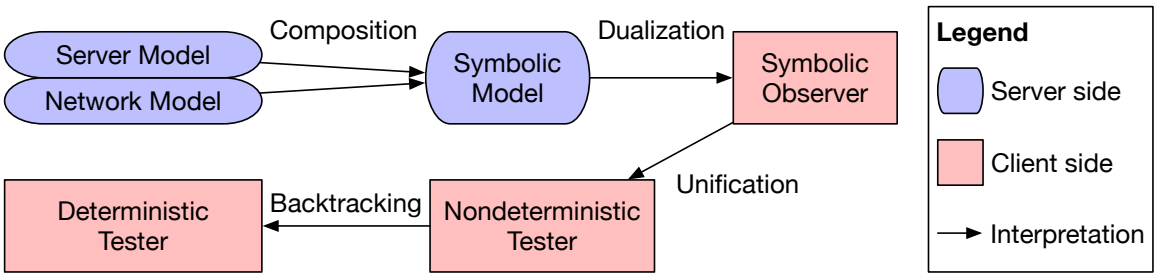



Figure 4.1: Deriving tester program from specification

## 4.1. From QAC to Interaction Trees

To write specifications for protocols' rich semantics, I employed "interaction trees" (ITrees), a generic data structure for representing interactive programs in the Coq programming language, introduced by Xia *et al.* [39]. I provide a brief introduction to the ITree specification language in Subsection 4.1.1.

ITrees allow specifying protocols as monadic programs that model valid implementations' possible behavior. In Subsection 4.1.2, I show how to embed specifications in the QAC family in terms of interaction trees.

The derivation from server specifications to interactive testers is by *interpreting* ITree programs, which corresponds to the arrows in Figure 4.1. In Subsection 4.1.3, I explain the mechanism of interpretation by deriving testers from deterministic server models. The rest of this chapter then shows how to handle nondeterminism when interpreting ITrees.

### 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 using the ITrees datatype (left), with a reference in C (right) as:

```
CoFixpoint echo :=
  c ← getchar;;
  if c is EOF
  then EXIT
  else
    putchar c;;
    echo.
```

```
void echo() {
  const char c = getchar();
  if (c == EOF)
    return;
  else {
    putchar(c);
    echo();
  }
```

```
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 4.2: Mock definition of interaction trees.

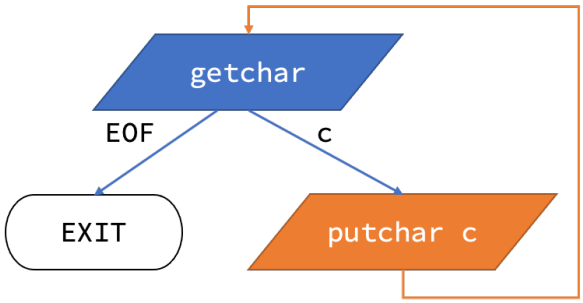



Figure 4.3: Interaction tree for echo program

Here the behavior after `getchar` depends on the value actually read. The monadic computation in Coq can be desugared into the following code, which uses the `Bind` constructor to represent sequential composition of programs:

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

Here, the first argument of `Bind` is program that returns some value and the second argument is a *continuation* that represents the subsequent computation that depends on the value returned by the first argument. 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 simplified 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 4.2, 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:

```
Definition EXIT := Ret 0.
```

- `(Trigger e)` performs an impure event `e` and returns its result. Here `(e: E 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: itreeM E 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 4.3, 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 `CoInductive` keyword indicates that type `itreeM` can represent infinite data. For example, the aforementioned `echo` program (also illustrated in Figure 4.3) is an instance of `(itreeM stdioE Z)` defined with keyword `CoFixpoint`, where `Z` indicates that the program's result value (if it ever returns by `EXIT`) is of type integer.

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.

**Practical interaction trees** Instead of binding a program—which might have many events—to a continuation, ITree binds a single impure event, as shown in Figure 4.4. I use

```
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 4.4: Formal definition of interaction trees (simplified)

`(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.
```

### 4.1.2. QAC in ITrees

The ITree specification language is a superset of the QAC language family. Any server model in Subsection 2.2.1 can be translated into an interaction tree that receives requests and sends responses.

The deterministic server's event type is defined as follows:

```
Variant ioE (I O: Type) : Type → Type := (* I/O event *)
  Recv: qaE I                    (* receive an input  *)
| Send: O → qaE unit.            (* send    an output *)
```

[1] For 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.

```
Definition qaE := ioE Q A.
```

Here `ioE` is a parameterized event that takes types `I` and `O` as arguments. Type `qaE` is an instance of I/O event that receives requests of type `Q` and sends responses of type `A`.

Given a loop body `sstep` and an initial state `s`, we can define the interaction tree of the server model as a recursive program:

```
CoFixpoint detServer (sstep: Q → σ → A * σ) (s: σ) : itree qaE void :=
  q ← trigger Recv;;
  let (a, s') := sstep q s in
  trigger (Send a);;
  detServer sstep s'.
```

The server program iterates over state `(s: σ)`. In each iteration, it receives a request `(q: Q)` and computes the response `(a: A)` using the state monad function `sstep`. It then sends back the response and recurses with the post state `(s': σ)`.

To specify a nondeterministic server with internal choices of type $C$, I introduce another event to represent its internal choices.

```
Variant choiceE: Type → Type :=
  Choice: choiceE C.    (* make an internal choice *)

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

Here `qacE` is a sum type of `qaE` and `choiceE` events, meaning that the server's events are either sending/receiving messages or making internal choices.

At the beginning of each iteration, the nondeterministic server makes an internal choice `(c: C)` as the seed of its `sstep` function:

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

Such server models can be derived into tester programs by interpretation.

### 4.1.3. Interpreting interaction trees

To interpret an ITree is to specify the semantics of its events. For example, the `tester` program in Section 1.1 can be constructed by interpreting a deterministic server model in Subsection 4.1.2.

**Tester event type** An interactive tester generates requests and sends them to the SUT, receives responses, and validates its observation against the specification:

```
Variant genE: Type → Type :=
  Gen : genE Q.          (* generate a request *)

Variant exceptE: Type → Type :=
  Throw: ∀ X, exceptE X. (* throw an exception *)

Definition testE := ioE A Q ⊕ genE ⊕ exceptE.
```

This type definition is pronounced as: "The tester receives responses `A` as input, sends requests `Q` as output, generates requests to send, or throws an exception when it detects a violation in its observations.

**Dualizing server events** The tester is constructed by dualizing the server specification's behavior: When the specification receives a request, the tester generates a request and sends it to the SUT; When the specification sends a response, the tester expects to receive the response from the SUT. We can write this correspondence as a function from server events to the tester's behaviors:

```
Definition dualize {R} (e: qaE R) : itree testE R :=
  match e with
  | Recv   ⇒ q ← trigger Gen;;
             trigger (Send q);;
             ret q
  | Send a ⇒ a' ← trigger Recv;;
             if a' =? a
             then ret tt
             else trigger (Throw ("Expect " ++ a ++ " but observed " ++ a'))
  end.
```

The `dualize` function takes an event and yields an ITree program. It maps the server's receive event to the tester's generate and send operations, and vice versa, maps the server's send event to the tester's receive operation. If the tester's received response differs from that

specified by the server model, then the tester throws an exception to indicate a violation it detected.

**Applying event handlers** The `dualize` function is also called a *handler*. It produces an ITree that has the same result type as its argument event. Therefore, we can construct the tester by substituting the server's events with the operations defined by the handler function. This process of substituting events with the results of handling them is called *interpretation*, and it is implemented as shown below:

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

Definition tester (sstep: Q → σ → A * σ) (s: σ) : itree testE void :=
  interp dualize (detServer sstep s).
```

Applying the dualization handler to the server model gives us an ITree program that performs tester interactions. The tester keeps generating, sending and receiving messages until observing an unexpected response when it throws an exception and rejects the SUT. It is semantically equivalent to the `tester` in Section 1.1.

To derive testers from nondeterministic server models, I design more complex tester events and handlers, as discussed in the following sections.

## 4.2. Handling Internal Nondeterminism

This section applies the idea of dualization in Chapter 3 to the ITree context, showing how to address internal nondeterminism by symbolic evaluation based on ITree specifications. It covers the derivation path from "symbolic model" to "nondeterministic tester" in Figure 4.1, using HTTP/1.1 entity tags introduced in Subsection 1.2.1 as an example.

As discussed in Section 3.1, dualization requires refining the representation of the server's computation, e.g., encoding its branches over symbolic conditions. This is done by designing ITrees' event types in Subsection 4.2.1 and specifying the server's behavior with a symbolic

model.

The server specification is derived into a tester client by *interpreting* interaction trees. To interpret is to define semantic rules that transform one ITree program into another, and corresponds to the arrows in Figure 4.1. Subsection 4.1.3 explains the interpretation of ITrees.

The interpretation from symbolic model to the nondeterministic tester model is implemented in two phases, illustrated as "dualization" and "unification" arrows in Figure 4.1: Subsection 4.2.2 dualizes the server's behavior into the tester client's, resulting in a "symbolic observer" that encodes symbolic evaluation as primitive events. Subsection 4.2.3 then instantiates the primitive events into pure computations that unify concrete observations against their symbolic representations.

### 4.2.1. Symbolic server model

The server specification is an ITree program that exhibits all valid behavior of the protocol. I combine the Prog language in Subsection 3.1.2 with the simple ITree `ioE` events for sends and receives as we saw in Subsection 4.1.2.

**Network packet type** Instead of receiving requests and sending responses, the server receives and sends *packets* that carry routing information. This will allow us to specify the server's interaction against concurrent clients. A packet consists of headers that indicate its source and destination, and a payload of either a request or a response:

```
Notation connection := N. (* N for natural number *)

Record packet Q A := {
  Source      : endpoint;
  Destination : endpoint;
  Payload     : Q + A
}.
```

This type definition says: the `packet` type is parameterized over the `Q` and `A` types that represent its request and response. Its `Source` and `Destination` fields each records an `endpoint` represented as a natural number. Its `Payload` type is the sum of request and response.

Here's an example trace of network packets that represents a simple HTTP exchange:

```
Context get: string → request.
Context ok : string → response.

Definition server_end: endpoint := 0.

Example trace: list (packet request response) :=
  [ { Source      := 1;
      Destination := server_end;
      Payload     := inl (get "/index.html")
    }
  ; { Source      := server_end;
      Destination := 1;
      Payload     := inr (ok "<p>Hello!</p>")
    }
  ].
```

This trace encodes a transaction between client 1 and the server (represented as endpoint 0). The client sends a GET request to fetch the resource in path `"/index.html"`, and the server responds with 200 OK and content `"<p>Hello!</p>"`. The `inl` and `inr` are constructors for sum types that here are used to distinguish requests from responses:

```
Context inl: ∀ {X Y}, X → X + Y.
Context inr: ∀ {X Y}, Y → X + Y.
```

**Symbolic representation** To specify systems' nondeterministic behavior, the Prog language in Subsection 3.1.2 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 [25] 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: choiceE (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. Here is an example of expressions of type string:

```
Context x y : var.

Example expressions: list (exp string) :=
  [ Const   "foo"
  ; Var      x
  ; Compare "bar" (Var y) Weak
  ].
```

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 comparison" or "weak comparison" mechanism[2] specified by `cmp`. The constant ETag is provided by the request, and the symbolic one comes from the server state.

Figure 4.5 shows an ITree model for If-Match requests (Subsection 1.2.1). The server state type $\sigma$ maps each path to its corresponding "resource"—file content and metadata like ETag. 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'` is permitted to be any value, so the model represents it as `Choice` event.

Note that the server model exhibits three kinds of branches:

1. The `if` branch in Line 19 is provided by ITree's embedding language Coq, and takes a boolean value as condition;

2. The `IFX` branch in Line 17 constructs an ITree that nondeterministically branches over a condition written as a symbolic expression of type bool:

[2]HTTP/1.1 servers may choose to generate ETags as "strong validators" (with uniqueness guarantee) or "weak validators" (for potentially better performance). Weak validators have prefix `W/` while strong validators do not. When handling compare-and-swap operations such as PUT requests conditioned over `If-Match` in Subsection 1.2.1, the server should evaluate its precondition with "strong comparison" that doesn't allow weak validators, e.g., `W/"foo"`) to match any ETag including itself. For GET requests conditioned over `If-None-Match`, the server may evaluate with "weak comparison" where a weak validator like `W/"bar"` matches itself and also matches strong validator `"bar"`, but doesn't match `W/"foo"` or `"foo"`.

```
1   Notation σ := (path → resource).
2
3   Context OK PreconditionFailed : symbolic_response.
4   Context process: request → σ → itree smE (symbolic_response * σ).
5
6   CoFixpoint server_http (state: σ) :=
7     pq ← trigger Recv;;
8     let respond_with a :=
9       trigger (Send { Source      := server_conn
10                    ; Destination := pq.(Source)
11                    ; Payload     := inr a } ) in
12    let q : request    := request_of pq        in
13    let v : string     := q.(Content)          in
14    let k : path       := q.(TargetPath)       in
15    let t : string     := if_match q           in
16    let tx: exp string := (state k).(ETag)     in
17    IFX (Compare t tx Strong)
18    THEN
19      if q.(Method) is Put
20      then
21        tx' ← or (trigger Choice)
22                 (Pure (Const EmptyString));;
23        let state' := state [k ↦ {Content := v; ETag := tx'}] in
24        respond_with OK;;
25        server_http state'
26      else                  (* handling other kinds of requests *)
27        (a, state') ← process q state;;
28        respond_with a;;
29        server_http state'
30    ELSE
31      respond_with PreconditionFailed;;
32      server_http s.
```

Figure 4.5: Server model for HTTP conditional requests

```
Variant branchE: Type → Type := (* symbolic-conditional branch *)
  Decide: exp bool → branchE bool.

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

3. The `or` operator in Line 21 takes two ITrees `x` and `y` as possible branches and constructs a nondeterministic program that may behave as either `x` or `y`:

   ```
   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 `nondetE` is a special case of `choiceE` 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 (qaE ⊕ nondetE ⊕ choiceE ⊕ branchE) void)`.

These three 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"—a request that doesn't involve symbolic variables, as opposed to "symbolic" ones—generated by the client 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 role 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.

3. The `or` branch defines multiple control flows the server may take. In the HTTP example, the server may generate an ETag for the resource's new content, but is not obliged to do so. It may choose to generate no ETags instead, using `(Pure (Const EmptyString))`

as an alternative to `(trigger Choice)`.

In addition to `IFX` branch conditions, the symbolic expressions may also appear in the server's responses. For example, after generating an ETag in Line 21 of Figure 4.5, the server may receive a GET request and send the ETag to the client:

```
Example ok_with_etag: symbolic_response :=
  { ResponseLine := { Version := { Major := 1; Minor := 1 }
                    ; Code    := 200
                    ; Phrase  := OK
                    }
  ; ResponseFields :=
    [ { Name := "Content-Length"; Value := Const "13" }
    ; { Name := "ETag"          ; Value := Var    x   }
    ]
  ; ResponseBody := "<p>Hello!</p>"
  }.
```

Suppose the server generated the ETag as expression `(Var x)`, then we can use the expression to construct the symbolic response in the specification, rather than determining its concrete value. The mechanism of producing expressions for ETags is explained in Subsection 4.2.2.

Now we can define the specification's event type `smE`. The symbolic server model receives concrete requests and sends symbolic responses, so its event is defined as:

```
Definition symbolic_packet := packet request symbolic_response.

Definition qaE := ioE symbolic_packet symbolic_packet.

Notation smE := (qaE ⊕ nondetE ⊕ choiceE ⊕ branchE).
```

The "Symbolic Model" in Figure 4.1 is an ITree constructed by applying the `server_http` function to an initial state:

```
Definition sm_http: itree smE void :=
  server_http init_state.
```

The rest of this section will explain the interpretations from this symbolic model.

### 4.2.2. Dualizing symbolic model

This subsection takes the symbolic model composed in Subsection 4.2.1 and dualizes its interactions, which corresponds to the "Dualization" arrow in Figure 4.1. It applies the idea of derivation rules (1)–(4) for Prog (Section 3.2) to models written as ITrees.

```
1  Notation oE := (observeE ⊕ nondetE ⊕ choiceE ⊕ constraintE).
2
3  Definition observe {R} (e: smE R) : itree oE R :=
4    match e with
5    | Recv      ⇒ trigger FromObserver
6    | Send px   ⇒ p ← trigger ToObserver;;
7                  trigger (Guard px p)
8    | Decide bx ⇒ or (trigger (Unify bx true);;  ret true)
9                    (trigger (Unify bx false);; ret false)
10   | Or        ⇒ trigger Or
11   | Choice    ⇒ trigger Choice
12   end.
13
14 Definition observer_http: itree oE void :=
15   interp observe sm_http.
```

Figure 4.6: Dualizing symbolic model into symbolic observer.

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 4.6 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.

```
Definition concrete_packet := packet request concrete_response.

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

Note 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" to represent the creation of constraints primitively.

```
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 unifies the symbolic branch condition `bx` with boolean value `b`. These constraints will be solved in Subsection 4.2.3.

The dualization algorithm in Figure 4.6 does the follows:

1. When the symbolic model receives a packet, in Line 5, the observer generates a request packet;

2. When the symbolic model sends a symbolic packet `px`, in Line 6, the observer receives a concrete packet `p`, and adds a `Guard` constraint that restricts the symbolic and concrete packets to 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 `Unify` 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 4.2.3, along with the constraints created in this phase.

The result of dualization is a symbolic observer that models tester behaviors like sending requests and receiving responses. The symbolic observer is a nondeterministic program with

```
Example observer_body: itree oE void :=
  let guard_response a :=
    p ← trigger ToObserver;;
    trigger (Guard { Source      := server_conn
                   ; Destination := pq.(Source)
                   ; Payload     := inr a } ) in
  let bx: exp bool := Compare t tx Strong in
  or (
      trigger (Unify (Compare bx true));;
      if q.(Method) is Put
      then
        tx' ← or (trigger Choice)
                 (Pure (Const EmptyString));;
        let state' := state [k ↦ {Content := v; ETag := tx'}] in
        guard_response OK;;
        interp observe (server_http state')
      else
        (a, state') ← interp observe (process q state);;
        guard_response a;;
        interp observe (server_http state')
    )
    (
     trigger (Unify (Compare bx false));;
     guard_response PreconditionFailed;;
     interp observe (server_http s)
    ).
```

Figure 4.7: Symbolic observer example.

primitives events like making choices and adding constraints over the choices.

For example, dualizing Line 17–32 in Figure 4.5 results in an observer program that is equivalent to Figure 4.7. Dualization transforms the `sm_http` specification into the `observer_http` program in Figure 4.6 automatically, and includes more details than the hand-written `observer_body` example. The next subsection interprets the observer's primitive `Guard` and `Unify` events into pure computations of symbolic evaluation.

### 4.2.3. Symbolic evaluation

This subsection takes the symbolic observer produced in Subsection 4.2.2 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.

```
1  Notation ntE := (observeE ⊕ nondetE ⊕ 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 ⇒ let x: var := fresh v in
9                ret (x::xs, cs, Var x)
10   | (constraint: unifyE) ⇒ let cs' := constraint::cs in
11                             if solvable cs'
12                             then ret (xs, cs', tt)
13                             else Trigger (Throw ("Conflict: " ++ print cs'))
14   | Or              ⇒ b ← trigger Or;; ret (v, b)
15   | (oe: observeE) ⇒ r ← trigger oe;; ret (v, r)
16   end.
17
18 Definition nondet_tester_http: itree ntE void :=
19   (_, vd) ← interp_state unify observer_http initV;;
20   match vd in void with end.
```

Figure 4.8: Resolving symbolic constraints.

As shown in Figure 4.8, 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.

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 → Type :=
     Throw: ∀ {X}, string → 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.

Note that the `unify` function interprets a symbolic observer's event `(oE R)` into a function of type `(V → itree tE (V * R))`, a so-called "state monad transformer". It takes a pre-validation state `(v: V)` and computes an ITree that yields the observer event's corresponding result `(r: R)` along with a post-validation state `(v': V)`. This stateful interpretation process is implemented by a variant of `interp` called `interp_state`:

```
CoFixpoint interp_state {E F V R}
                        (handler: ∀ {X}, E X → V → 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 symbolic model into a tester model that observes incoming and outgoing packets, nondeterministically branches, and in some cases throws exceptions. In this section, I used the server model verbatim as the symbolic model, and the derived tester can handle internal nondeterminism by symbolic evaluation.

The server model's receive and send events are linear. It doesn't receive new requests before responding to its previous request. As a result, the tester dualized from the server model doesn't send new requests before it receives the response to the previous request, so it doesn't reveal the server's interaction with multiple clients simultaneously. In the next section, I'll explain how to create a multi-client tester by extending the symbolic model, addressing external nondeterminism.

### 4.3. 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,

using the networked server example. It corresponds to the "Composition" arrow in Figure 4.1. Subsection 4.3.1 defines a model for concurrent TCP connections. Subsection 4.3.2 then composes the network model with the server specification, yielding a tester-side specification that defines the space of valid observations.

### 4.3.1. Modelling the network

When testing servers over the network, request and response packets may be 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[3] and later emits them:

```
Notation packet := symbolic_packet.

Definition netE: Type → Type :=
  ioE packet packet.

Notation Absorb := Recv.
Notation Emit   := Send.
```

For example, the network model for concurrent TCP connections is defined in Figure 4.9. 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)` by choosing the left branch. In this case, the wire can emit packet `p`.

[3] In this section, "packet" is a shorthand for the "symbolic packet" defined on Page 56.

```
(* filter the oldest packet in each connection *)
Context oldest_in_each_conn : list packet → list packet.

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.

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 4.9: 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 `absorb`s or `emit`s 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.

> Or the function might choose the right branch and recursing on `l'`, meaning that the wire can emit some packet in `l'` or absorb some packet into the buffer.

This 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.

### 4.3.2. 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 at some given moment to absorb packets sent by the server. The network might also emit a packet before the server is ready to receive it.

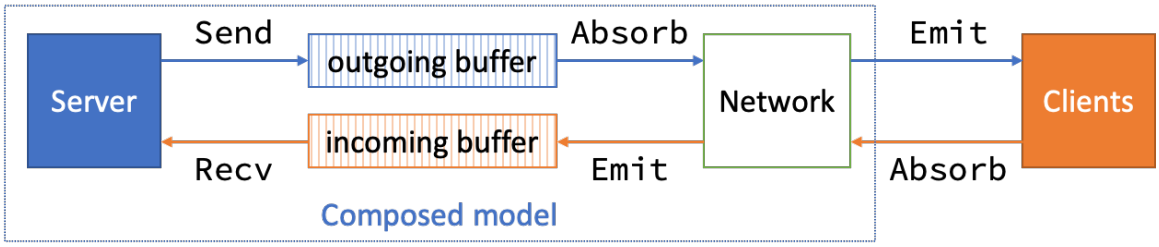



Figure 4.10: Network composition architecture

To handle the asynchronicity among the server and network events, I insert message buffers between them. As shown in Figure 4.10, 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 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:

```
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 4.11. The function takes the symbolic server model in Subsection 4.2.1 and the network model in Subsection 4.3.1, and yields a symbolic model of the server's behavior observable from across the network.

The composition function analyzes the server and the network's behavior:

1. When the server wants to send a packet in Line 29, the packet is appended to the outgoing buffer.

2. When the network wants to absorb a packet in Line 9, it first checks whether the server has sent some packet to its outgoing buffer. If yes, then the network absorbs the oldest packet in the buffer. Otherwise, it absorbs from the clients.

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

Figure 4.11: 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, a packet sent by the client is 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.

3. After absorbing some packets, when the network wants to emit a packet in Line 13, the packet is either emitted to the client or appended to the server's incoming buffer, based on its destination.

4. When the server wants to receive a packet in Line 24, it first checks whether the network has emitted some packet to the incoming buffer. If yes, then the server takes the oldest packet in the buffer. Otherwise, it waits for the network model to emit one.

Note 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. This design is to avoid divergence of the derived tester program, which I'll further explain in Section 4.4.

By composing the server and network models, we get a symbolic model that may send and receive packets asynchronously, as opposed to the server model that processes one request at a time. Dualizing this asynchronous symbolic model results in an asynchronous tester model that may send multiple requests simultaneously rather than waiting for previous responses. Next I'll show how to execute the tester's ITree model against the SUT.

## 4.4. Executing the Tester Model

This section takes the nondetermistic tester model derived in Subsection 4.2.3 and transforms it into an interactive program. Subsection 4.4.1 handles the nondeterministic branches via backtracking and produces a deterministic tester model. Subsection 4.4.2 then interprets the deterministic tester into an 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 4.1.

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 4.12 shows the backtracking algorithm. It interacts with the SUT and checks whether 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`:

```
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 (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.

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

Figure 4.12: 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: observeE concrete_packet) 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 ⇒
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 4.13: Matching tester model against existing observation.

3. When the tester wants to observe a packet *from* itself, it generates a packet and sends it to the SUT in Line 15.

   Note 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 recognize this sent packet and check whether future interactions are valid follow-ups of it. This is done by matching the branches against the send event, using the `expect` function.

   As shown in Figure 4.13, `(expect e r l)` matches every tester in list `l` against the observation `e` that has return value `r`. For each element `m` ∈ `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, 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 that is 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 4.11, where the server model is scheduled at a higher priority than the network model. Suppose the SUT has produced some invalid output. Then the tester should reject its observation by throwing an exception in every branch. However, the network model is always ready to absorb a packet, because the `pick_one` function on Page 63 always includes a branch that returns `None`. If the composition algorithm prioritizes the network model over the server model, then when one branch rejects, the derived tester backtracks to other branches, which includes generating and sending another packet to the SUT (dualized from the network model's absorbtion event). 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, instead of continuously sending request packets.

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.

```
1  Fixpoint execute (fuel: nat) (m: itree tE void) : IO bool :=
2    match fuel with
3    | O       ⇒ 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 4.14: Interpreting ITree tester to IO monad.

### 4.4.2. From ITree model to IO program

The deterministic tester model derived in Figure 4.12 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 4.14, 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 [38], which are eventually extracted into OCaml programs that can be compiled into executables that

can communicate with the SUT over the operating system's 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 nondetermism, 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 inputs that reveal violations effectively", and unveil the techniques behind `gen_packet` in Line 13 of Figure 4.14.

# 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. In this chapter, I present a language-based design for the test harnesses, showing how to generate and shrink test inputs effectively and address 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 a shrinking mechanism that keeps the test inputs interesting among different executions.

### 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.

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 them into abstract messages.

A generator is a randomized program that produces test inputs. One example is the `gen_packet` function in Figure 4.14. The HTTP packet generator can be naïvely implementation as shown in Figure 5.2. This version fills in the request's fields with arbitrary values, and has limited coverage of the SUT's behavior. This is because the request tar-

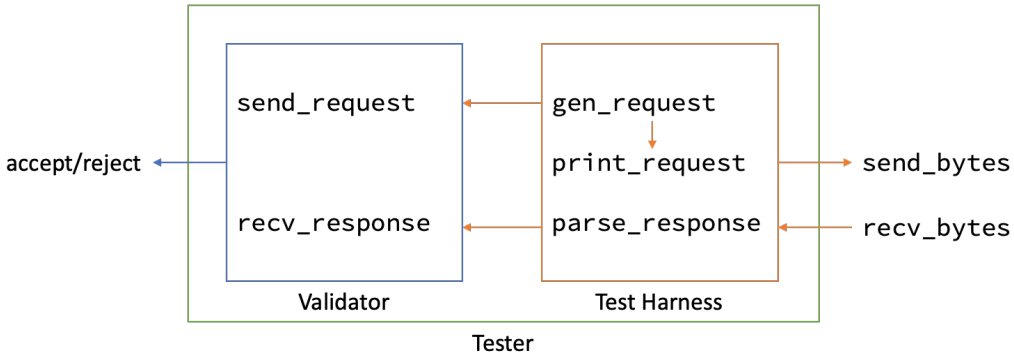



Figure 5.1: Tester architecture outline.

```
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.

get and ETags are both generated randomly, but a request is interesting only if its ETag matches its target's corresponding resource stored on the server. A randomly generated request would result 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 the set of paths where the server has stored resources, then it can generate more paths within the set to hit the existing 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

**Motivation** 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 $\sigma$ from `(path → resource)` in Figure 4.5 into `(list (path * resource))`, which allows the generator to access the list of all `paths` in the server state. The generator chooses from these existing 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` function to carry the server state to instantiate `gen_path`.

As shown in Figure 4.12, the `GenPacket` generator is triggered when the tester wants to observe a packet from itself to the SUT. Figure 4.6 then shows that such `FromObserver` expectation happens when the symbolic model `Send`s a packet. Such a `Send` event only happens when the server wants to receive a packet in Figure 4.10. The `Recv` events are triggered by the server model in Figure 4.5, which iterates over the server state $\sigma$.

**Implementation** To expose the server state to the request generator, I extend the symbolic server model's `Recv` event type on Page 56 to include the server state:

```
Variant qaE: Type → Type :=
```

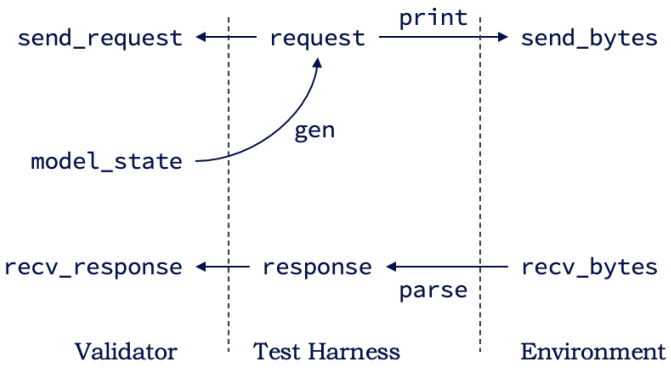



Figure 5.3: State-based heuristics.

```
  Recv : σ       → 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: σ      → netE packet.

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

Variant genE: Type → Type :=
  GenPacket : σ → genE concrete_packet
| GenBool   : genE bool.

Definition gen_packet: σ → IO concrete_packet.
```

As a result, when instantiating the `(GenPacket state)` event in Figure 4.14, we can feed the `gen_packet` function with argument `state`, so that `gen_path` can generate interesting paths based on the server state. Figure 5.3 illustrates this state-based heuristics. It refines the test harness box in Figure 5.1, and will be extended with trace-based heuristics in the next subsection.

### 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 4.14, adding an accumulator called "`trace`" 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 4.14 *)
```

[LYS: `trace ++ [q]` is indeed less inefficient than storing a reversed trace, but this code is

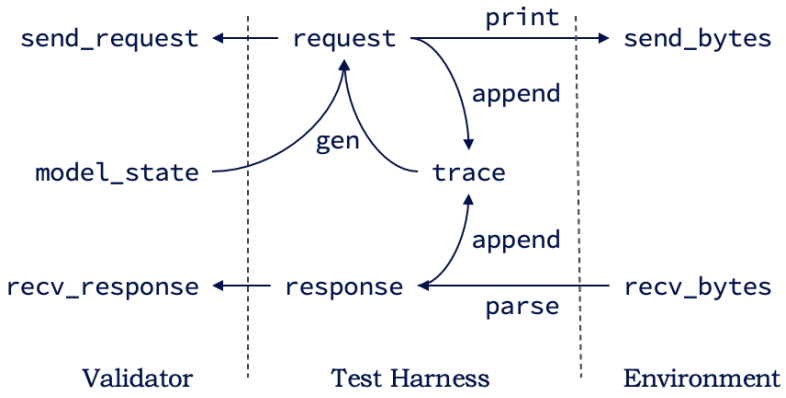



Figure 5.4: Combining state-based and trace-based heuristics.

to show the concept of "appending" requests and responses to a trace, and assumes that readers have the ability of optimizing its performance.]

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: σ) (trace: list concrete_packet) :=
  target ← gen_path state;;
  etag   ← gen_etag trace;;
  ... (* same as Figure 5.2 *)
```

Now we can generate interesting test inputs by combining state-based and trace-based heuristics, as shown in Figure 5.4. In the next section, I'll further extend this framework and 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) *signal* that is essential to triggering violations, and (2) *noise* that does not contribute to revealing such violations. We need to shrink the counterexample by removing the noise and keeping the signal.

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. How-

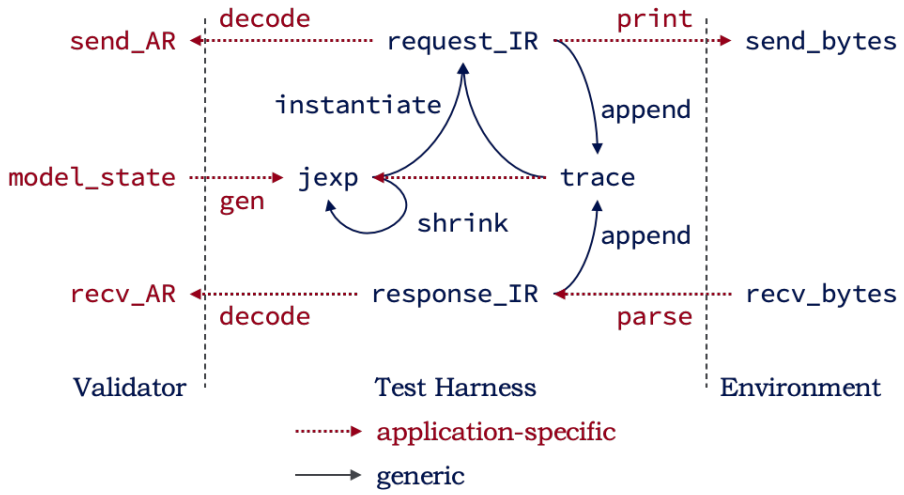



Figure 5.5: Complete architecture of the test harness.

ever, rerunning an interesting request might produce trivial results, due to inter-execution nondeterminism discussed in Subsection 1.3.2.

To prevent turning signal into noise 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 embeds 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 the concrete value, e.g., "Wed, 8 Jun 2022 07:20:29 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., " Thu, 9 Jun 2022 07:20:29 GMT".

The test generation and shrinking framework is shown in Figure 5.5. It involves four lan-

guages, 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.

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, which correspond to the blue solid arrows in Figure 5.5. It can be used for testing various protocols, provided application-specific translations from IR to AR and between IR and bytes, illustrated as red dotted arrows. The test developer needs to tune the generator that produces Jexps, encoding their domain knowledge as state-based

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

Figure 5.6: Intermediate representation and J-expression.

```
Notation   labelT := nat.
Definition traceT := list (labelT * IR).

Context q1 q2 a1 a2 : IR.
Example labelled_trace: traceT :=
  [(1, q1); (3, q2); (4, a2); (2, a1)].
```

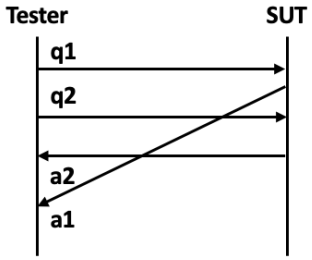



Figure 5.7: Labelled trace example.

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.

**Syntax** The J-expression is an extension of JSON that can encode trace-based heuristics. As shown in Figure 5.6, a Jexp may include syntax (*label*.Jpath.*function*) that represents trace-based heuristics, specified as:

1. The test harness records the trace as a list of labelled messages, where the requests are labelled odd, and their responses are labelled as the next even number. The *label* in a Jexp locates the IR in the trace with which the heuristics computes the input. Labelling messages allows the reproducing trace-based heuristics despite shrinking and inter-execution nondeterminism.

   For example, consider the trace in Figure 5.7: If a trace-based heuristics is interested in

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

```
(* Jpath syntax defined in Figure 5.6. *)
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 5.8: 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 5.8. The response IR is a JSON object whose `"files"` field is an array of objects, each has a `"name"` and a `"mode"` field. A heuristic 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 → 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 heuristic wants to add write permission to the mode bits, it can do so with the `mode_add_write` function in Figure 5.8, which produces mode 700. Some heuristics might use the sub-IR 500 as-is, using the identity function `id`.

**Semantics** 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 occurences 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 5.7, with `a2` is defined in Figure 5.6, 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:

   ```
   (* j2 = *)
   [
     { "foo": 21 },
     { "bar": 22 }
   ]
   ```

   ```
   (* j3 = *)
   [
     { "bar": 31 },
     { "baz": 32 },
     { "foo": 33 }
   ]
   ```

   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 `l` and Jpath `p` on a trace, if the message labelled `l` 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 t2: traceT :=
     [(3,q1); (4,j3); (5,q2); (6,a2)].
   ```

Here `t1` doesn't have a message labelled 6, probably caused by environment delays; `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 `t1`, and `33` for `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 question is whether the tester can reveal the SUT's violation against the specification.

## 6.1. Testing Web Servers

This thesis is motivated by the Deep Specification project [2], whose goal is to build systems with rigorous guarantees of functional correctness, studying HTTP as an example. I formalized a subset of HTTP/1.1 specification, featuring WebDAV requests GET, PUT, and POST [10], ETag preconditions [13], and forward proxying [12].

From the protocol specification written as ITrees, I derive 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 [11], one of the most popular servers on the World Wide Web [27, 37]. I used the latest release 2.4.46, and edited its configuration file to enable WebDAV and proxy modules.
- Nginx [32], 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 in Coq.

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 DeepWeb server was compiled into an executable binary, and also ran on localhost.

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

### 6.1.2. Qualitative Results

**Apache** My tester rejected the Apache HTTP Server, which uses strong comparison (Page 53) for PUT requests conditioned over `If-None-Match`, while RFC 7232 specified that `If-None-Match` preconditions must be evaluated with weak comparison. I reported this bug to the developers, and figured out that Apache was conforming with an obsoleted HTTP/1.1 standard [14]. 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 that are known as buggy, and expected the tester to detect all the intentional bugs.

The buggy implementations were created by mutating the Apache source code manually[1] and recompiling the server program. I created 20 mutants whose bugs were located in

[1] I 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.

```
static int default_handler(request_rec *r) {
    ...
        if (r→finfo.filetype == APR_NOFILE) {
            ap_log_rerror(APLOG_MARK, APLOG_INFO, 0, r, APLOGNO(00128)
                          "File does not exist: %s",
                          apr_pstrcat(r→pool, r→filename, r→path_info, NULL));
            // return HTTP_NOT_FOUND;
        }
    ...
```

Figure 6.1: Server mutant example, created by commenting out a line of code.

various modules of the Apache server: `core`, `http`, `dav`, and `proxy`. Some bugs appeared in the control flow, e.g., removing a return statement in the request handler (as shown in Figure 6.1) or skipping the check of preconditions. Others appeared in the data values, e.g., calling functions with wrong parameters, flipping bits in computations, accessing buffer off by one byte, etc.

The tester rejected all of the 20 mutants. Some mutants took the tester a 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 [17], reporting the same issue with `If-Unmodified-Since` preconditions.

This issue has been recognized by the developers in 2016 but never resolved. To fix this bug, Nginx needs to redesign its core logic and evaluate the request's precondition *before*—instead of *after*—handling the request itself. As a result, I tested mutants only for Apache, whose original violation was fixed by simply removing a few lines of bad code.

**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 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.

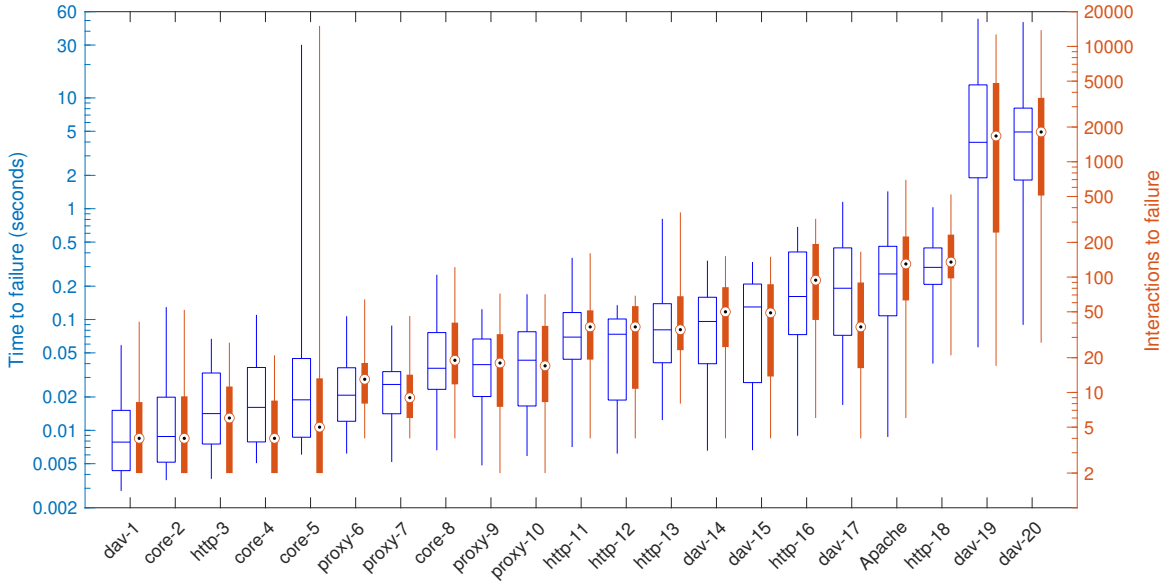



Figure 6.2: Cost of detecting bug in each server implementation. The left box with median line is the tester's execution time before rejecting the server, which includes interacting with the server and checking its responses. The right bar with median circle is the number of HTTP/1.1 messages sent and received by the tester before finding the bug. Results beyond 25%–75% are covered by whiskers.

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. The bug was revealed by manually observing the flow of network packets, where the tester kept sending requests while the server never responded. My experiments have shown the complementarity between testing and verification for improving software quality.

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.

### 6.1.3. Quantitative Results

To answer the research question at the beginning of this chapter quantitatively, I measured the execution time and network interactions taken to reject vanilla Apache and its mutants, as shown in Figure 6.2. The 20 mutants are named after the modules where I inserted the

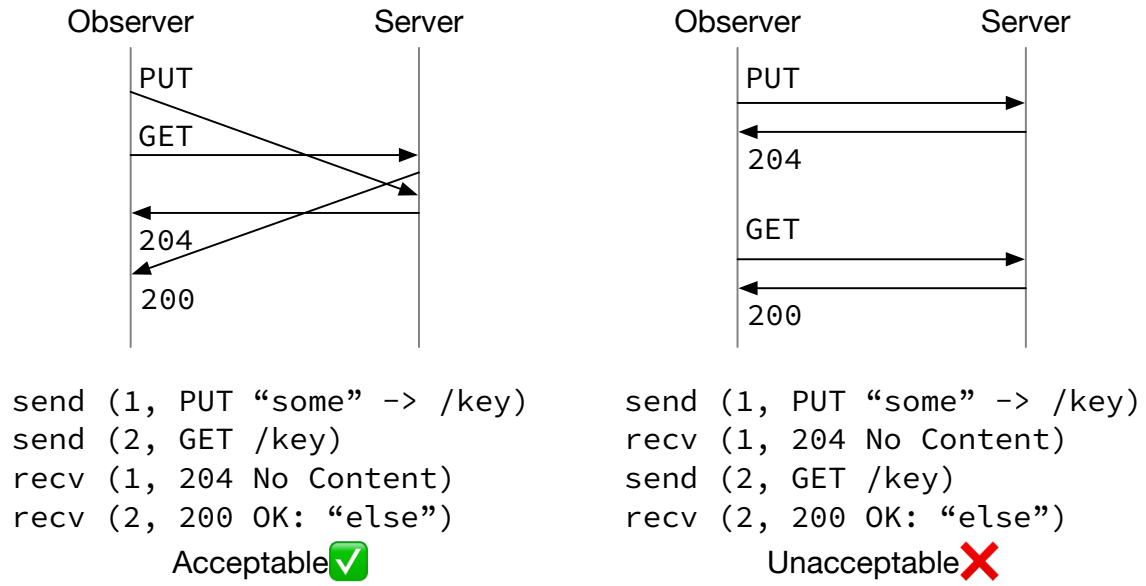



Figure 6.3: GET-after-PUT bug pattern in Apache mutants. The trace on the left does not convince the tester that the server is buggy, because there exists a certain network delay that explains why the PUT request was not reflected in the 200 response. When the trace is ordered as shown on the right, the tester cannot imagine any network reordering that causes such observation, thus must reject the server.

bugs. 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 counterexamples.

Some bugs took longer time to find, and they usually required more interactions to reveal. 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 6.3.

- 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 a 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 in Figure 6.3.

- Mutant 5 is the one shown in Figure 6.1. It causes the server to skip the return statement when the response should be 404 Not Found. The counterexample can be as small as one GET request on a non-existent target, followed by an unexpected response like 403 Forbidden. 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 the 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

```
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 6.4: File system specification.

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 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, possibly by integrating packet dynamics [30] in the test executor.

## 6.2. Testing a File Synchronizer

To demonstrate the generality of my specification-based testing methodology, I also 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 [29], 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 [5]. 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 6.4, 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 that 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 reconcilation algorithm by Balasubramaniam and Pierce [5]:

   ```
   Definition σ := node*node*node.

   Context reconcile: σ → σ.
   ```

   Note 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 reconcilation semantics, I modelled the SUT as a deterministic QA server described in Definition 2.6. As shown in Figure 6.5, 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,

```
  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 6.5: Query and response types for testing file synchronizer.

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 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 rejected the Unison file synchronizer in two ways, and I reported the counterexample to the developers. By analyzing the program's behavior, we determined that one rejection was a valid but defective feature, and the other rejection was a documented feature not included in my specification. 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 [20] 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 6.4 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 nondeterminism 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 4.3.

**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 there is a conflict

between the two replicas. My specification required Unison to indicate this conflict with exit code 1. During the tests, Unison did not notice that `B/foo.txt` was changed, decided to propagate the deletion from replica `A` to replica `B`, and halted with exit code 2, representing "non-fatal failures occured during file transfer".

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 synchronizer detects file modifications by checking if their timestamps have changed. When the tester writes to the file within a short interval, 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 or turning of the preference.

Note 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 will propagate no updates and leave the two replicas as-is, which results the same as enabling fastcheck but at lower performance.

To include the space of behaviors with fastcheck enabled or upon failing file transfers, I modified the synchronization semantics in two ways: (1) Instead of computing the set of all dirty paths and synchronizing all of them, allow choosing any subset of the dirty paths to synchronize. (2) If there is a conflict in the chosen paths, allow halting the synchronizer with exit code 1 (conflict detected) or 2 (propagation failed). The loosened specification accepted all behavior it observed from Unison.

These experiments show that my testing methodology can effectively reject implementations that do not comply with the specification. The incompliance indicates that either (i) the implementation does not conform with the protocol, e.g., Apache and Nginx violating RFC 7232, or (2) the specification is incorrect, e.g., my initial file synchronizer specification not

modelling all features of Unison. Automatically detecting the incompliances helps developers to correctly specify and implement systems in real-world practices.

# CHAPTER 7

## Related Work

This chapter explores methodologies in specifying and automatically testing interactive systems. Section 7.1 compares different specification techniques for testing purposes. 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 [18]. 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 implemented 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" motivated me to rephrase the specification. I applied the idea of model-based testing [8], 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.* [19] 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.* [21] 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 4.3 was inspired by these ideas 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 asynchronous systems: (i) To adjust the network configurations, e.g., limiting the buffer size or the number of concurrent connections, we only need to adjust the network model without modifying the server's events. (ii) Instead of carefully defining the space of valid scrambled traces for each network configuration, we can derive it from the network model by dualization. (iii) The network model is reusable and allows specifying various protocols on top of it.

### 7.1.2. Process algebra: LOTOS and TorXakis

Language of termporal ordering specification (LOTOS) [7] 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 [34] 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 inifitite 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.* [6] have developed rigorous specification for TCP, UDP, and the Sockets API. To handle internal nondeterminism in real-world implementations, they used symbolic the model states, which are then evaluated with a special-purpose symbolic model checker. The tester helped reveal their post-hoc specifications' mismatch with mainstream systems like FreeBSD, Linux, and WinXP. I borrowed the idea of symbolic evaluation in validating observations, and used it for detecting mainstream servers' violations against the formalized RFC specification.

Artho and Rousset [3] 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 modulo external nondeterminism.

## 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.

### 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 [1] allows generating data that satisfy certain constraints. Lampropoulos *et al.* [23] have applied the idea of narrowing to the QuickChick testing framework in Coq [22], representing constraints as inductive relations. The relations are compiled into efficient generators that produce satisfying data.

The narrowing-based generator in QuickChick was used during my preliminary experiments with HTTP/1.1, where I defined a well-formed relation for HTTP requests to guide the generator. However, the QuickChick implementation was unstable and failed to derive generators as my type definition becomes more and more complex. This constraint-based heuristics may be integrated to my current testing framework by interpreting QuickChick's `Gen` (generator) monad into the `IO` monad used by the test harness, provided the generator derivation failure gets resolved.

**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 [26], with popular implementations AFL [40] and Honggfuzz [31], and combined with property-based testing by Lampropoulos *et al.* [24].

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 by the comtemporary fuzzing tools, which only checks whether the SUT crashes or hangs and does not care about functional correctness like the responses' contents. 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 [18] 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, which requires them to define the state transition rules for each interaction. The

heuristics are implemented based on the tester's point of view.

In comparison, my framework replaced these requirements with state- and trace-based heuristics, where the model state was exposed by the underlying specification, and the trace was automatically recorded by the test harness. This allows test developers to design heuristics based on the implementation's point of view, instead of reasoning on the implementation's internal and external nondeterminism based on its observations.

# CHAPTER 8

## Conclusion and Future Work

This dissertation presented a systematic technique for testing interactive systems with uncertain behavior. I proposed a theory of dualizing protocol specification into validators, with formal guarantee of soundness and completeness (Chapter 2). To test systems in real-world practice, 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 was evaluated by detecting programs' incompliance with the specification using automatically derived testers (Chapter 6).

To address challenges posed by internal, external, and inter-execution nondeterminism, I introduced various flavors of symbolic abstract interpretation. Systems' internal choices are represented as symbolic variables and unified against the tester's observations (Section 3.2, Section 4.2). Possible impacts made by the environment are represented as nondeterministic branches (Section 4.3) 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 techniques in this thesis can be expanded in several dimensions:

**Specifying and testing in various scenarios** The Unison file synchronizer in Section 6.2 was specified and tested as a command line tool that is manually triggered. Whereas, Unison can run as a background process that monitors the file system and automatically propagates the updates, just like Dropbox. Running in the background introduces more nondeterminism, e.g., whether the tester's modification has been buffered or propagated to the file system, whether the operating system has scheduled the synchronizer process to check the file system's updates, whether the synchronizer has finished propagating the changes to the other replica, *etc.* Migrating testing scenarios might expose the limitations

of my methodology or introduce new challenges in testing.

**Integrating other testing techniques to the framework** The experiments with Apache and Unison have shown that some bugs are time-related, e.g., only revealed by specific request and response orders, or require conflicting modifications to occur within the same millisecond. My current test harness doesn't tune the timing of executions, and might benefit from integrating packet dynamics by Sun *et al.* [30].

The randomized generator in my framework produced a large number of inputs with guidance of heuristics based on domain-specific knowledges. To find bugs with fewer tests, Goldstein *et al.* [16] applied ideas from combinatorial testing to tune the generators' distributions for better coverage. Measuring and improving coverage of asynchronous tests is yet to be studied.

# BIBLIOGRAPHY


[1] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. *J. ACM*, 47(4):776–822, July 2000. ISSN 0004-5411. doi: 10.1145/347476.347484.

[2] 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, October 2017. ISSN 1364-503X. doi: 10.1098/rsta.2016.0331.

[3] Cyrille Artho and Guillaume Rousset. Model-based testing of the Java network API. *Electronic Proceedings in Theoretical Computer Science*, 245:46–51, March 2017. ISSN 2075-2180. doi: 10.4204/eptcs.245.4.

[4] 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.

[5] 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.

[6] 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.

[7] 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: 10.1016/0169-7552(87)90085-7.

[8] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-Based Testing of Reactive Systems*. Springer Berlin Heidelberg, 2005. doi: 10.1007/b137241.

[9] 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.

[10] Lisa M. Dusseault. HTTP extensions for web distributed authoring and versioning (WebDAV). RFC 4918.

[11] 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.

[12] Roy T. Fielding and Julian Reschke. Hypertext transfer protocol (HTTP/1.1): Semantics and content. RFC 7231, June 2014.

[13] Roy T. Fielding and Julian Reschke. Hypertext transfer protocol (HTTP/1.1): Conditional requests. RFC 7232, June 2014.

[14] 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.

[15] George Fink and Matt Bishop. Property-based testing: A new approach to testing for assurance. *SIGSOFT Software Engineering Notes*, 22(4):74–80, July 1997. ISSN 0163-5948. doi: 10.1145/263244.263267.

[16] 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. Springer International Publishing, 2021. ISBN 978-3-030-72019-3.

[17] Marijn Haverbeke. DAV module does not respect if-unmodified-since, November 2012. URL https://trac.nginx.org/nginx/ticket/242.

[18] John Hughes. Experiences with QuickCheck: Testing the hard stuff and staying sane. In Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella, editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, pages 169–186. Springer International Publishing, 2016. ISBN 978-3-319-30936-1. doi: 10.1007/978-3-319-30936-1_9.

[19] John Hughes, Benjamin C. Pierce, Thomas Arts, and Ulf Norell. Mysteries of DropBox: Property-based testing of a distributed synchronization service. In *2016 IEEE International Conference on Software Testing, Verification and Validation*, ICST 2016, pages 135–145, Chicago, IL, USA, April 2016. doi: 10.1109/ICST.2016.37.

[20] IEEE Computer Society. IEEE standard for information technology—portable operating system interface (POSIX) base specifications, 2017.

[21] 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.

[22] 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.

[23] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages (POPL)*, 2, December 2017. doi: 10.1145/3158133.

[24] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 3, October 2019. doi: 10.1145/3360607.

[25] Yishuai Li, Benjamin C. Pierce, and Steve Zdancewic. Model-based testing of networked applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, pages 529–539, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384599. doi: 10.1145/3460319.3464798.

[26] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990. ISSN 0001-0782. doi: 10.1145/96267.96279.

[27] Netcraft. Web server survey, May 2022. URL https://news.netcraft.com/archives/category/web-server-survey/.

[28] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[29] 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.

[30] 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.

[31] Robert Swiecki. Honggfuzz. URL https://github.com/google/honggfuzz.

[32] Igor Sysoev. Nginx. URL http://nginx.org/.

[33] 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: 10.1016/S0169-7552(96)00017-7.

[34] Jan Tretmans and Piërre van de Laar. Model-based testing with TorXakis: The mysteries of Dropbox revisited. In *Proceedings of the 30th Central European Conference on Information and Intelligent Systems*, CECIIS, pages 247–258. Zagreb: Faculty of Organization and Informatics, University of Zagreb, October 2019.

[35] Jan Tretmans and Piërre van de Laar. TorXakis. URL https://github.com/TorXakis/Torxakis.

[36] Lionel Sujay Vailshery. Quality assurance and testing budget allocation 2012–2019, February 2022. URL https://www.statista.com/statistics/500641/worldwide-qa-budget-allocation-as-percent-it-spend/.

[37] W3Techs. Historical trends in the usage statistics of web servers. URL https://w3techs.com/technologies/history_overview/web_server.

[38] Li-yao Xia and Yishuai Li. Coq SimpleIO. URL https://github.com/Lysxia/coq-simple-io.

[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. *Proceedings of the ACM on Programming Languages (POPL)*, 4: 51:1–51:32, December 2019. ISSN 2475-1421. doi: 10.1145/3371119.

[40] Michal Zalewski. American fuzzy lop, 2017. URL https://lcamtuf.coredump.cx/afl/.

[41] Hengchu Zhang, Wolf Honoré, 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.