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 Commitee

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

Acknowledgments

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

ABSTRACT

TESTING BY DUALIZATION

Yishuai Li
Benjamin C. Pierce

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

Contents

List of Figures

CHAPTER 1

Introduction

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

This thesis studies the testing of interactive systems' semantics: The system under test (SUT) interacts with the tester by sending and receiving messages, and the tester determines whether the messages sent by the SUT are valid or not with respect to the protocol specification.

This chapter provides a brief view of interactive testing (Section 1.1), explains why nondeterminism makes this problem difficult (Sections 1.2–1.3), and discusses how language designs address the challenges caused by nondeterminism (Section 1.4).

1.1. Interactive Testing

Suppose we want to test a web server that supports GET and PUT methods:

```
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.
```

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

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

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

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

```
(* Compute the expected response and next state of the server. *)
Definition serverSpec request data :=
  match request with
  | GET k   ⇒ (data k, data)
  | PUT k v ⇒ (Done   , data [k ↦ v])
  end.
(* Validate the transaction against the stateful specification. *)
Definition validate spec request response data :=
  let (expect, next) := spec request data in
  if response =? expect then Success next else Failure.
(* Produce transactions for the validator. *)
CoFixpoint harness validator state :=
  request ← random;;
  send request;;
  response ← recv;;
  if validator request response state is Success next
  then harness validator next
  else reject.
Definition tester := harness (validate serverSpec).
```

Such testing method works for deterministic systems, whose behavior can be precisely computed from its input. Whereas, many systems are allowed to behave nondeterministically. How to test systems that involve randomness? How to validate servers' behavior against concurrent clients? 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. 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. For example, HTTP [5] allows requests to be conditional: If the client has a local copy of some resource and the copy on the server has not changed, then the server needn't resend the resource. To achieve this, an HTTP server may generate a short string, called an “entity tag” (ETag), identifying the content of some resource, and send it to the client:

`/* Client: */`	`/* Server: */`
`GET /target HTTP/1.1`	`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: */`	`/* Server: */`
`GET /target HTTP/1.1`	`HTTP/1.1 304 Not Modified`
`If-None-Match: "tag-foo"`	

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 `If-Match` precondition, which instructs the server to only update the content if its current ETag matches:

`/* Client: */`	`/* Server: */`
`PUT /target HTTP/1.1`	`HTTP/1.1 204 No Content`
`If-Match: "tag-foo"`	
`... content (A) ...`	

`/* Client: */`	`/* Server: */`
`GET /target HTTP/1.1`	`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: */`	`/* Server: */`
`PUT /target HTTP/1.1`	`HTTP/1.1 412 Precondition Failed`
`If-Match: "tag-baz"`	
`... content (B) ...`	

```
/* Client: */                 /* Server: */
GET /target HTTP/1.1          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 targetting 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 should not match with 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 its knowledge gained in previous interactions.

Even a simple nondeterminism like ETags requires such careful design of the validator, based on thorough comprehension of the specification. For more complex protocols, we hope to construct the validator in a reasonable way.

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

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

When testing servers, the environment is the network stack between the client and the server. The input is the request sent by the client, and the output is the response sent by the server. The response is transmitted via the network, until reaching the client side as observations.

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

To reveal the server's behavior upon concurrent requests, the tester needs to simulate multiple clients, sending new requests before receiving previous responses. The network delay might cause the server to receive requests in a different order from

```
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`.

that on the tester side. Vice versa, responses sent by the server might be reordered before arriving at the tester, as shown in Figure 1.3. Such tester's observation can be explained by various outputs on the SUT side. The validator needs to consider all possible outputs that can explain such observation, and see if anyone of them complies

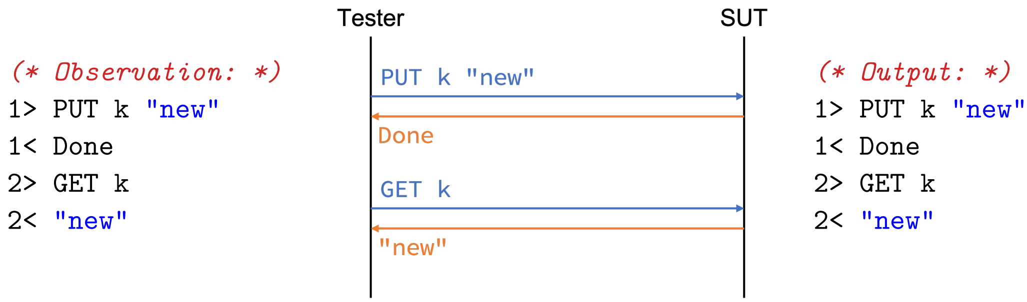


FIGURE 1.2. Upon no concurrency, the observation is identical to the output.

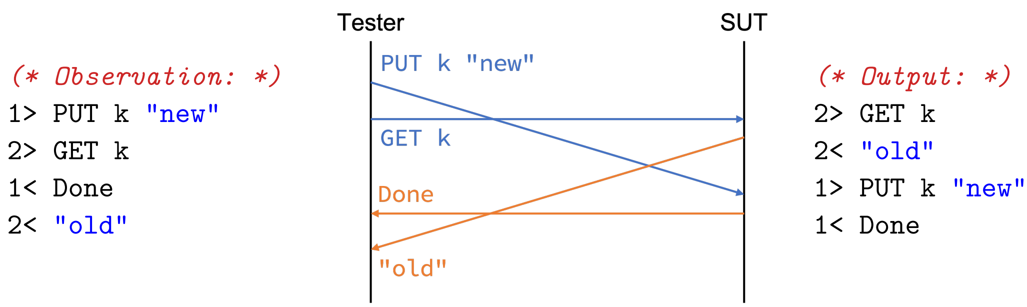


FIGURE 1.3. Acceptable: The observation can be explained by a valid output reordered by the network environment.

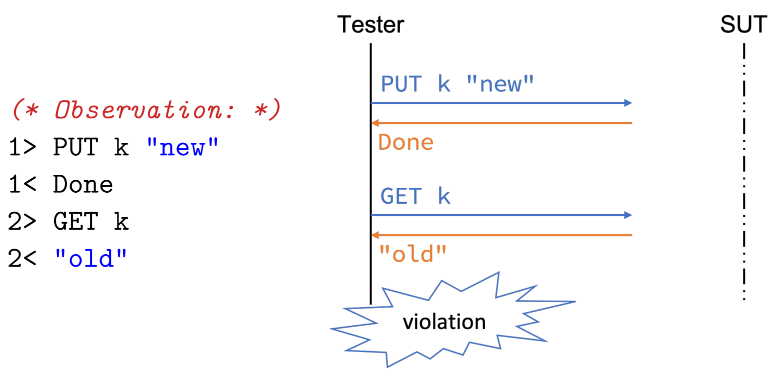


FIGURE 1.4. Unacceptable: The tester received the `Done` response before sending the `GET` request, thus the SUT must have processed the `PUT` request before the `GET` request. Therefore, the `"old"` response must be invalid.

with the specification. If no valid output can explain the observation, then the tester should reject the SUT, as shown in Figure 1.4.

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

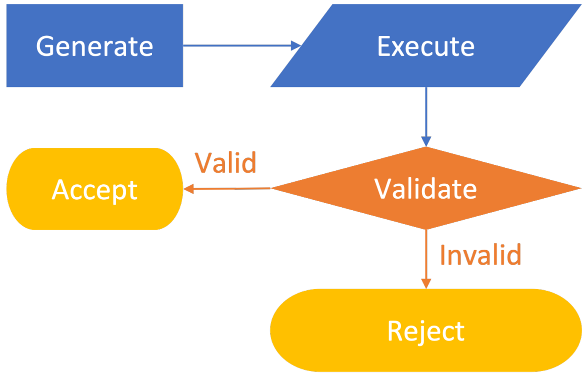


FIGURE 1.5. Simple tester architecture without shrinking.

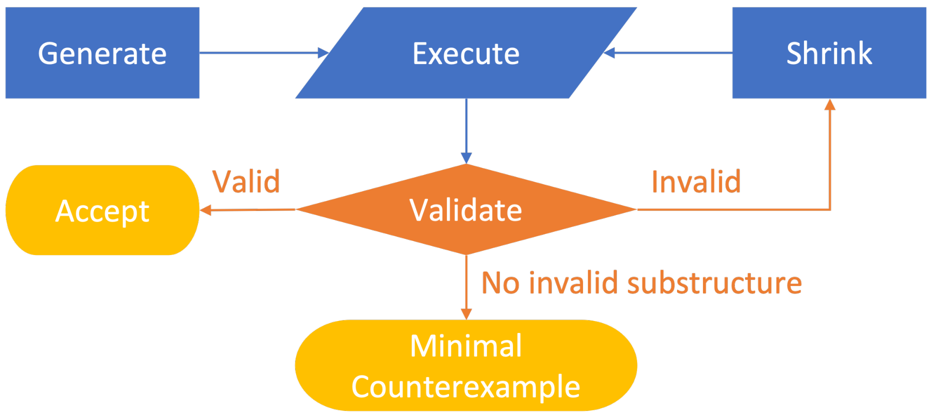


FIGURE 1.6. Tester architecture with shrinking mechanism.

1.3. Test harness and inter-execution nondeterminism

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

1.3.1. Test harness. Intuitively, a tester generates test input and executes the test. It then validates the observation and accepts/rejects the SUT, as shown in Figure 1.5.

However, to achieve better coverage, a randomized generator might produce huge test input. Suppose the tester has revealed invalid observation after thousands of interactions, such 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.

The test harness consists of generator, shrinker, and executor. This thesis studies the generator and the shrinker that produce the test input. The executor that produces observations based on the input is discussed in the related works chapter.

Interesting test inputs are those that are more likely to reveal invalid observations. Such subset is usually sparse and cannot be enumerated within reasonable budget *e.g.* in Subsection 1.2.1, request ETags that match the target resources'. The tester needs to manipulate the inputs' distribution, by implementing heuristics that emphasize certain input patterns. Such heuristics 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, 15 Feb 2022 08:31:23 GMT
                           /* Server: */
                           HTTP/1.1 200 OK
                           Last-Modified: Wed, 16 Feb 2022 08:31:23 GMT
                           ... content of target ...
/* Client: */
GET /index.html HTTP/1.1
If-Modified-Since: Wed, 16 Feb 2022 08:31:23 GMT
                           /* Server: */
                           HTTP/1.1 304 Not Modified
```

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

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

Such inter-execution 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 hope to implement a generic shrinking mechanism that can reproduce the heuristics in the test generator's design.

1.4. Contribution

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

(1) The specification is written as a reference implementation—a nondeterministic program that exhibits all possible behavior 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 of 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 protocol compliance problem into constraint solving.

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

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

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

(3) The language for test inputs is designed with inter-execution nondeterminism in mind. By instantiating the inputs' symbolic intermediate representation with different observations, the test harness gets different test inputs but preserves the pattern.

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

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

This claim is supported by the following publications:

(1) *From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server* [10], with Nicolas Koh, Yao Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, and William Mansky, where I developed a tester program based on a swap server's specification written as ITrees [17], and evaluated the tester's effectiveness by mutation testing.

(2) *Verifying an HTTP Key-Value Server with Interaction Trees and VST* [18], with Hengchu Zhang, Wolf Honoré, Nicolas Koh, Yao Li, Li-yao Xia, Lennart Beringer, and William Mansky, 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* [12], 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.
(4) *Testing by Dualization* (to be submitted to OOPSLA), a theory for interactive testing, explaining how to specify protocols using abstract model implementations, and how to guarantee the soundness and completeness of validators derived from the abstract model.

Outline. This thesis is structured as follows:

CHAPTER 2

Dualization Theory

This chapter provides a theoretic view for validators, and shows how to address internal nondeterminism by dualizing symbolic specifications.

Section 2.1 defines the basic concepts in testing. Section 2.2 introduces a simple language that exhibits internal nondeterminism. From specifications written in this language, Section 2.3 derives validators by dualization. The derived validators are proven correct in Section 2.4.

2.1. Concepts

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

DEFINITION 2.1 (Implementations and Traces). *Implementations* are programs that can interact with their environment. *Traces* are the outputs and inputs during execution.[1] "Implementation i can *produce* trace t" is written as "$i \xrightarrow{t}$".

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

DEFINITION 2.3 (Tester components and correctness). A tester consists of (i) a *validator* that accepts or rejects traces, and (ii) a *test harness* that triggers different traces with various input.

A tester is *correct* if its acceptances and rejections are sound and complete. A tester is *rejection-sound* if it only rejects incompliant implementations; it is *rejection-complete* if it can reject all incompliant implementations, provided sufficient time of execution.[2]

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.

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

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

[1]This chapter focuses on internal nondeterminism, and assumes no external nondeterminism. The tester's observation is considered identical to the SUT's output.

[2]The semantics of "soundness" and "completeness" vary among contexts. This thesis inherits terminologies from existing literature [15], but explicitly use "rejection-" prefix for clarity. "Rejection soundness" is equivalent to "acceptance completeness", and vice versa.

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

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

The rest of this chapter shows how to build validators that can be proven sound and complete.

2.2. QAC language family

To illustrate how to write specifications for testing purposes, this section introduces the "query-answer-choice" (QAC) language family for specifying network protocols that involve internal nondeterminism.

DEFINITION 2.5 (Deterministic server model). Network protocols can be specified with "reference implementations" *i.e.* model programs that exhibit the space of valid behavior. Networked servers can be modelled as infinite stateful programs that compute the answer for each query.

Let Q be the query type, A be the response type, and S be some server state type. Then a deterministic server is an infinite loop, defined by a loop body and an initial state. The loop body is a state monad that takes a query, produces the response based on its current state, and computes the next server state:

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

For example, consider an CMP-INC protocol: The server stores a number `S`. If the client sends zero, then the server increases `S` by one. Otherwise, the server responds with `1` if and only if the request is equal to the current value of `S`:

```
int S = 0;
while (true) {
  int request = recv();
  if (request == 0) { ++S; send(0); }
  else send(request == S ? 1 : 0);
}
```

Such server can be modelled as:

$$\mathsf{pack}\ S = \mathbb{N}\ \mathsf{with}\ (\lambda(q,s) \Rightarrow \begin{cases} (0, s+1) & q\ \mathsf{is}\ 0 \\ (\mathsf{if}\ q\ \mathsf{is}\ s\ \mathsf{then}\ 1\ \mathsf{else}\ 0, s) & \text{otherwise} \end{cases},$$
$$0)$$

DEFINITION 2.6 (Nondeterministic server model). In general, servers' responses and transitions might depend on choices that are invisible to the testers. These choices include inter-implementation nondetermism like algorithm design, and inter-execution nondeterminism like random numbers and timestamps. Let C be the space of invisible choices, then a nondeterministic server is specified as:

$$\begin{aligned}
&\mathsf{Server} \triangleq \{\exists S, (Q \times C \times S \to A \times S) \times S\} \\
&\mathsf{stepServer} : Q \times C \times \mathsf{Server} \to A \times \mathsf{Server} \\
&\mathsf{stepServer}(q, c, \mathsf{pack}\ S = \sigma\ \mathsf{with}\ (\mathsf{sstep}, \mathit{state})) \triangleq \\
&\qquad \mathsf{let}\ (a, \mathit{state}') = \mathsf{sstep}(q, c, \mathit{state})\ \mathsf{in} \\
&\qquad (a, \mathsf{pack}\ S = \sigma\ \mathsf{with}\ (\mathsf{sstep}, \mathit{state}'))
\end{aligned}$$

Here $\mathsf{pack}\ S = \sigma\ \mathsf{with}\ (\mathsf{sstep}, \mathit{state})$ is an instance of the Server existential type [13], where sstep is of type $Q \times C \times \sigma \to A \times \sigma$, and state has type σ.

Consider changing the aforementioned CMP-INC into CMP-RST: Upon request zero, the server resets `S` to a random number:

```
int S = 0;
while (true) {
  int request = recv();
  if (request == 0) { S = rand(); send(0); }
  else send(request == S ? 1 : 0);
}
```

Its corresponding server model can be written as

$$\mathsf{pack}\ S = \mathbb{N}\ \mathsf{with}\ (\lambda(q, c, s) \Rightarrow \begin{cases} (0, c) & q\ \mathsf{is}\ 0 \\ (\mathsf{if}\ q\ \mathsf{is}\ s\ \mathsf{then}\ 1\ \mathsf{else}\ 0, s) & \text{otherwise} \end{cases},$$
$$0)$$

2.3. Dualizing specifications into validators

2.3.1. Encoding servers and validators. The QAC language family allows specifying a wide variaty of protocols with state monads. To derive a validator from a server model, we need to analyze the computations of the model program, which is represented in some programming language.

Definition 2.7 (Server and validator of a program). A program $p \in \mathsf{Prog}$ is a representation of computation that can be "instantiated" into a server model:

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

A program can also be "interpreted" into other computations, including validators:

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

For example, protocols like CMP-RST can be encoded with a simple Prog language:

$$\begin{array}{lll}
\mathsf{Prog} \triangleq & \mathsf{return} & \text{respond with value at address } !0 \\
& \mid\ !\mathit{dst} \coloneqq \mathsf{SExp}; \mathsf{Prog} & \text{write to memory} \\
& \mid\ \mathsf{if}\ \mathsf{SExp} \le \mathsf{SExp}\ \mathsf{then}\ \mathsf{Prog}\ \mathsf{else}\ \mathsf{Prog} & \text{conditional branch} \\
\mathsf{SExp} \triangleq & \mathbb{N} & \text{constant number} \\
& \mid\ !\mathit{src} & \text{read from memory} \\
& \mid\ \mathsf{SExp}\ \mathit{op}\ \mathsf{SExp} & \mathit{op} \in \{+, -, \times, \div\}
\end{array}$$

This language assumes Q, A, C to be natural numbers, and the server state to be a key-value store of natural numbers. The initial server state maps all keys to zero:

$$\mathsf{serverOf}(p) \triangleq \mathsf{pack}\ S = \mathbb{N} \to \mathbb{N}\ \mathsf{with}\ (\mathsf{sstep}_p, (\_ \mapsto 0))$$

The server's loop body sstep_p is defined as follows: First write the internal choice to address !1, and write the query to address !0. Then execute the program p until it returns. Finally, send back the value stored in address !0 as response:

$$\begin{aligned}
\mathsf{sstep}_p(q, c, s_0) &\triangleq \mathsf{let}\ s_1 = s_0[1 \mapsto c]\ \mathsf{in} \\
&\quad\ \mathsf{let}\ s_2 = s_1[0 \mapsto q]\ \mathsf{in} \\
&\quad\ \mathsf{let}\ s_3 = \mathsf{exec}(p, s_2)\ \mathsf{in} \\
&\quad\ (s_3!0, s_3) \\
\mathsf{exec}(p, s) &\triangleq \begin{cases} s & p\ \mathsf{is}\ \mathsf{return} \\ \mathsf{exec}(p', s[dst \mapsto e^s]) & p\ \mathsf{is}\ !dst := e; p' \\ \mathsf{exec}(\mathsf{if}\ {e_1}^s \le {e_2}^s\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2, s) & p\ \mathsf{is}\ \mathsf{if}\ e_1 \le e_2\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 \end{cases} \\
e^s &\triangleq \begin{cases} n & e\ \mathsf{is}\ n : \mathbb{N} \\ s!src & e\ \mathsf{is}\ !src \\ {e_1}^s\ op\ {e_2}^s & e\ \mathsf{is}\ e_1\ op\ e_2 \end{cases}
\end{aligned}$$

Accordingly, the model program p should read the query from address !0, and parameterize the space of nondeterministic behavior over the internal choice in address !1. When the program returns, it should have stored the computed response in address !0. Addresses greater than !1 are only writable by the specification, and can be used for storing the server state. For example, the Prog representation of the CMP-RST server is:

$$\begin{aligned}
&\mathsf{if}\ !0 \equiv 0\ \mathsf{then}\ !2 :=!1; \mathsf{return} && (1) \\
&\mathsf{else\ if}\ !0 \equiv !2\ \mathsf{then}\ !0 := 1; \mathsf{return} && (2) \\
&\mathsf{else}\ !0 := 0; \mathsf{return} && (3)
\end{aligned}$$

Here "$\mathsf{if}\ e_1 \equiv e_2\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2$" is a syntactic sugar for:

$$\mathsf{if}\ e_1 \le e_2\ \mathsf{then}\ (\mathsf{if}\ e_2 \le e_1\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2)\ \mathsf{else}\ p_2$$

When the query is zero (case 1), the nondeterministic value is stored in address !2, and compared against non-zero queries. Address !0 is untouched, so the server echoes the request back as response.

For non-zero queries (cases 2 and 3), the server checks whether the request is equal to the value stored in address !2, and stores the corresponding response to address !0.

2.3.2. Dualize model program into validator. The validator for this Prog language needs to determine whether the trace is produceable by the server model. More specifically, whether the responses in the trace can be *explained* by their corresponding symbolic expressions in the server model.

To achieve this goal, the validator maintains "validation states" that consist of: (1) a mapping from server model's addresses to *symbolic variables* that represent the value stored therein, and (2) a set of *constraints* over the symbolic variables. The key idea is to compute the symbolic representation of the server's outputs, and determine whether the actual responses observed in the trace can be explained by (*i.e.* unified with) these symbolic expressions:

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

Here the initial validation state says "the server server state has all addresses storing the value represented by variable #0", and has a single constraint that "variable #0's value is equal to zero".

The validator's loop body is derived as follows:

(1) When the server performs a write operation $!dst \coloneqq exp$, 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.
(2) When the server makes a nondeterministic branch $\mathsf{if}\ e_1 \leq e_2\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2$, consider both cases: (a) If p_1 was taken, then the validator should add a constraint $e_1 < e_2$ or $e_1 \equiv e_2$; or (b) If p_2 was taken, then the validator should add a constraint $e_1 > e_2$.
(3) Before executing the program, the server writes the nonderministic value c to address !1. Accordingly, the validator creates a fresh variable to represent the new value stored in address !1, without adding any constraint.
(4) After executing the program, the server sends back the value stored in !0 as response. Accordingly, the validator adds a constraint that says the variable representing address !0 is equal to the observed response.

A validation state is rejecting if its constraints are not satisfiable, *i.e.* the observed response cannot unify with its symbolic representation defined in the server model.

Notice that derivation rule (2) needs to take both branches into account. This requires the validator to maintain a set of validation states , one for each possible execution path of the server model. The validator should reject the trace if all of its validation states are rejecting, indicating that no branch in the server model can explain the trace:

$$
\begin{array}{ll}
\mathsf{vstep}_p(q,a,v) & \triangleq \mathsf{let}\ v' = \mathsf{vstep}'_p(q,a,v)\ \mathsf{in} \\
& \quad \mathsf{if}\ v'\ \mathsf{is}\ \varnothing\ \mathsf{then\ None\ else\ Some}\ v' \\
\mathsf{vstep}'_p(q,a,v) & \triangleq (vs_0, cs_0) \leftarrow v; \\
& \quad \mathsf{let}\ x_c = \mathsf{fresh}\ vs_0\ \mathsf{in} \\
& \quad \mathsf{let}\ vs_1 = vs_0[1 \mapsto x_c]\ \mathsf{in} \\
& \quad \mathsf{let}\ x_q = \mathsf{fresh}\ vs_1\ \mathsf{in} \\
& \quad \mathsf{let}\ v_2 = (vs_1[0 \mapsto x_q], cs_0 \cup \{\#x_q \equiv q\})\ \mathsf{in} \\
& \quad (vs_3, cs_3) \leftarrow \mathsf{eval}(p, v_2); \\
& \quad \mathsf{let}\ cs_4 = cs_3 \cup \{\#(vs_3!0) \equiv a\}\ \mathsf{in} \\
& \quad \mathsf{if\ solvable}\ cs_4\ \mathsf{then}\ \{(vs_4, cs_4)\}\ \mathsf{else}\ \varnothing \\
\mathsf{eval}(p,(vs,cs)) & \triangleq \begin{cases}
\{(vs,cs)\} & p\ \mathsf{is\ return} \\
\begin{pmatrix}\mathsf{let}\ x_e = \mathsf{fresh}\ vs\ \mathsf{in} \\ \mathsf{let}\ v' = (vs[d \mapsto x_e], cs \cup \{\#x_e \equiv e^{vs}\})\ \mathsf{in} \\ \mathsf{eval}(p', v')\end{pmatrix} & p\ \mathsf{is}\ !d \coloneqq e; p' \\
\begin{pmatrix}\mathsf{let}\ v_1 = (vs, cs \cup \{{e_1}^{vs} \le {e_2}^{vs}\})\ \mathsf{in} \\ \mathsf{let}\ v_2 = (vs, cs \cup \{{e_2}^{vs} < {e_1}^{vs}\})\ \mathsf{in} \\ \mathsf{eval}(p_1, v_1) \cup \mathsf{eval}(p_2, v_2)\end{pmatrix} & p\ \mathsf{is\ if}\ e_1 \le e_2\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2
\end{cases} \\
\mathsf{constraint} & \triangleq \mathsf{VExp}\ cmp\ \mathsf{VExp} \qquad \text{where } cmp \in \{<, \le, \equiv\} \\
\mathsf{VExp} & \triangleq \mathbb{N} \mid \#x \mid \mathsf{VExp}\ op\ \mathsf{VExp} \qquad \text{where } op \in \{+, -, \times, \div\} \\
e^{vs} & \triangleq \begin{cases}
n & e\ \mathsf{is}\ n : \mathbb{N} \\
\#(vs!src) & e\ \mathsf{is}\ !src \\
{e_1}^{vs}\ op\ {e_2}^{vs} & e\ \mathsf{is}\ e_1\ op\ e_2
\end{cases}
\end{array}
$$

Here the notation $x \leftarrow v; f(x)$ is a monadic bind for sets: Let f map each element (vs, cs) in v to a set of validation states ($f(vs, cs) : \mathsf{set}\ ((\mathbb{N} \to \mathbb{N}) \times \mathsf{set\ constraint})$). The return value of vstep'_p is the union of all result sets.

This validator assumes a constraint solver that can determine whether a set of constraints is satisfiable, *i.e.* there exists an assignment of all variables that satisfy all the constraints:

$$\forall cs, \mathsf{solvable}\ cs \iff \exists (asgn : \mathbb{N} \to \mathbb{N}), asgn\ \mathsf{satisfy}\ cs$$

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

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

For example, the initial validation state can be satisfied by any assignment whose value at $\#0$ is equal to zero. The vstep'_p function adds constraints to each validation state, until its constraints become unsolvable, and gets dropped from the result.

If the constraints in all validation states are unsolvable, then vstep'_p returns an empty result set $\varnothing$, which leads to rejection of the trace (vstep_p returning None).

2.4. Soundness and completeness of derived validators

Validators derived in this approach can be proven sound and complete:

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

In this subsection, I first present a generic framework for proving validators' correctness properties, and then demonstrate its usage by applying it to Prog-based validators.

The "equivalence between server production and validator consumption of all traces" is proven by introducing a loop invariant. The invariant is a bisimulation between the server and the validator, which is preserved for each step of the trace produced/consumed.

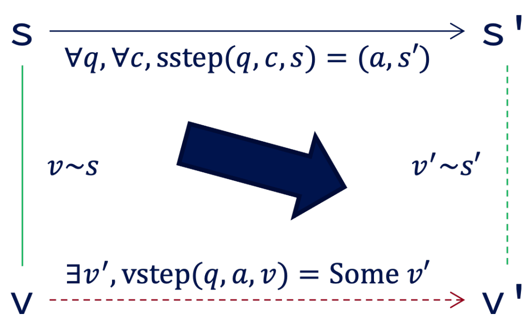
2.4.1. Proving rejection soundness. The proof of "any trace produceable by the server is consumable by the validator" is by forward induction on the server's execution path. The corresponding validation path is constructed based on the following hypotheses:

(1) The initial server state reflects the initial validator state:

$$(v_0 : V) \sim (s_0 : S) \tag{RejSound1}$$

(2) Any server step whose pre-execution state reflects some pre-validation state can be consumed by the validator into a post-validation state that reflects the post-execution state:

$$\begin{aligned}&\forall (q : Q)(c : C)(a : A)(s, s' : S)(v : V),\\ &\mathsf{sstep}(q, c, s) = (a, s') \wedge v \sim s\\ &\implies \exists v' : V, \mathsf{vstep}(q, a, v) = \mathsf{Some}\ v' \wedge v' \sim s'\end{aligned} \tag{RejSound2}$$



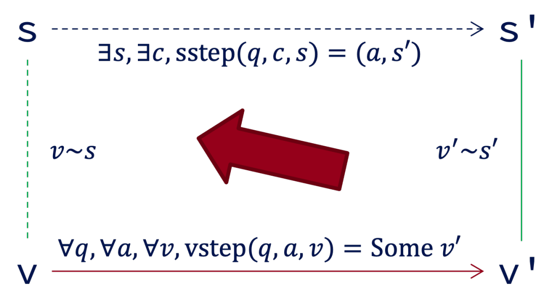
2.4.2. Proving rejection completeness. The proof of "any trace consumable by the validator is produceable by the server" is by backward induction on the validator's execution path. The corresponding server path is constructed based on the following hypotheses:

(1) Any accepting validator step has some server state that reflects the post-validation state:

$$\forall (q : Q)(a : A)(v, v' : V), \mathsf{vstep}(q, a, v) = \mathsf{Some}\ v' \implies \exists s' : S, v' \sim s' \tag{RejComplete1}$$

(2) Any accepting validator step whose post-validation state reflects some post-execution server state has a corresponding server step from a pre-execution state that reflects the pre-validation state:

$$\begin{aligned} &\forall (q : Q)(a : A)(v, v' : V)(s' : S), \\ &\mathsf{vstep}(q, a, v) = \mathsf{Some}\ v' \land v' \sim s' \\ &\implies \exists (s : S)(c : C), \mathsf{sstep}(q, c, s) = (a, s') \land v \sim s \end{aligned} \tag{RejComplete2}$$



(3) The initial validator state only reflects the initial server state:

$$\{s \mid v_0 \sim s\} = \{s_0\} \tag{RejComplete3}$$

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, while unknown from the validator step: Given a validator step, we cannot predict "what choices the server will make in the future", but can analyze "what choices the server might have made in the past". This proof method is further explained with the Prog example:

For specifications written in the Prog language, the server state is a mapping from addresses to data; the validation state is a mapping from addresses to symbolic variables, and constraints over the variables. A validation state is accepting if its constraints are satisfiable, *i.e.* there exists an *assignment* of the symbolic variables that can unify the trace with the server model. The internal choices made by the server are represented as symbolic variables. The assignment maps these variables to the choices' value at each step, from which we can reconstruct the server's execution path.

Definition 2.8 (Bisimulation for Prog specifications). Validator state v *simulates* server state s if it contains a validation state (vs, cs) that *reflects* the server state:

(1) There exists an assignment $asgn$ that can satisfy the constraints cs; and (2) The address-variable mapping vs can be *instantiated* with the assignment (written as "vs^{asgn}") into an address-data mapping that is equivalent with s:

$$\begin{aligned} v \sim s \quad &\triangleq \quad \exists((vs, cs) \in v)(asgn : \mathbb{N} \to \mathbb{N}), asgn \ \mathsf{satisfy} \ cs \land vs^{asgn} = s \\ vs^{asgn} \quad &\triangleq \quad addr \mapsto asgn!(vs!addr) \end{aligned}$$

This bisimulation definition satisfies the hypotheses for proving soundness and completeness:

Hypotheses RejSound1 and RejComplete3 are immediate from the initial states' definition: The initial server state is all-zero map. The initial validator state is a singleton that maps all addresses to a variable that is constrained to have value zero.

Hypothesis RejComplete1 is based on the fact that vstep_p checks the nonemptiness of the result:

$$\forall q \ a \ v \ v', \mathsf{vstep}_p(q, a, v) = \mathsf{Some} \ v' \implies (\mathsf{vstep}'_p(q, a, v) = v' \land \exists(vs, cs) \in v')$$

and that vstep'_p guards the satisfiability of all constraints in its result:

$$\forall q \ a \ vs \ cs, (vs, cs) \in \mathsf{vstep}'_p(q, a, v) \implies \exists asgn, asgn \ \mathsf{satisfy} \ cs$$

Therefore, any element in the resulting validator state can construct a simulating server state:

$$\forall asgn \ cs \ vs \ v, (asgn \ \mathsf{satisfy} \ cs \land (vs, cs) \in v) \implies v \sim vs^{asgn}$$

Hypothesis RejSound2 is based on the fact that the pre-validation state must contain an element that reflects the server's pre-execution state, as defined by the bisimulation relation. Given the server's internal choices, we can compute its execution path. By induction on the server's execution path, we can construct the corresponding post-validation state by making the same internal choice and branch decisions as the server did, and construct the assignment that satisfies the validator's constraints.

Hypothesis RejComplete3 observes that validator design increases the constraints monotonically. Therefore, "assignments that can satisfy the post-validation constraints" is a subset of "assignments that can satisfy the pre-validation constraints":

$$\forall q \, a \, vs \, cs \, vs' \, cs' \, asgn, ((vs', cs') \in \mathsf{vstep}'_p(q, a, (vs, cs)) \land asgn \ \mathsf{satisfy} \ cs') \implies asgn \ \mathsf{satisfy} \ cs$$

As a result, the corresponding pre-step server state and the internal choice can be constructed, and proven to perform the server-side step:

$$\begin{aligned} &\forall q \ a \ vs \ cs \ vs' \ cs' \ asgn, \ (vs', cs') \in \mathsf{vstep}'_p(q, a, (vs, cs)) \\ &\qquad \implies \mathsf{sstep}_p(q, asgn!(\mathsf{fresh} \ vs), vs^{asgn}) = vs'^{\ asgn} \end{aligned}$$

The intuition here is that the assignment includes "all choices made by the server, past and future", which is narrowed upon more and more observations. Therefore, the assignment can instantiate all previous validator states into corresponding servers, and reconstruct the server's execution path by inferring its internal choices.

CHAPTER 3

Testing in Practice

3.1. Specification Languages

3.1.1. Property-based specification with QuickChick. My first formal specification of HTTP/1.1 was written as QuickChick [11] properties, which takes a trace of requests, and determines whether the traces is valid per protocol specification, like that shown in Figure 1.1. The specification implemented a constraint solving logic by hand, making it hard to scale when the protocol becomes more complex, as discussed in **??**

3.1.2. Model-based specification with ITrees. To write specifications for protocols' rich semantics, I employed "interaction tree" (ITree), a generic data structure for representing interactive programs, introduced by Xia et al. [17]. ITree enables specifying protocols as monadic programs that model valid implementations' possible behavior. The model program can be interpreted into a tester program, to be discussed in Section 3.2.

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

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

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

Definition trace E := list (event E)

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

FIGURE 3.1. Interaction trees and their traces of events.

There are three ways to construct an ITree. The `Ret r` constructor corresponds to the trivial computation that halts and yields the value `r`. The `Tau t` constructor corresponds to a silent step of computation, which does something internal that does not produce any visible effect and then continues as `t`. Representing silent steps explicitly with `Tau` allows us, for example, to represent diverging computation without violating Coq's guardedness condition [3]:

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

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

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

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

CoInductive echo : itree IO () :=
  Vis Input (λ x ⇒ Vis (Output x) (λ _ ⇒ echo)).
```

3.2. From Specification to Tester

From an ITree specification, I conducted "offline" testing, which takes a trace and determines its validity [10], and "online" testing, where the specification is derived into a client program that validates the system under test interactively [12].

3.2.1. Offline testing of swap server. I started with testing a simple "swap server" [10], specified in Figure 3.2. The specification says that the server can either accept a connection with a new client (`obs_connect`) or else receive a message from a client over some established connection (`obs_msg_to_server c`), send back the current stored message (`obs_msg_from_server c last_msg`), and then start over with the last received message as the current state.

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

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

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

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

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

3.2.2. Online testing of HTTP. To test protocols with internal nondeterminism (*e.g.* HTTP) effectively, I introduced a symbolic representation for the server's invisible choices, as shown in Figure A.2. I then defined a TCP network model in Figure A.1. Combining the server and network models produces a model program that exhibits all valid observations, considering both internal and network nondeterminism.

From the server and network models, I derived a tester client that interacts with servers over the network, and validates the observations against the protocol specification, as shown in Figure A.3.

Using this automatially derived tester program, I have found violations against HTTP/1.1 in the latest version of both Apache and Nginx. More details are explained in Li, Pierce, and Zdancewic [12].

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

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

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

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

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

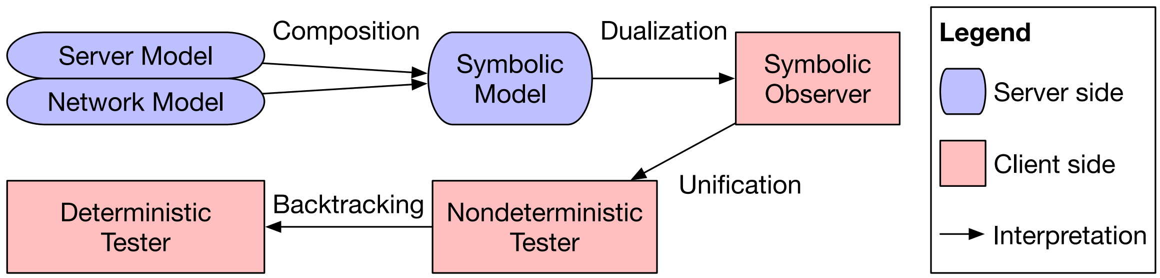


FIGURE 3.5. Deriving tester program from specification

CHAPTER 4

Test Harness Design

CHAPTER 5

Related Work

5.1. Specifying and Testing Protocols

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

The area of model-based testing is well-studied, diverse, and difficult to navigate [1]. Here we focus on techniques that have been practiced in testing real-world programs, which includes notations (1) and (2). Notation (3) is infeasible for protocols with nontrivial nondeterminism, because the specification needs to define observer-side knowledge of the SUT's all possible internal states, making it complex to implement and hard to reason about, as shown in Figure 1.1.

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

Using a formal language strongly insired by LOTOS, Tretmans and Laar [16] implemented a test generation tool for symbolic transition systems called TorXakis, which has been used for testing Dropbox [16].

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

Bishop et al. [2] have developed rigorous specifications for transport-layer protocols TCP, UDP, and the Sockets API, and validated the specifications against mainstream implementations in FreeBSD, Linux, and WinXP. Their specification represents internal nondeterminism as symbolic states of the model, which is then evaluated using a special-purpose symbolic model checker. They focused on developing a post-hoc specification that matches existing systems, and wrote a separate tool for generating test cases.

5.2. Reasoning about Network Delays

For property-based testing against distributed applications like Dropbox, Hughes et al. [9] have introduced "conjectured events" to represent uploading and downloading events that nodes may perform at any time invisibly.

Sun, Xu, and Elbaum [14] symbolised the time elapsed to transmit packets from one end to another, and developed a symbolic-execution-based tester that found transmission-related bugs in Linux TFTP upon certain network delays. Their tester used a fixed trace of packets to interact with the server, and the generated test cases were the packets' delay time.

CHAPTER 6

Discussions

CHAPTER 7

Conclusion

Bibliography

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

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

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

[4] Roy T. Fielding and Gail Kaiser. "The Apache HTTP Server Project". In: *IEEE Internet Computing* 1.4 (July 1997), pp. 88–90. ISSN: 1941-0131. DOI: 10.1109/4236.612229.

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

[6] Roy T. Fielding and Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. June 2014. DOI: 10.17487/RFC7231. URL: https://rfc-editor.org/rfc/rfc7231.txt.

[7] Roy T. Fielding et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. DOI: 10.17487/RFC2616. URL: https://rfc-editor.org/rfc/rfc2616.txt.

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

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

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

[11] 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`.

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

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

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

[15] Jan Tretmans. "Conformance testing with labelled transition systems: Implementation relations and test generation". In: *Computer Networks and ISDN Systems* 29.1 (1996). Protocol Testing, pp. 49–79. ISSN: 0169-7552. DOI: `https://doi.org/10.1016/S0169-7552(96)00017-7`. URL: `http://www.sciencedirect.com/science/article/pii/S0169755296000177`.

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

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

[18] Hengchu Zhang et al. "Verifying an HTTP Key-Value Server with Interaction Trees and VST". In: *12th International Conference on Interactive Theorem Proving (ITP 2021)*. Ed. by Liron Cohen and Cezary Kaliszyk. Vol. 193. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 32:1–32:19. ISBN: 978-3-95977-188-7. DOI: `10.4230/LIPIcs.ITP.2021.32`. URL: `https://drops.dagstuhl.de/opus/volltexte/2021/13927`.

APPENDIX A

Unstructured contents

A.1. Challenges: Testing Internal and Network Nondeterminism

To illustrate the challenges in testing networked applications, we discuss two features of HTTP/1.1—conditional requests [5] and message forwarding [6]—showcasing internal nondeterminism and network nondeterminism, respectively.

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

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

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

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

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

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

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

[LY: This is a good example, but how general is the problem, since one might question the popularity of ETags? On the other hand, if your testing framework targets application layer protocols rather than just HTTP, maybe there are more similar examples? For example, file/mail servers or databases might also require some synchronization mechanisms similar to compare-and-swap? And there might be other examples that's not compare-and-swap?][BCP: Agree that this is important to discuss.] [LYS: Mentioned at the end of this section.]

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

It is possible, but tricky, to write an ad hoc tester for HTTP/1.1 by manually "dualizing" the behaviors described by the informal specification documents (RFCs). The protocol document describes *how* a valid server should handle requests, while the tester needs to determine *what* responses received from the server are valid. For example, "If the server has revealed some resource's ETag as `"foo"`, then it must not reject requests targetting this resource conditioned over `If-Match: "foo"`, until the resource has been modified"; and "Had the server previously rejected an `If-Match` request, it must reject the same request until its target has been modified." Figure 1.1 shows a hand-written tester for checking this bit of ETag functionality; we hope the reader will agree that this testing logic is not straightforward to derive from the informal "server's eye" specifications.

Network Nondeterminism. When testing an HTTP/1.1 server over the network, although TCP preserves message ordering within each connection, it does not guarantee any order between different connections. Consider a proxy model in **??**: it specifies how a server should forward messages. [BCP: I don't understand why we are talking about proxies here: a simple "server + several clients" situation is enough to create network nondeterminism. (I would expect that proxying might create *additional* possibilities for nondeterminism, of course.) [LYS: We need to talk about proxy somewhere, and I didn't find a good place elsewhere.]] [BCP: Moreover, the more I look at figures 2–5 the more confusing I find them. Only figure 5 mentions connections, but — for example, in figure 3, if we assume just a single connection between the observer and the proxy and a single connection from the proxy back to the observer, then the reordering shown in the figure is NOT valid. [LYS: Updated figure. No proxy uses the same connection for multiple requests. The proxy never knows if there's a next request that can use the same connection.]] When the forwarded messages are scrambled as in **??**, the tester should be *loose* enough to accept the server, because a valid server may exhibit such reordering due to network delays. The tester should also be *strict* enough to reject a server that behaves as **??**, because no network delay can let the proxy forward a message before the observer sends it.

The kinds of nondeterminism exemplified here can be found in many other scenarios: (i) Servers may use some (unknown) algorithm to generate internal state for nonces, sequence numbers, caching metadata, *etc*, featuring internal nondeterminism. (ii) When the server runs multiple threads concurrently (*e.g.* to serve multiple clients), the operating system might schedule these threads nondeterministically. When testing the server over the network, such "nondeterminism outside the code of the server program but still within the machine on which the server is executing" is indistinguishable from nondeterminism caused by network delays, and thus can be covered by the concept "network nondeterminism."

A.2. Specification Language

A specification in our framework consists of two parts: a server model specifying server-side behavior,[BCP: there was a discussion of this somewhere else: isn't our "application model" here just specifying HTTP and WebDAV? And so isn't it also generic? [LYS: Not generic over all L7 protocols.]] and a network model describing network delays. By composing these two models, we get a tester-side specification of valid observations over the network.

Formally, our specifications are written as *interaction trees*, a generic data structure for representing interactive programs in Coq. This language allows us to write rigorous mathematical specifications, and transform the specification into tester conveniently. In this paper, we present models as pseudocode for readability. Technical details about interaction trees can be found in [17].

Subsection A.2.1 shows how to handle network nondeterminism. Subsection A.2.2 then expands the model to address internal nondeterminism.

A.2.1. Server and Network Models. The *server model* specifies how the server code interacts with the network interface. For example, an extremely simplistic model of an HTTP proxy[BCP: again, it feels like proxies are coming out of nowhere [LYS: I'll try to make proxy more like a part of HTTP than an extension.]] (shown in **??**) is written as:

```
let proxy() =
    msg := recv();
    send(msg);
    proxy()
```

An implementation is said to be *valid* if it is indistinguishable from the model when viewed from across the network. Consider the following proxy implementation that reorders messages: [BCP: Why are we suddenly switching to C syntax?? [LYS: To distinguish implementation from specification.]]

```
void proxy_implementation() {
  while (true) {
    recv(&msg1); recv(&msg2);
    send(msg2);  send(msg1);
  }
}
```

This reordered implementation is valid, because the model itself may exhibit the same behavior when observed over the network, as shown in **??**. This "implementation's behavior is explainable by the model, considering network delays" relation is called *network refinement* by Koh et al. [10].

To specify network refinement in a testable way, we introduce the *network model*, a conceptual implementation of the transport-layer environment between the server and the tester. It models the network as a nondeterministic machine that absorbs packets and, after some time, emits them again. Figure A.1 shows the network model for concurrent TCP connections: The network either receives a packet from some node, or sends the first packet en route of some connection. This model preserves the

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

FIGURE A.1. Network model for concurrent TCP connections. The model maintains a `buffer` of all packets en route. In each cycle, the model may nondeterministically branch to either absorb `or` emit a packet. Any absorbed packet is appended to the end of buffer. When emitting a packet, the model may choose a connection and send the oldest packet in it.

message order within each connection, but it exhibits all possible reorderings among different connections.

The network model does not distinguish between server and tester. When one end `send`s some message, the network `recv`s the message and `send`s it after some cycles of delay; it is then observed by the other end via some `recv` call.

In Subsection A.3.3, we compose the server and network models to yield an observer-side specification for testing purposes.

A.2.2. Symbolic Representation of Nondeterministic Data. To incorporate symbolic evaluation in our testing framework, our specification needs to represent internally generated data as symbols. Consider HTTP PUT requests with `If-Match` preconditions: Upon success, the server generates a new ETag for the updated content, and the tester does not know the ETag's value immediately. Our symbolic model in Figure A.2 represents the server's generated ETags as fresh variables. The server's future behavior might depend on whether a request's ETag matches the generated (symbolic) ETag. Such matching produces a symbolic boolean expression, which cannot be evaluated into a boolean value without enough constraints on its variables. Our model introduces `IF` operator to condition branches over a symbolic boolean expression. Which branch the server actually took is decided by the derived tester in Section A.3.

In Subsection A.3.2, we implement the symbolic evaluation process that checks servers' observable behavior against this symbolic model.

A.3. Derivation: from Server Specification to Testing Program

From the specified the application and network models, our framework automatically derives a tester program that interacts with the server and determines its validity. The derivation framework is shown in outline in Figure A.3. Each box is an interaction

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

FIGURE A.2. Symbolic model handling conditional PUT request. The model maintains two states: `data` that maps keys to their values, and `xtag` that maps keys to symbolic variables that represent their corresponding ETags. Upon receiving a PUT request conditioned over "If-Match: `t`", the server should decide whether the request ETag `matches` that stored in the server. Upon matching, the server processes the PUT request, and represents the updated value's ETag as a fresh variable.

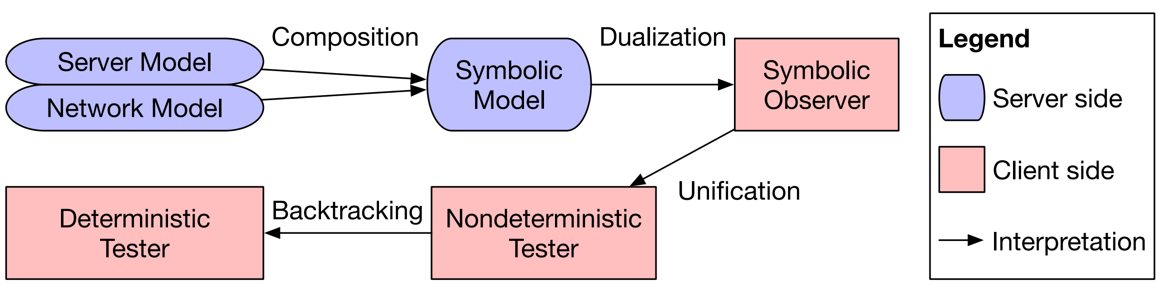


FIGURE A.3. Deriving tester program from specification

tree program, and the arrows are "interpretors" that transform one interaction tree into another. Subsection A.3.1 explains the concept of interpretation, and the rest of this section describes how to interpret the specification into a tester program.

A.3.1. Interpreting Interaction Trees. Interaction tree programs can be destructed into an interaction event followed by another interaction tree program. Such structure allows us to *interpret* one program into another. Figure A.4 shows an example of interpretation: The original `acc` program sends and receives messages, and the `tee` interpretor transforms the `acc` into another program that also prints the messages sent and received.

```
1  let acc(sum) =
2    x := recv(); send(x+sum); acc(x+sum) in
3  let tee(m) =
4    match m with
5    | x := recv(); m'(x) ⇒
6      a := recv(); print("IN" ++ a); tee(m'(a))
7    | send(a); m' ⇒
8      print("OUT" ++ a); send(a); tee(m')
9    end in
10 tee(acc(0))
11 (* ... is equivalent to ... *)
12 let tee_acc(sum) =
13   a := recv(); print("IN" ++ a);
14   print("OUT" ++ (a+sum)); send(a+sum);
15   tee_acc(a+sum) in
16 tee_acc(0)
17
```

FIGURE A.4. Interpretation example. `acc` receives a number and returns the sum of numbers received so far. `tee` prints all the numbers sent and received. Interpreting `acc` with interpretor `tee` results in a program that's equivalent to `tee_acc`.

Such interpretation is done by pattern matching on the program's structure in Line 4. Based on what the original program wants to do next, the interpretor defines what the result program should do in Line 6 and Line 8. These programs defined in accordance to events are called *handlers*. By writing different handlers for the events, interpretors can construct new programs in various ways, as shown in following subsections. Further details about interpreting interaction trees are explained by Xia et al. [17].

A.3.2. From Server Specification to Tester Program. For simplicity, we first explain how to handle servers' internal nondeterminism with symbolic evaluation. This subsection covers a subgraph of Figure A.3, starting with dualizing the symbolic model. Here we use the server model itself as the symbolic model, assuming no reorderings by network delays. We will compose the server model with the network model in Subsection A.3.3, addressing network nondeterminism.

Dualization. To *observe* the server's behavior, we have to interpret the specified server-side events into tester-side events: When the server should send a certain message, the tester expects to receive the specified message, and rejects the server upon receiving an unexpected message; when the server should receive some message, the tester generates a message and sends it to the server, as shown in Figure A.5.

Besides sending and receiving messages, the model also has `IF` branches conditioned over symbolic expressions, like that shown in Figure A.2. Upon nondeterministic

```
let observe (server) =
  match server with
  | pkt := recv(); s'(pkt) ⇒
    p := gen_pkt(); send(p); observe (s'(p))
  | send(pkt); s' ⇒
    p := recv(); guard(pkt, p); observe (s')
  | IF (x, s1, s2) ⇒
    (* Allow validating observation with [s1],
     * provided [x] is unifiable with [true];
     * Or, unify [x] with [false],
     * and validate observation with [s2]. *)
    determine(unify(x, true ); observe (s1),
              unify(x, false); observe (s2))
  | r  := _(); s'(r) ⇒
    r1 := _(); observe (s'(r1))
  end
```

FIGURE A.5. Dualizing server model into observer model. Upon `recv` events, the observer generates a packet and sends it to the server. For `send` events, the observer receives a packet `p1`, and fails if it does not match the specified `pkt`. When the server makes nondeterminstic `IF` branches, the observer `determine`s between the branches by `unify`ing the branch condition with its conjectured value, and then observing the corresponding branch.

branching, the tester needs to determine which branch was actually taken, by constructing observers for both branches. Each branch represents a possible explanation of the server's behavior. Upon further interacting with the server, some branches might fail because its conjecture cannot explain what it has observed. The tester rejects the server if all branches have failed, indicating that the server corresponds to no possible case in the model.

Dualizing the server-side model produces an observer model that performs interactions to reveal the server's behavior and check its validity. This model includes all possible observations from a valid server, and needs to `determine` which branch in the server model matches the observed behavior. The model validates its observations with unification events `unify` and `guard`. These primitive events are handled by later interpretations: The `unify` and `guard` events in each branch are instantiated into symbolic evaluation logic that decides whether this branch should fail or not; The `determine` events are instantiated into backtracking searches to find if all branches have failed, which rejects the server.

Symbolic Evaluation. In this interpretation phase, we handle nondeterminism at data level by handling `fresh` events in the server model, as well as `unify` and `guard` events introduced by dualization. The interpretor instantiates these events into symbolic evaluation algorithms.

```
(* unifyS = list variable * list constraint    *)
(* new_var : unifyS → variable * unifyS        *)
(* assert : exp T * T * unifyS → option unifyS *)
let unifier (observer, map : mcid → pcid,
         vars : unifyS) =
  match observer with
  | x := fresh(); o'(x) ⇒
    let (x1, vars') = new_var(vars) in
    unifier (o'(x1), vars', map)
  | unify(x, v); o' ⇒
    match assert(x, v, vars) with
    | Some vars' ⇒ unifier (o', vars', map)
    | None ⇒ failwith "Unexpected payload"
    end
  | guard(p0, p1); o' ⇒
    match assert(p0, p1, vars) with
    | Some vars' ⇒
      let mc = p0.source in
      let pc = p1.source in
      if mc.is_created_by_server
      then match map[mc] with
           | pc ⇒ unifier (o', vars', map)
           | unknown ⇒
             let map' = update(map, mc, pc) in
             unifier (o', vars', map')
           | others ⇒
             failwith "Unexpected connection"
           end
      else unifier (o', vars', map)
    | None ⇒ failwith "Unexpected payload"
    end
  | r  := _(); o'(r) ⇒
    r1 := _(); unifier (o'(r1), vars, map)
  end
```

FIGURE A.6. Instantiating symbolic events. The tester maintains a `unifyState` which stores the constraints on symbolic variables. When the specification creates a `fresh` symbol, the tester creates an entry for the symbol with no initial constraints. Upon `unify` and `guard` events, the tester checks whether the `assert`ion is compatible with the current constraints. If yes, it updates the constraints and move on; otherwise, it raises an error on the current branch.

As shown in Figure A.6 (skip Line 18–28 for now—we'll explain that part later), the tester checks whether the observed/conjectured value matches the specification, by maintaining the constraints on the symbolic variables. These constraints are initially empty when the variables are generated by `fresh` events. As the test runs into `unify` and `guard` events, it adds constraints `assert`ing that the observed value matches the specification, and checks whether the constraints are still compatible. Incompatibility among constraints indicates that the server has exhibited behavior that cannot be explained by the model, implying violation against the current branch of specification.

Handling Incoming Connections. In addition to generating data internally, the server might exhibit another kind of nondeterminism related to the outgoing connections it creates. For example, when a client uses an HTTP server as proxy, requesting resources from another server, the proxy server should create a new connection to the target server. However, as shown in **??**, when the tester receives a request from an accepted connection, it does not know which client's request the proxy was forwarding, due to network delays.

Outgoing connections created by the server model are identified by "model connection identifiers" (`mcid`), and the tester accepts incoming connections identified by "physical connection identifiers" (`pcid`). As shown in Line 18–28 of Figure A.6, to determine which `mcid` in the specification does a runtime `pcid` corresponds to, the tester maintains a `map`ping between the connection identifiers. Such mapping ensures the tester to check interactions on an accepted connection against the right connection specified by the server model.

Backtracking. Symbolic evaluation determines whether the observations matches the tester's conjectures on each branch. So far, the derived tester is a nondeterministic program that rejects the server if and only if all possible branches have raised some error. To simulate this tester on a deterministic machine, we execute one branch until it fails. Upon failure in the current branch, the simulator switches to another possible branch, until it exhausts all possibilities and rejects the server, as shown in Line 9–13 of Figure A.7.

When switching from one branch to another, the tester cannot revert its previous interactions with the server. Therefore, it must match the server model against all interactions it has performed, and filter out the mismatching branches, as shown in Line 15 and Line 21 of Figure A.7.

We've now derived a tester from the server model. The specified server runs forever, and so does the tester (upon no violations observed). We accept the server if the tester hasn't rejected it after some large, pre-determined number of steps of execution.

Test Case Generation. Counterexamples are sparsely distributed, especially when the bugs are related to server's internally generated data like ETags, which can hardly be matched by a random test case generator. After observing the `ETag` field of some response, the generator can send more requests with the same ETag value, rather than choosing an unknown value arbitrarily.

As shown in Figure A.8, our derivation framework allows passing the programs' internal state as the events' parameters, so the test case generator can utilize the states in all intermediate interpretation phases, and apply heuristics to emphasise certain bug patterns.

```
(* filter : event T * T * list M → list M *)
(* [filter(e, r, l)] returns a subset in [l],
 * where the model programs' next event is [e]
 * that returns [r]. *)
let backtrack (current, others) =
  match current with
  | determine(t1, t2) ⇒
    backtrack (t1, t2::others)
  | failwith error ⇒ (* current branch failed *)
    match others with
    | [] ⇒ failwith error
    | another::ot' ⇒ backtrack (another, ot')
    end
  | send(pkt); t' ⇒
    let ot' = filter(SEND, pkt, others) in
    send(pkt); backtrack (t', ot')
  | pkt := recv(); t'(pkt) ⇒
    opkt := maybe_recv();
    match opkt with
    | Some p1 ⇒
      let ot' = filter(RECV, pkt, others) in
      backtrack (t'(p1), ot')
    | None ⇒             (* no packet arrived *)
      match others with
      | [] ⇒ backtrack (current, []) (* retry *)
      | another::ot' ⇒            (* postpone *)
        backtrack (another, ot'++[current])
      end
    end
  end in
backtrack (tester_nondet, [])
```

FIGURE A.7. From nondeterministic model to deterministic tester program. If the model makes nondeterministic branches, the tester picks a branch to start with, and puts the other branch into a set of other possibilities. If the current branch has failed, the tester looks for other possible branches to continue checking. When the current branch sends a packet, the tester filters the set of other possibilities, and only keeps the branches that match the current send event. If the model wants to receive a packet, the tester handles both cases whether some packet has arrived or not.

```
let http_server (http_st) =
  request := recv_HTTP(http_st);
  (response, st') := process(request, http_st);
  http_server (st')
...
let observer (server) =
  match server with
  | req := recv_HTTP(http_st); s'(req) ⇒
    r1  := gen_Observer(http_st);
    send(r1); observe (s'(r1))
...
let unifier (observer, vars, conn) =
  match observer with
  | req := gen_Observer(http_st); o'(req) ⇒
    r1  := gen_Unifier(http_st, vars, conn);
    unifier (o'(r1), vars, conn)
...
```

FIGURE A.8. Embedding programs' internal state into the events. By expanding the events' parameters, we enrich the test case generator's knowledge along the interpretations.

Notice that the state-passing strategy only allows tuning *what* messages to send. To reveal bugs more efficiently in an interactive scenario, we need to tune *when* the interactions are made, which is further discussed in Subsection A.4.2. Generating test cases in certain orders is to be explored in future work.

```
1  let compose (net, bi, bo, srv) =
2    let step_net =
3      match net with
4      | send(pkt); n' ⇒
5        if pkt.to_server
6        then compose (n', bi++[pkt], bo, srv)
7        else send(pkt);    (* to client *)
8             compose (n', bi, bo, srv)
9        end
10     | pkt := recv(); n'(pkt) ⇒
11       match bo with
12       | p0::b' ⇒ compose (n'(p0), bi, b', srv)
13       | []     ⇒ p1 := recv();
14                  compose (n'(p1), bi, bo, srv)
15       end
16     | r  := _(); n'(r) ⇒
17       r1 := _(); compose (n'(r1), bi, bo, srv)
18     end in
19   match srv with
20   | send(pkt); s' ⇒
21     compose (net, bi, bo++[pkt], s')
22   | pkt := recv(); s'(pkt) ⇒
23     match bi with
24     | p0::b' ⇒ compose (net, b', bo, s'(p0))
25     | []     ⇒ step_net
26     end
27   | r  := _(); s'(r) ⇒
28     r1 := _(); compose (net, bi, bo, s'(r1))
29   end in
30 compose (tcp, [], [], http)
31
```

FIGURE A.9. Composing `http` server model with `tcp` network model by interpreting their events and passing messages from one model to another. The composing function takes four parameters: server and network models as `srv` and `net`, and the message buffers between them. When `srv` wants to `send` a packet in Line 21, the packet is appended to the outgoing buffer `bo` until absorbed by `net` in Line 12, and eventually emitted to the client in Line 7. Conversely, packets sent by clients are absorbed by `net` in Line 13, emitted to the application's incoming buffer `bi` in Line 6, until `srv` consumes it in Line 24.

A.3.3. Network Composition. We have shown how to derive a tester from the server model itself. The server model describes how a reference server processes messages. For protocols like HTTP/1.1 where servers are expected to handle one request at a time, a reasonable server model should be "linear" that serves one client after another. As a result, the derived tester only simulates a single client, and does not attempt to observe the server's behavior via multiple simultaneous connections.

The network model describes how messages sent by one end of the network are eventually received by the other end. When interacting with multiple clients, a valid server's observable behavior should be explainable by "server delayed by the network", as discussed in Subsection A.2.1. To model this set of observations, we compose the server and network models by attaching the server model as one end on the network model.

As shown in Figure A.9, we `compose` the events of server and network models. Messages sent by the server are received by the network and sent to clients after some delay, and vice versa. Such composition produces a model that branches nondeterministically, and includes all possible interactions of a valid HTTP server that appear on the client side.

The composed model does not introduce new events that were not included in the server model: The network model in Figure A.1 does perform nondeterministc `or` branches, but `or(x,y)` is a syntactic sugar for `b := fresh(); IF(b,x,y)`. Therefore, using the same derivation algorithm from the server model to single-connection tester program, we can derive the composed server+network model into a multi-connection tester.

Notice that the server and network events are scheduled at different priorities: The composition algorithm steps into the network model lazily, not until the server is blocked in Line 25. When the network wants to `recv` some packet in Line 10, it prioritizes packets sent by the server, and only receives from the clients if the server's outgoing buffer has been exhausted. Such design is to enforce the tester to terminate upon observing invalid behavior: When the server's behavior violates the model, the tester should check all possible branches and determine that none of them can lead to such behavior. If the model steps further into the network, it would include infinitely many `absorb` branches in Figure A.1, so the derived tester will never exhaust "all" branches and reject the server. Scheduling network events only when the server model is blocked produces sufficient nondeterminism to accept valid servers.

A.4. Evaluation

To evaluate whether our derived tester is effective at finding bugs, we ran the tester against mainstream HTTP servers, as well as server implementations with bugs inserted by us.

A.4.1. Experiment Setup.

Systems Under Test (SUTs). We ran the tests against Apache HTTP Server [4], which is among the most popular servers on the World Wide Web. We used the latest release 2.4.46, and edited the configuration file to enable WebDAV and proxy modules. Our tester found a violation against RFC 7232 in the Apache server, so we modified its source code before creating mutants.

We've also tried testing Nginx and found another violation against RFC 7232. However, the module structure of Nginx made it difficult to fix the bug instantly.

(The issue was first reported 8 years ago and still not fixed!) Therefore, no mutation testing was performed on Nginx.

Infrastructure. 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 SUT was deployed as a Docker instance, using the same host machine as the tester runs on. They communicate with POSIX system calls, in the same way as over Internet except using address `localhost`. The round-trip time (RTT) of local loopback is 0.08 ± 0.04 microsecond (at 90% confidence).

A.4.2. Results.

Finding Bugs in Real-World Servers and Mutants. Our tester rejected the unmodified Apache HTTP Server, which uses strong comparison for PUT requests conditioned over `If-None-Match`, while RFC 7232 specified that `If-None-Match` preconditions must be evaluated with weak comparison[BCP: What are strong and weak comparison? [LYS: ETag jargons.]]. We reported this bug to the developers, and figured out that Apache was conforming with an obsoleted HTTP/1.1 standard [7]. The latest standard has changed the semantics of `If-None-Match` preconditions, but Apache didn't update the logic correspondingly.

We created 20 mutants by manually modifying the Apache source code. The tester rejected all the 20 mutants, located in various modules of the Apache server: `core`, `http`, `dav`, and `proxy`. They appear both in control flow (*e.g.*, early return, skipped condition) and in data values (*e.g.*, wrong arguments, flip bit, buffer off by one byte).

We didn't use automatic mutant generators because (i) Existing tools could not mutate all modules we're interested in; and (ii) The automatically generated mutants could not cause semantic violations against our protocol specification.

When testing Nginx, we found that the server did not check the preconditions of PUT requests. We then browsed the Nginx bug tracker and found a similar ticket opened by Haverbeke [8]. These results show that our tester is capable of finding bugs in server implementations, including those we're unaware of.

Performance. As shown in Figure A.10, the tester rejected all buggy implementations within 1 minute. In most cases, the tester could find the bug within 1 second.

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 our generator didn't optimize for, or (2) The tester did produce a counter-example, but failed to reject the wrong behavior. We determine the real cause by analysing 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 A.11.
- 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

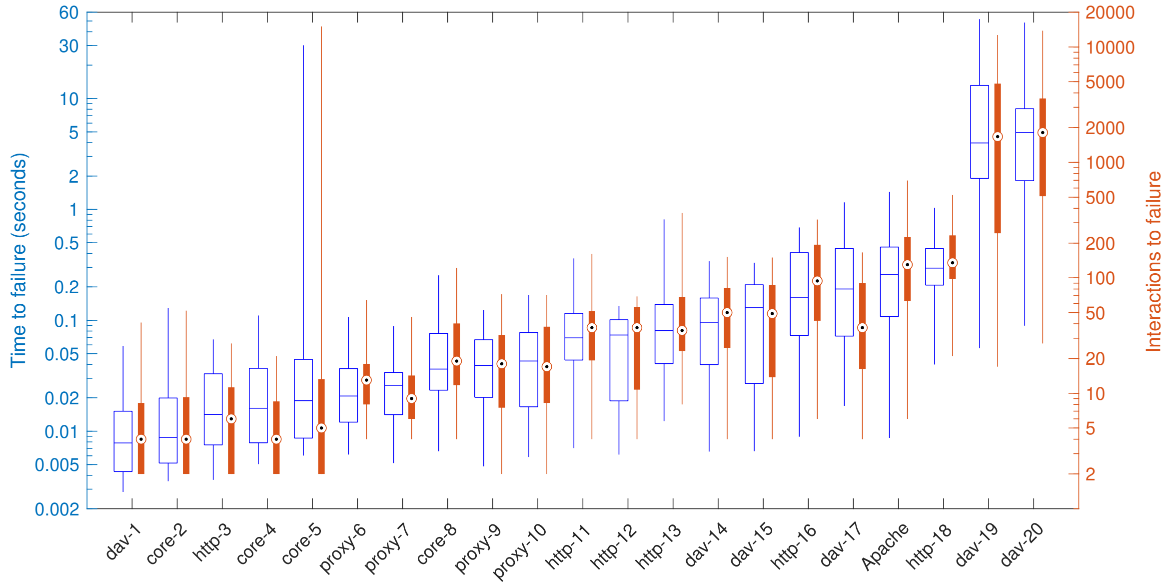


FIGURE A.10. Cost of detecting bug in each server/mutant. The left box with median line is the tester's execution time before rejecting the server, which includes interacting with the server and checking its responses. The right bar with median circle is the number of HTTP/1.1 messages sent and received by the tester before finding the bug. Results beyond 25%–75% are covered by whiskers.

such violation, a minimal counterexample consists of 4 messages: (1) GET request, (2) 200 OK response with some ETag `x`, (3) GET request conditioned over `If-None-Match: x`, and (4) 200 OK response, indicating that the ETag `x` did not match itself. Notice that (2) must be observed before (3), otherwise the tester will not reject the server, with a similar reason as Figure A.11.

- Mutant 5 causes the server to skip some code in the core module, and send nonscence messages when it should respond with 404 Not Found. The counterexample can be as small as one GET request on a non-existential target, followed by a non-404, non-200 response. However, our tester generates request targets within a small range, so the requests' targets are likely to be created by the tester's previous PUT requests. Narrowing the range of test case generation might improve the performance in aforementioned Mutants 18–20, but Mutant 5 shows that it could also degrade the performance of finding some bugs.
- The mutants in proxy module caused the server to forward wrong requests or responses. When the origin server part of the tester accepts a connection from the proxy, it does not know for which client the proxy is forwarding requests. Therefore, the tester needs to check the requests sent by all clients, and make sure none of them matches the incoming proxy request, before rejecting the proxy.

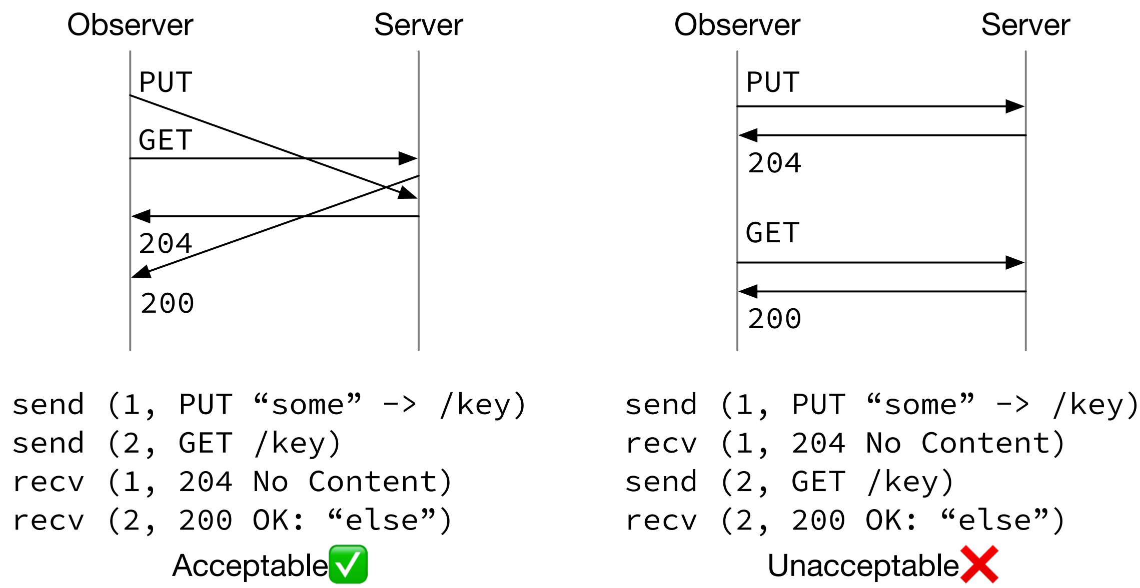


FIGURE A.11. The trace on the left does not convince the tester that the server is buggy, because there exists a certain network delay that explains why the PUT request was not reflected in the 200 response. When the trace is ordered as shown on the right, the tester cannot imagine any network reordering that causes such observation, thus must reject the server.

These examples show that the time-consuming issue of some mutants are likely caused by limitations in the test case generators. Cases like Mutant 5 can be optimized by tuning the request generator based on the tester model's runtime state, but for Mutants 18–20, the requests should be sent at specific time periods so that the resulting trace is unacceptable per specification. How to produce a specific order of messages is to be explored in future work.