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 [4] 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

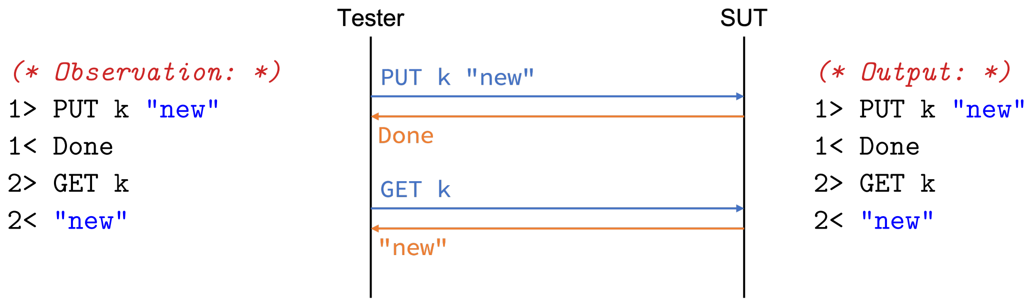


FIGURE 1.2. Upon no concurrency, the observation is identical to the output.

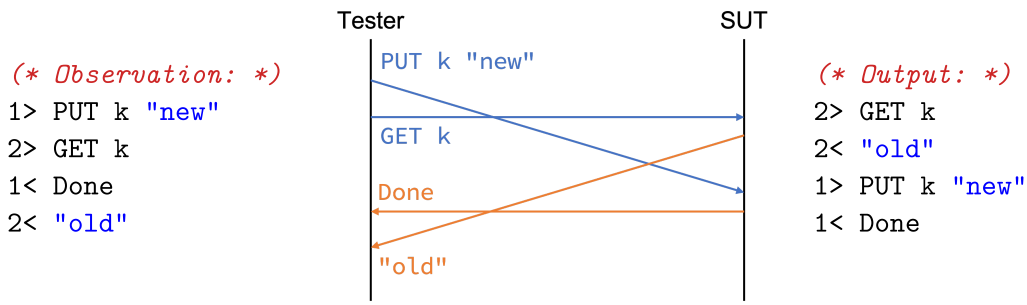


FIGURE 1.3. Acceptable: The observation can be explained by a valid output reordered by the network environment.

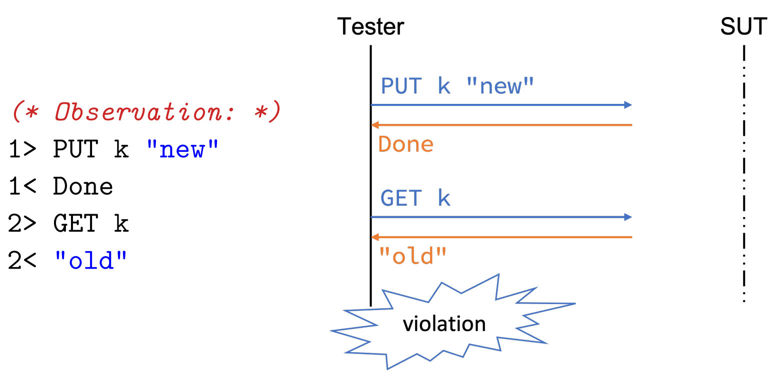


FIGURE 1.4. Unacceptable: The tester received the `Done` response before sending the `GET` request, thus the SUT must have processed the `PUT` request before the `GET` request. Therefore, the `"old"` response must be invalid.

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.

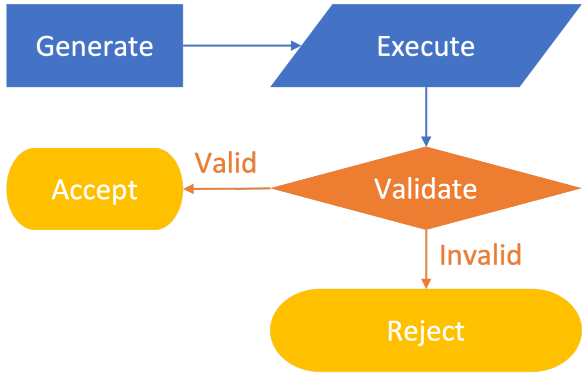


FIGURE 1.5. Simple tester architecture without shrinking.

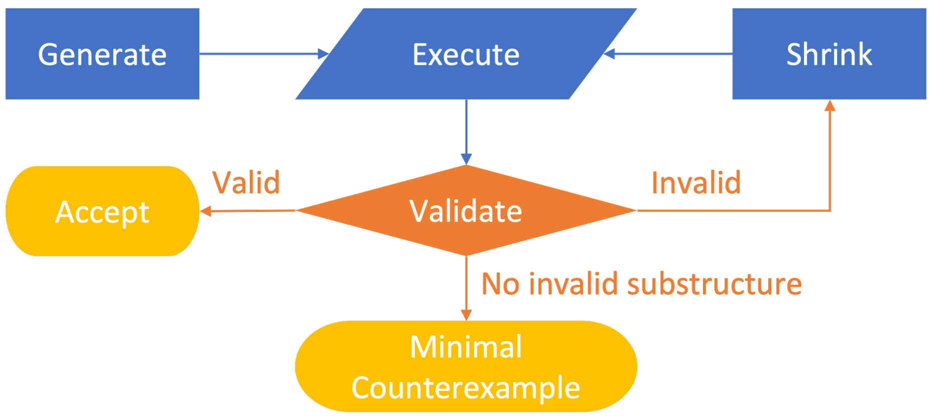


FIGURE 1.6. Tester architecture with shrinking mechanism.

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, 29 Mar 2022 10:06:20 GMT
                           /* Server: */
                           HTTP/1.1 200 OK
                           Last-Modified: Wed, 30 Mar 2022 10:06:20 GMT
                           ... content of target ...
/* Client: */
GET /index.html HTTP/1.1
If-Modified-Since: Wed, 30 Mar 2022 10:06:20 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* [9], 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 [15], and evaluated the tester's effectiveness by mutation testing.

(2) *Verifying an HTTP Key-Value Server with Interaction Trees and VST* [16], 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* [10], 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 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, Validity, and Compliance). A *specification* is a description of valid traces. "Trace t is *valid* per specification s" is written as "$\mathsf{valid}_s\ t$".

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

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

Definition 2.3 (Tester components and correctness). A tester consists of (i) a *validator* that accepts or rejects traces, 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.

[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 [13], but explicitly use "rejection-" prefix for clarity. "Rejection soundness" is equivalent to "acceptance completeness", and vice versa.

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:

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

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.

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

DEFINITION 2.5 (Deterministic server model). 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:

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

Here $\mathsf{pack}\ S = \sigma\ \mathsf{with}\ (\mathsf{sstep}, \mathit{state})$ is an instance of the $\mathsf{DeterministicServer}$ existential type [11], where sstep is of type $Q \times \sigma \to A \times \sigma$, and state has type σ.

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

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

Such server can be modelled as:

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

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.

Definition 2.6 (Nondeterministic server model). Let C be the space of invisible choices, then a nondeterministic server is specified as:

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

Consider changing the aforementioned CMP-SET into CMP-RST: When the request is greater than `S`, the server resets `S` to a random number:

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

Its corresponding server model can be written as

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

2.2.2. Validating traces. In the QAC language family, a trace is a list of $Q \times A$ pairs. The validator takes a trace and determines whether it is valid per the protocol specification.

Definition 2.7 (Trace validity). Trace t is valid per protocol specification s (written as "$\mathsf{valid}_s\ t$") if and only if it can be *produced* by the specification *i.e.* server model:

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

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

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

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

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

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

The validator is encoded as an infinite loop, where the loop body is a state monad that determines whether each $Q \times A$ pair is valid.

Definition 2.8 (Validator). Let V be some validator state type, then a validator starts from an initial state, takes a query and its corresponding response, determines whether the interaction are valid, and computes the next validator state upon valid:

$$\begin{aligned}
&\mathsf{Validator} \triangleq \{\exists V, (Q \times A \times V \to \mathsf{option}\ V) \times V\} \\
&\mathsf{stepValidator} : Q \times A \times \mathsf{Validator} \to \mathsf{option\ Validator} \\
&\mathsf{stepValidator}(q, a, \mathsf{pack}\ V = \beta\ \mathsf{with}\ (\mathsf{vstep}, \mathit{state})) \\
&\qquad \triangleq \begin{cases} \mathsf{Some}\ (\mathsf{pack}\ V = \beta\ \mathsf{with}\ (\mathsf{vstep}, \mathit{state}')) & \mathsf{vstep}(q, a, \mathit{state}) = \mathsf{Some}\ \mathit{state}' \\ \mathsf{None} & \mathsf{vstep}(q, a, \mathit{state}) = \mathsf{None} \end{cases}
\end{aligned}$$

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

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

Definition 2.9 (Trace acceptance). A validator accepts a trace if its step function *consumes* the entire trace:

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

Here the cossumability relation "$v \xrightarrow{t} v'$ " is pronounced "validator v can consume trace t and step into validator v' ":

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

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

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

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

2.2.3. Soundness and completeness of validators. We can now phrase the correctness properties in Section 2.1 in terms of the QAC language family:

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

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

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

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

2.3. Dualizing specifications into validators

So far we have defined the QAC language family, where specifications and validators are represented as state monads. This section will show how to derive validators from the specification.

2.3.1. Encoding specifications and validators. To write an algorithm from the specification to the validator, we need to analyze the computations defined by the specification's model program. The QAC language family only provides a state monad interface, which is not destructable by itself. We need to introduce a programming language to represent the specification, and derive validators by interpreting programs written in that language.

Definition 2.10 (Server and validator of a program). A program $p \in \mathsf{QAC\text{-}Lang}$ is a representation of computation that can be "instantiated" into a server model:

$$\mathsf{serverOf} : \mathsf{QAC\text{-}Lang} \to \mathsf{Server}$$

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

$$\mathsf{validatorOf} : \mathsf{QAC\text{-}Lang} \to \mathsf{Validator}$$

To encode specifications for protocols like CMP-RST, I introduce a simple language Prog in the QAC family, which supports arithmetic operations and memory access:

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

Servers specified in this Prog language are defined as follows:

(1) The server state is a key-value mapping, where the keys are natural numbers, and the values are integers.
(2) The initial server state maps all keys to zero:

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

(3) The server's query, response, and choices (Q, A, C) are all natural numbers.
(4) At the beginning of each server loop, the query is written to address !0, and the internal choice is written to address !1.
(5) After writing the query and response, the server $\mathsf{execute}$s the Prog model, which manipulates the key-value store.
(6) When the Prog model returns, the server sends back the value stored in address !0 as the response.

Let $p \in \mathsf{Prog}$ be the model program, then the server's loop body sstep_p is defined as:

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

Here "e^s" is pronounced "evaluating server expression ($e : \mathsf{SExp}$) with state ($s : \mathbb{N} \to \mathbb{Z}$)". It substitutes all occurences of "$!src$" with the value stored at address src of mapping s, written as "$s!src$". "$s[k \mapsto v]$" is pronounced "updating mapping s at address k to value v". It produces a new state where k is mapped to v, while other addresses remain unchanged from s:

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

To specify protocols with this Prog language, the model program should read the query from address !0, and parameterize the space of nondeterministic behavior over the internal choice in address !1. When the model 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 CMP-RST specification in Section 2.2 can be written in Prog as:

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

When the query is less than or equal to the value stored in !2 (case 1), the server writes response 0 to address !0, and leave address !2 untouched. For queries greater than the value in !2 (case 2), the server writes 1 as response, and updates address !2 with the internal choice stored in !1.

This Prog language features arithmetic operations, conditional branches, memory access, and internal nondeterminism. It also exhibits a tree structure that allows inductive reasoning. The rest of this section derives validators from Prog models, and prove the correctness of such derived validators.

2.3.2. Dualize model program into validator. The validator of a model $p \in \mathsf{Prog}$ needs to determine whether the trace is producible by p. More specifically, whether the responses in the trace can be *explained* by p's return value stored at address !0.

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

However, when the specification is nondeterministic, the expectation of response A is parameterized over the internal choice C. Therefore, the validator should determine whether there exists such C that led the specification to produce the observed A.

This reduces the trace validation problem to constraint solving. Upon observing a response, the validator adds a constraint that the observation can be explained by running the specification with certain value of choices.

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

To achieve this goal, the validator needs to store the symbolic expression for each address of the server model. It also needs to remember all the constraints added upon observation. I store these information as "validation states":

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

Here the $\mathsf{constraint}$s are relations between validator expressions (VExps) that may depend on symbolic variables:

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

In practice, I use an equivalent definition for the validator state:

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

The key-expression mapping $(k \mapsto e)$ above can be simulated with the key-variable $(k \mapsto x)$ mapping here, by adding $(\#x \equiv e)$ to the set of constraints. I alter the type interface for convenience of developing the validator.

Notice that the internal choices might affect branch conditions, so the validator doesn't know which branch in the specification was taken. Therefore, it should maintain multiple validation states, one for each possible execution path of the specification:

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

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

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

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

The validator's loop body is derived by dualzing the server model:

(1) When the server performs a write operation $!\mathit{dst} := \mathit{exp}$, the validator creates a fresh variable x to represent the new value stored in address $!\mathit{dst}$, and adds a constraint that says x's value is equal to that of exp.

$$
\begin{array}{llll}
\mathsf{validatorOf}(p) & \triangleq & \mathsf{pack}\ V = \mathsf{set}\ ((\mathbb{N} \to \mathsf{var}) \times \mathsf{set\ constraint})\ \mathsf{with} & \\
 & & (\mathsf{vstep}_p, \{(\_ \mapsto \#0, \{\#0 \equiv 0\})\}) & \\
\mathsf{vstep}_p(q, a, v) & \triangleq & \mathsf{let}\ v' = v_0 \leftarrow v; \mathsf{vstep}'_p(q, a, v_0)\ \mathsf{in} & \\
 & & \mathsf{if}\ v'\ \mathsf{is}\ \varnothing\ \mathsf{then\ None\ else\ Some}\ v' & (6) \\
\mathsf{vstep}'_p(q, a, v_0) & \triangleq & \mathsf{let}\ v_1 = \mathsf{havoc}(1, v_0)\ \mathsf{in} & \\
 & & \mathsf{let}\ v_2 = \mathsf{write}(0, q, v_1)\ \mathsf{in} & \\
 & & (vs_3, cs_3) \leftarrow \mathsf{exec}(p, v_2); & \\
 & & \mathsf{let}\ cs_4 = cs_3 \cup \{\#(vs_3!0) \equiv a\}\ \mathsf{in} & (4) \\
 & & \mathsf{if\ solvable}\ cs_4\ \mathsf{then}\ \{(vs_3, cs_4)\}\ \mathsf{else}\ \varnothing & (5) \\
\mathsf{exec}(p, (vs, cs)) & \triangleq & \begin{cases} \{(vs, cs)\} & p\ \mathsf{is\ return} \\ \mathsf{exec}(p', \mathsf{write}(d, e, (vs, cs))) & p\ \mathsf{is}\ !d := e; p' \\ \begin{pmatrix} \mathsf{let}\ v_1 = (vs, cs \cup \{{e_1}^{vs} \leq {e_2}^{vs}\})\ \mathsf{in} \\ \mathsf{let}\ v_2 = (vs, cs \cup \{{e_2}^{vs} < {e_1}^{vs}\})\ \mathsf{in} \\ \mathsf{exec}(p_1, v_1) \cup \mathsf{exec}(p_2, v_2) \qquad (2) \end{pmatrix} & \begin{array}{l} p\ \mathsf{is} \\ \mathsf{if}\ e_1 \leq e_2 \\ \mathsf{then}\ p_1\ \mathsf{else}\ p_2 \end{array} \end{cases} & \\
\mathsf{write}(d, e, (vs, cs)) & \triangleq & \mathsf{let}\ x_e = \mathsf{fresh}\ (vs, cs)\ \mathsf{in} & (1) \\
 & & (vs[d \mapsto x_e], cs \cup \{\#x_e \equiv e^{vs}\}) & \\
\mathsf{havoc}(d, (vs, cs)) & \triangleq & \mathsf{let}\ x_c = \mathsf{fresh}\ (vs, cs)\ \mathsf{in}\ (vs[d \mapsto x_c], cs) & (3) \\
e^{vs} & \triangleq & \begin{cases} n & e\ \mathsf{is}\ n : \mathbb{N} \\ \#(vs!src) & e\ \mathsf{is}\ !src \\ {e_1}^{vs}\ op\ {e_2}^{vs} & e\ \mathsf{is}\ e_1\ op\ e_2 \end{cases} &
\end{array}
$$

FIGURE 2.1. Dualizing server model into validator, with derivation rules annotated.

(2) When the server makes a nondeterministic branch if $e_1 \leq e_2$ then p_1 else p_2, consider both cases: (a) If p_1 was taken, then the validator should add a constraint $e_1 \leq e_2$; or (b) If p_2 was taken, then the validator should add constraint $e_2 < e_1$.

(3) Before executing the program, the server writes the internal choice c to address !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.

(5) When the constraints of a validation state becomes unsatisfiable, it indicates that the server model cannot explain the observation. This is because either (i) the observation is invalid *i.e.* not producible by the server model, or (ii) the observation is valid, but was produced by a different execution path of the server model.

(6) The validator accepts the trace if it can be produced by any execution path of the server model. Since each execution path corresponds to a validation state, the validator only needs to remove the unsatisfiable state from the set of states. If the set of validation states becomes empty, it indicates that the

observation cannot be explained by any execution path of the specification, so the validator should reject the trace.

This mechanism is formalized in Figure 2.1. Here notation "$v_0 \leftarrow v; \mathsf{vstep}'_p(q, a, v_0)$" is a monadic bind for sets: Let vstep'_p map each element v_0 in v to a set of validation states ($\mathsf{vstep}'_p(q, a, v_0) : \mathsf{set}\ ((\mathbb{N} \to \mathsf{var}) \times \mathsf{set\ constraint})$), and return the union of all result sets as v'.

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

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

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

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

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

When the Prog model writes to memories or makes conditional branches, the operands are represented as specification expressions (SExp) that refer to server addresses. To construct the constraints over symbolic variables, the validator translates the expressions ($e : \mathsf{SExp}$) into validator expressions ($e^{vs} : \mathsf{VExp}$) by *symbolizing* it with the validation state ($vs : \mathbb{N} \to \mathsf{var}$), which substitutes all addresses ($!src$) with their corresponding variable $\#(vs!src)$.

For example, by dualizing the Prog model for CMP-RST in Subsection 2.3.1, we get a validator as shown in Figure 2.2. Such derived validators are proven sound and complete in the following section.

2.4. Soundness and completeness of derived validators

So far I have introduced the QAC language family for representing servers and validators, and demonstrated the derivation mechanism with a Prog language. Next I'll show how to prove that QAC validators are sound and complete:

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

This section first presents a generic framework for proving validators' correctness properties, and then demonstrates its usage by applying it to Prog-based validators.

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

FIGURE 2.2. Validator for CMP-RST, derived from Prog model. This program consists of three parts: (1) symbolizing the query and internal choice before executing the model, (2) considering both branches in the model program, propagating a validation state for each branch, (3) filtering the validation states by constraint satisfiability, removing invalid states.

2.4.1. Proof strategy. Both the specification and the validator are infinite loops, and the correctness property is defined as equivalence between production and consumption of traces. Therefore, we can prove this bisimulation relation by introducing some loop invariant, and show that it is preserved in each step between the specification and the validator.

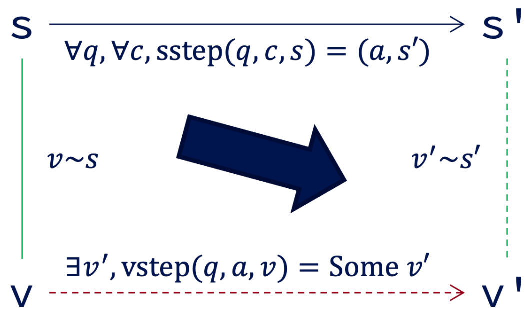
Rejection soundness (acceptance completeness). To prove that any trace producible by server $\mathsf{pack}\ S = \sigma\ \mathsf{with}\ (\mathsf{sstep}, s_0)$ is consumable by validator $\mathsf{pack}\ V = \beta\ \mathsf{with}\ (\mathsf{vstep}, v_0)$, we need forward induction on the server's execution path, and show that every step has a corresponding validator step:

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

$$(v_0 : \beta) \sim (s_0 : \sigma) \qquad \text{(RejSound-Init)}$$

- Any server step $\mathsf{sstep}(q, c, s) = (a, s')$ whose pre-execution state s reflects some pre-validation state v can be consumed by the validator into a post-validation state v' that reflects the post-execution state s':

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



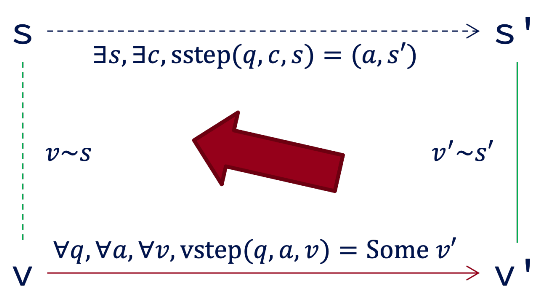
Rejection completeness (acceptance soundness). To prove that any trace consumable by validator $\mathsf{pack}\ V = \beta\ \mathsf{with}\ (\mathsf{vstep}, v_0)$ is producible by server $\mathsf{pack}\ S = \sigma\ \mathsf{with}\ (\mathsf{sstep}, s_0)$, we need backward induction on the validator's execution path, and show that every step has a corresponding server step:

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

$$\begin{aligned} &\forall (q : Q)(a : A)(v, v' : \beta), \mathsf{vstep}(q, a, v) = \mathsf{Some}\ v' \\ &\qquad\qquad \Longrightarrow \exists s' : \sigma, v' \sim s' \end{aligned} \tag{RejComplete-End}$$

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

$$\begin{aligned} &\forall (q : Q)(a : A)(v, v' : \beta)(s' : \sigma), \\ &\mathsf{vstep}(q, a, v) = \mathsf{Some}\ v' \wedge v' \sim s' \\ &\quad \Longrightarrow \exists (s : \sigma)(c : C), \mathsf{sstep}(q, c, s) = (a, s') \wedge v \sim s \end{aligned} \tag{RejComplete-Step}$$



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

$$\{s \mid v_0 \sim s\} = \{s_0\} \tag{RejComplete-Init}$$

Rejection soundness is proven by forward induction, while rejection completeness is proven by backward induction. This is because the choice C is known from the server step, but unknown from the validator step: Given a validator step, we cannot predict "what choices the server will make in the future", but can analyze "what choices the server might have made in the past". This proof strategy is further explained with the Prog example.

2.4.2. Case study: Proving Prog-based validators' correctness. Specifications written in the Prog language are dualized into validators as shown in Subsection 2.3.2. Here I show that validators dualized in this way are sound and complete, using the proof strategy described in Subsection 2.4.1.

Invariant design. The hypotheses in the proof strategy are based some loop invariant, which depends on the modelling language. We need to define the invariant for the language, and show that it is preserved between the server and validator steps.

A Prog-based validator maintains a set of validation states, each state corresponds to a possible execution path of the server model.

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 validator accepts the trace if any of its validation states is accepting, which indicates that some execution path of the server model can produce the trace.

Given an accepting validation state, we can construct the server steps that produce the trace, using the assignment ($\mathsf{var} \to \mathbb{Z}$) that satisfies the constraints. This assignment evaluates internal choices' symbolic variables into concrete values, and evaluates the validator's key-variable mapping ($\mathbb{N} \to \mathsf{var}$) to the server's key-value mapping ($\mathbb{N} \to \mathbb{Z}$).

Therefore, we only need to show that each server and validator step preserves the existence of such assignment that relates their states, thus defines the invariant:

Definition 2.11 (Loop invariant between Prog-based specification and validator). Validator state v simulates server state s if it contains a validation state (vs, cs) that *reflects* the server state, *i.e.* (1) There exists an assignment $asgn$ that can satisfy the constraints cs; and (2) The key-variable mapping vs can be evaluated with $asgn$ (written as vs^{asgn}) into a key-value mapping that is equivalent with s:[3]

$$\begin{array}{lcl} (v:\beta) \sim (s:\sigma) & \triangleq & \exists((vs, cs) \in v)(asgn : \mathsf{var} \to \mathbb{Z}), asgn \ \mathsf{satisfy}\ cs \wedge vs^{asgn} \equiv s \\ vs^{asgn} & \triangleq & addr \mapsto asgn!(vs!addr) \end{array}$$

Applying proof strategy. Having defined the loop invariants, we only need to instantiate the QAC-generic proof strategy with Prog-based definitions. If the hypotheses are all satisfied, then we have the soundness and completeness guarantee of every validator derived from Prog models.

[3]For the rest of this section, $\beta = \mathsf{set}\ ((\mathbb{N} \to \mathsf{var}) \times \mathsf{set\ constraint})$ represents the validator state type, and $\sigma = \mathbb{N} \to \mathbb{Z}$ represents the server state type.

Lemma 2.1 (RejSound-Init).

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

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

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

By constructing the assignment as:

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

We have:

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

Thus:

$$asgn \ \mathsf{satisfy}\ cs$$

We also know that:

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

Thus:

$$vs^{asgn} \equiv s$$

$\square$

Lemma 2.2 (RejSound-Step).

$$\begin{aligned} &\forall (p : \mathsf{Prog})(q, c, a : \mathbb{Z})(s, s' : \sigma)(v : \beta), \\ &\mathsf{sstep}_p(q, c, s) = (a, s') \wedge v \sim s \\ &\implies \exists v' : \beta, \mathsf{vstep}_p(q, a, v) = \mathsf{Some}\ v' \wedge v' \sim s' \end{aligned}$$

Proof. The invariant $v \sim s$ tells us that v contains a validation state that reflects the server state s_0:

$$\exists (vs, cs) \in v, \exists asgn : \mathsf{var} \to \mathbb{Z}, \quad asgn \ \mathsf{satisfy}\ cs \wedge vs^{asgn} \equiv s$$

Since the server's internal choice was provided, we can compute the server's actual execution path. For each small step of the server's execution, we can construct its corresponding validator small step, based on the derivation rules in Section 2.3. By making the same internal choice and branch decisions as the server did, we can construct the assignment that unifies the validator with the server. The proof details are shown in Section A.1. $\square$

Lemma 2.3 (RejComplete-End).

$$\begin{aligned} \forall (p : \mathsf{Prog})(q, a : \mathbb{Z})(v, v' : \beta), \mathsf{vstep}_p(q, a, v) = \mathsf{Some}\ v' \\ \implies \exists s' : \sigma, v' \sim s' \end{aligned}$$

Proof. Since vstep_p checks the nonemptiness of the result, we know that v' must be nonempty. Consider validation state $(vs', cs') \in v'$. Since vstep'_p checks that $(\mathsf{solvable}\ cs')$, we know that:

$$\exists asgn, \quad asgn \ \mathsf{satisfy}\ cs'$$

Let:

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

Then we have:

$$\begin{aligned} &(vs', cs') \in v' \wedge asgn \ \mathsf{satisfy}\ cs' \wedge vs'^{\ asgn} \equiv s' \\ &i.e.\ v' \sim s' \end{aligned}$$

□

Lemma 2.4 (RejComplete-Step).

$$\begin{aligned} &\forall (p : \mathsf{Prog})(q, a : \mathbb{Z})(v, v' : \beta)(s' : \sigma), \\ &\mathsf{vstep}_p(q, a, v) = \mathsf{Some}\ v' \wedge v' \sim s' \\ &\implies \exists (s : \sigma)(c : \mathbb{Z}), \mathsf{sstep}_p(q, c, s) = (a, s') \wedge v \sim s \end{aligned}$$

Proof. We first construct the initial server state $(s : \sigma \mid v \sim s)$. We then compute the internal choice c and construct the server step that corresponds with the validator step.

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

$$\exists (vs', cs') \in v', \exists asgn, \quad asgn\ \mathsf{satisfy}\ cs' \wedge vs'^{\ asgn} \equiv s'$$

From the definition of vstep_p, we know that:

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

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

$$asgn\ \mathsf{satisfy}\ cs$$

Let:

$$s = vs^{asgn}$$

Then we have:

$$\begin{aligned} &(vs, cs) \in v \wedge asgn\ \mathsf{satisfy}\ cs \wedge vs^{asgn} \equiv s \\ &i.e.\ v \sim s \end{aligned}$$

From the definition of vstep'_p, the validator first creates a fresh variable to represent the server's internal choice. Let:

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

We now have a server step $\mathsf{sstep}_p(q, c, s)$, and need to show that it results in response a and post-execution state s'. Since the post-validation state v' simulates s' and guarantees the response to be a, we only need to show that the server step is reflected in the validator. This is done by analyzing the server's execution path, proving that each derivation rule preserves such small-step reflection. The proof details are shown in Section A.2 □

Lemma 2.5 (RejComplete-Init).

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

Proof. The requirement for s says:

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

The constraint satisfaction tells us that:

$$asgn!0 = 0$$

We then have:

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

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

Now we have proven that all Prog-based validators satisfy the hypotheses defined in Subsection 2.4.1, and conclude that these validatos are sound and complete. The entire proof is formalized in the Coq proof assistant.

The main idea of the proof is to show the reflection between the server and the validator, by constructing the assignments that unifies them. This also answers why proving rejection completeness requires backward induction: The assignment evaluates the symbolic variables during the validation process, which includes all choces made by the server, past and future. An assignment might include wrong predictions about the server's future choices, in which case the validator will drop it upon contradicting observations. By the end of validation, the surviving assignment can let us reconstruct a server's execution path, by infering its internal choices.

So far I have presented the theory of constructing validators with correctness guarantee. Next I'll explain how to apply this theory to test real-world programs.

CHAPTER 3

Testing in Practice

In Chapter 2, I introduced the theory of validators using the QAC language family. I also shown how to prove validators' correctness with a simple Prog language.

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

This chapter describes how to derive specifications into tester programs that can interact with the SUT and reveal potential defects, using HTTP/1.1 as an example. The derivation framework is shown in outline in Figure 3.1. Each box is a model program, and the arrows are "interpretors" that transform one model into another.

Section 3.1 introduces the ITree modelling language for specifying protocols and deriving them into testers. Section 3.2 and Section 3.3 address external and internal nondeterminism in the ITree context. Section 3.4 explains how to execute the derived tester model as an interactive program.

3.1. ITree Specification Language

To write specifications for protocols' rich semantics, I employed "interaction tree" (ITree), a generic data structure for representing interactive programs in the Coq programming language, introduced by Xia et al. [15]. ITree allows 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 later sections.

3.1.1. Language definition. Consider an echo program, which keeps reading some data and writing it out verbatim, until reaching EOF:

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

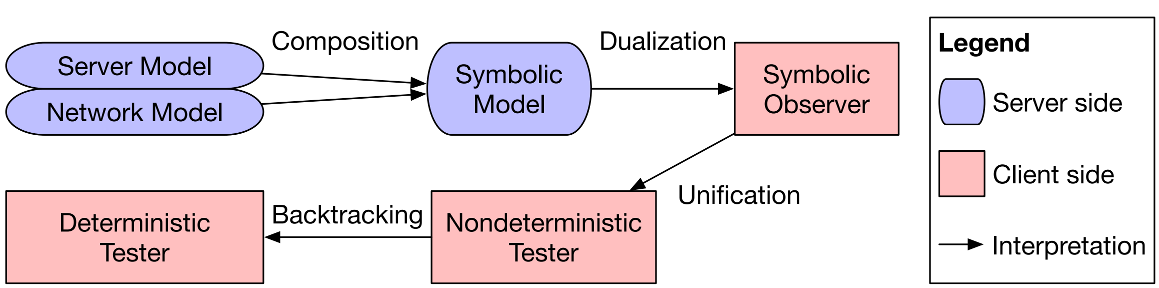


FIGURE 3.1. Deriving tester program from specification

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

FIGURE 3.2. Mock definition of interaction trees.

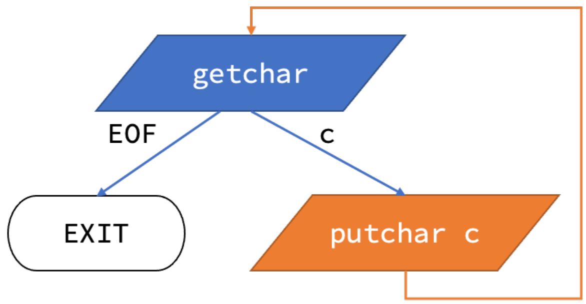


FIGURE 3.3. Interaction tree for echo program

Here the behavior after `read` depends on the value actually read. This monadic computation can be desugarized into:

```
CoInductive echo2 := (* equivalent to echo *)
  Bind getchar (λ c ⇒ if c is EOF then EXIT
                      else Bind (putchar c) (λ _ ⇒ echo)).
```

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

Mock interaction trees. As shown in Figure 3.2, a mock interaction tree (`itreeM`) has two kinds of nodes, `Ret` and `Trigger`, and has edges constructed by `Bind`:

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

  ```
  Definition EXIT {E} : itreeM E Z := Ret 0.
  ```

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

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

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

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

FIGURE 3.4. Formal definition of interaction trees

```
                                     := Trigger (PutChar c).
```

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

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

  As illustrated in Figure 3.3, each possible return value x is an edge that leads to the child it instantiates *i.e.* (k x). In this way, the Ret and Trigger nodes are connected into a tree structure.

The mock interaction tree provides an intuitive continuation-passing structure for representing impure programs. However, this language is not suitable for writing specifications and deriving them into tester programs, because the test derivation requires analyzing and transforming the specification program.

A mock interaction tree has infinitely many syntactic variants that are semantically equivalent, due to monad laws. For example, consider the following programs:

```
Example bind_ret   r k     := Bind (Ret r) k.
Example bind_bind m k1 k2 := Bind (Bind m k1) k2.
```

These programs are semantically equivalent to:

```
Example bind_ret2   r k     := k r.
Example bind_bind2 m k1 k2 := x ← m;; Bind (k1 x) k2.
```

To make program analysis more effective, we need to redefine the tree structure in a normal form, where each semantics corresponds to a unique syntax. The revised language eliminates expressions like bind_ret and bind_bind.

Practical interaction trees. [4] The type definition of ITree restricts that only single events can be bound to a continuation. As shown in Figure 3.4, I use (Impure e k) to replace (Bind (Trigger e) k) representations in itreeM. A Pure computation cannot be bound to a continuation, and must be the leaf of an ITree.

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

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

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

CoFixpoint bind {X E R} (m: itree E X) (f: X → itree E R) : itree E R :=
```

[4]For readability, the "practical" ITree definition here is a simplified version from Xia et al. [15].

```
  match m with
  | Pure   x   ⇒ f x
  | Impure e k ⇒ Impure e (λ r ⇒ bind (k r) f)
  end.

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

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

ITrees can specify various kinds of programs like servers and testers, by defining different event types. For example, the QAC server in Definition 2.6 exhibits internal nondeterminism. The internal choices made by the server can be represented as `Choice` events whose result can be any value in the space of choices:

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

The server also needs to send requests and receive responses:

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

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

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

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

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

This subsection has provided a brief taste of the ITree specification language. To construct a tester from the specification, we need to dualize the model's behavior into the tester-side behavior, based on the theory explained in Section 2.3. To dualize specifications written in ITrees, we need an *interpretation* mechanism that transforms ITrees into other programs, which will be explained in the next subsection.

3.1.2. Interpreting interaction trees. To interpret a program `p` is to specify a rule that defines "if `p` does this, then do that". For example, shell syntax `(p < input > output)` executes `p` but redirects its standard I/O. Suppose `p` is the `echo` program in Subsection 3.1.1, then the redirected program should perform file operations specified in `redirect_echo`:

```
Variant fileE: Type → Type :=      (* file operation events *)
  Fgetc: file →         fileE char
| Fputc: file → char → fileE unit.

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

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

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

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

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

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

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

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

3.2. Handling External Nondeterminism

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

This section explains how to address external nondeterminism by specifying the environment, with the networked server example. It corresponds to the "Composition" arrow in Figure 3.1. Subsection 3.2.1 defines a model for concurrent TCP connections. Subsection 3.2.2 then composes the network model with the server specification, yielding a tester-side specification that defines the space of valid observations.

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

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

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

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

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

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

Here `(or x y)` creates a nondeterministic program that may behave as either `x` or `y`. The `nondetE` here is a special case of `choiceE` defined in Subsection 3.1.1 with boolean space of choices, but they'll be handled differently during test derivation. The type signature `{E R} `{nondetE -< E}` says the `(or)` combinator can apply to ITrees whose event `E` is a super-event of `nondetE`, and with arbitrary return type `R`. For example, arguments `x` and `y` can be of type `(itree (netE ⊕ nondetE) void)`.

For example, the network model for concurrent TCP connections is defined in Figure 3.5. The model captures TCP's feature of maintaining the order within each connection, but packets in different connections might be reordered arbitrarily. When the wire chooses a packet to send, the candidate must be the oldest in its connection.

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

- When the given list `pkts` is empty, `pick_one` always returns `None`, becuase the wire has no packet in the `buffer`, and must absorb some packet before emitting anything.

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

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

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

FIGURE 3.5. Network model for concurrent TCP connections. The model is an infinite program iterating over a `buffer` of all packets en route. In each iteration, the model either `absorb`s or `emit`s some packet, depending on the current `buffer` state and the choice made in `pick_one`. Any absorbed packet is appended to the end of buffer. When emitting a packet, the model may choose a connection and send the oldest packet in it.

- Given a non-empty linked list `(p::l')`, with `p` as head and `l'` as tail, `pick_one` might return `(Some p)`, meaning the wire can emit that packet; or it might return `None`, meaning the wire can still absorb packets into the buffer.

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

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

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

To handle the asynchronicity among the server and network events, I insert message buffers between them. As shown in Figure 3.6, the *incoming buffer* stores the packets emitted by the network but not yet consumed by the server's `Recv` events, and the

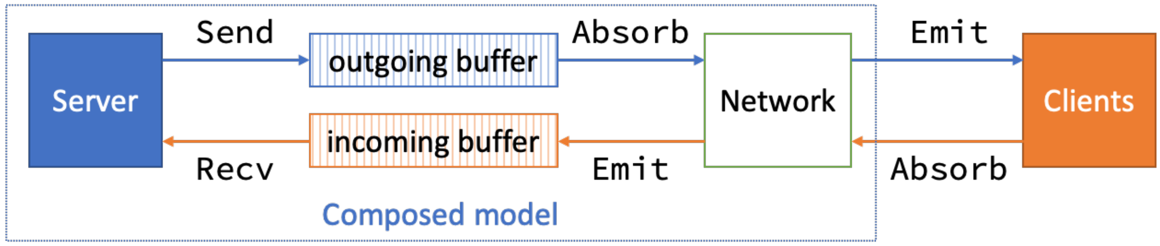


FIGURE 3.6. Network composition architecture

outgoing buffer stores the packets sent by the server but not yet absorbed by the network.

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

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

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

Now we can define the composition algorithm formally, as shown in Figure 3.7. In this example, we reuse the `qaE` definition in Subsection 3.1.1, and let the `Q` and `A` both be the `packet` type. The composed ITree takes the server and network models as parameters, and makes steps in the two ITrees in certain order.

The composed model exhibits to the client three kinds of events: (i) Network operations (`netE`) where packets are emitted to or absorbed from clients, (ii) Nondetermistic branches (`nondetE`) made by the network model, and (iii) Other events `{E}` performed by the server model *e.g.* internal choices (`choiceE`) in Subsection 3.1.1.

Notice that this algorithm schedules the server at a higher priority than the network model. The composed model only steps into the network model when the server is starved in Line 27, by calling the `step_net` process defined in Line 5. Such design is to avoid divergence of the derived tester program, which I'll further explain in Section 3.4.

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

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

Figure 3.7. Network composition algorithm. When the server wants to send a packet in Line 29, the packet is appended to the outgoing buffer until absorbed by the network in Line 9, and eventually emitted to the client in Line 16. Conversely, packets sent by clients are absorbed by the network in Line 10, emitted to the server's incoming buffer in Line 15, until the server consumes it in Line 26.

3.3. Handling Internal Nondeterminism

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

Subsection 3.3.1 explains how to represent systems' internal choices as ITree's symbolic events. It fills in the `{E}` hole in Figure 3.7, and constructs the "Symbolic Model" box in Figure 3.1. Subsection 3.3.2 and Subsection 3.3.3 takes the observer-side specification composed in Subsection 3.2.2 and interprets it into a tester model, covering the "Dualization" and "Unification" arrows in the derivation framework.

3.3.1. Symbolizing internal choices. The key idea of language design is to expose symbolic representations to the dualzation algorithm. The Prog language in Subsection 2.3.1 encodes data as symbolic expressions SExp, so that the responses and branch conditions may depend on internal choices. I do the same for ITree specifications, by symbolizing the choice events and branch conditions. Take my HTTP specification [10] as an example, its choice event has symbolic expression as result type:

```
Variant comparison := Strong | Weak.

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

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

Here I instantiate the `choiceE` in Subsection 3.1.1 with symbolic return type `(exp string)`, pronounced "expression of type string". In this example, I use strings to represent entity tags (ETags) that HTTP servers may generate, which was discussed in Subsection 1.2.1. The type interface can be adjusted to other protocols under test.

Symbolic expressions may be constructed as constant values, as variables, or with operators. The `Compare` constructor takes an expression of type string and compares it against a constant string. `(Compare t tx cmp)` represents the ETag comparison between `t` and `tx`, using "strong comparison" or "weak comparison" mechanism [4] specified by `cmp`. The constant ETag is provided by the request, and the symbolized one comes from the server state.

Figure 3.8 shows an ITree model for If-Match requests in Subsection 1.2.1. The server first evaluates the request's `If-Match` condition by "strong comparison" as required by HTTP. If the request's ETag matches its target's, then the server updates the target's contents with the request payload. The target's new ETag `tx'` can be of any value, so the model represents it as `Choice` event.

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

```
Notation σ := (path → resource).

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

FIGURE 3.8. Server model for HTTP conditional requests

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

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

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

(1) The "pure" `if` condition is used for deterministic branches like `(q.(Method) is Put)` in the example. Here `q` is a "concrete request"[5] generated by the tester and

[5]In this chapter, "concrete" messages are those that don't involve symbolic variables, as opposed to "symbolic" messages.

sent to the server, so its method is known by the tester and needn't be symbolically evaluated.

(2) The "symbolic" `IFX` condition here plays a similar rule as the if branches in the Prog language: Which branch to take depends on the server's internal choices, so the tester needs to consider both cases.

Now we can fill the hole `{E}` in Figure 3.7. The server model receives concrete requests and sends symbolic responses, so its event type is defined as:

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

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

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

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

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

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

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

This corresponds to the "Symbolic Model" in Figure 3.1. The rest of this section will explain the interpretations from this symbolic model.

3.3.2. Dualizing interactions. This subsection takes the symbolic model composed in Subsection 3.3.1 and dualizes its interactions, which corresponds to the "Dualization" arrow in Figure 3.1. It applies the derivation rules (1)–(4) for Prog in Subsection 2.3.2 to models written as ITrees.

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

Figure 3.9 shows the dualization algorithm. It interprets the symbolic model's events with the `observe` handler, whose types are explained as follows:

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

```
Notation oE := (observeE ⊕ nondetE ⊕ choiceE ⊕ constraintE).

Definition observe {R} (e: smE R) : itree oE R :=
  match e with
  | (Absorb |)     ⇒ trigger FromObserver
  | (Emit px|)     ⇒ p ← trigger ToObserver;;
                      trigger (Guard px p)
  | (|||Branch bx) ⇒ or (trigger (Unify bx true);;  ret true)
                         (trigger (Unify bx false);; ret false)
  | (|Or|)         ⇒ trigger Or
  | (||Choice|)    ⇒ trigger Choice
  end.

Definition observer_http: itree oE void :=
  interp observe sm_http.
```

FIGURE 3.9. Dualizing symbolic model into symbolic observer.

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

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

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

As discussed in Subsection 2.3.2, when the server sends a symbolic response or branches over a symbolic condition, the tester needs to create symbolic constraints accordingly. The observer introduces "constraint events" for this derivation rule:

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

Here `(Guard px p)` creates a constraint that the symbolic packet `px` emitted by the specification matches the concrete packet `p` observed during runtime. `(Unify bx b)` creates a constraint that the symbolic branch condition `bx` is unifiable with boolean value `b`. These constraints will be solved in Subsection 3.3.3.

The dualization algorithm in Figure 3.9 does the follows:

(1) When the symbolic model absorbs a packet in Line 5, the observer generates a request packet;
(2) When the symbolic model emits a symbolic packet `px` in Line 6, the observer receives a concrete packet `p`, and adds a constraint that restricts the symbolic and concrete packets match each other.
(3) When the symbolic model branches on a symbolic condition `bx` in Line 8, the tester accepts the observation if it can be explained by any branch. This is done by constructing the observer as a nondeterministic program that has both branches, using the `or` combinator. For each branch, the observer adds a constraint that the symbolic condition matches the chosen branch.
(4) Nondeterministic branches in Line 10 are preserved in this interpretation phase, and will be resolved in Section 3.4.
(5) Internal choices in Line 11 are addressed by the next phase in Subsection 3.3.3, along with the constraints created in this phase.

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

3.3.3. Symbolic evaluation. This subsection takes the symbolic observer produced in Subsection 3.3.2 and solves the constraints it has created. The constraints unify symbolic packets and branch conditions against the concrete observations. The tester should accept the SUT if the constraints are satisfiable.

As shown in Figure 3.10, the unification algorithm evaluates the primitive symbolic events into a stateful checker program, which reflects the Prog-based validator in Subsection 2.3.2. The interpretor maintains a validation state `V` which stores the symbolic variables and the constraints over them. The derivation rules are as follows:

(1) When the server makes an internal choice in Line 8, the tester creates a fresh variable and adds it to the validation state.
(2) When the observer creates a constraint in Line 10, the tester adds the constraint to the validation state, and solves the new set of constraints. If the constraints become unsatisfiable, then the tester `Throw`s an exception that indicates the current execution branch cannot accept the observations:

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

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

Notice that the `unify` function interprets a symbolic observer's event `(oE R)` into a state monad transformer `(V → itree tE (V * R))`. It makes a step from pre-validation state `(v: V)` to post-validation state `(v': V)`, and yields the event's

```
Notation ntE := (observeE ⊕ nondetE ⊕ exceptE).

Definition V: Type := list var * list (constraintE unit).

Definition unify {R} (e: oE R) (v: V) : itree ntE (V * R) :=
  let (xs, cs) := v in
  match e with
  | (||Choice|)      ⇒ let x: var := fresh v in
                       ret (x::xs, cs, Var x)
  | (|||constraint) ⇒ let cs' := constraint::cs in
                       if solvable cs'
                       then ret (xs, cs', tt)
                       else Trigger (Throw ("Conflict: " ++ print cs'))
  | (|Or|) ⇒ b ← trigger Or;; ret (v, b)
  | (oe|)  ⇒ r ← trigger oe;; ret (v, r)
  end.

Definition nondet_tester_http: itree ntE void :=
  (_, vd) ← interp_state unify observer_http initV;;
  match vd in void with end.
```

FIGURE 3.10. Resolving symbolic constraints.

corresponding result `(r: R)`. Such stateful interpretation process is handled by `interp_state`:

```
CoFixpoint interp_state {E F V R}
                        (handler: ∀ {X}, E X → V → itree F (V * X))
                        (m: itree E R) (v: V)
          : itree F (V * R) :=
  match m with
  | Pure   r   ⇒ ret (v, r)
  | Impure e k ⇒ '(v', r) ← handler e v;;
                 interp_state handler (k r) v'
  end.
```

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

3.4. Executing Tester Model

This section takes the nondetermistic tester model derived in Subsection 3.3.3 and transforms it into an interactive program. Subsection 3.4.1 handles the nondeterministic branches via backtrack execution, and produces a deterministic tester model.

Subsection 3.4.2 then interprets the deterministic tester into IO program that interacts with the SUT.

3.4.1. Backtrack execution. This subsection explains how to run the nondeterministic tester on a deterministic machine. It reflects the derivation rules (5) and (6) for Prog in Subsection 2.3.2, and constructs the "Backtracking" arrow in Figure 3.1.

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

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

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

Figure 3.11 shows the backtracking algorithm. It interacts with the SUT and checks whether the observations can be explained by the nondeterministic tester model. That is, checking whether the tester has an execution path that matches its interactions. This is done by maintaining a list of all possible branches in the tester, and checking if any of them accepts the observation.

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

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

The execution rule is defined as follows:

(1) When the tester nondeterministically branches in Line 9, randomly pick a branch `(k b)` to evaluate, and push the other branch `(k (negb b))` to the list of other possible cases.
(2) When the `current` tester throws an exception in Line 11, it indicates that the current execution path rejects the observations. The tester should try to explain its observations with other branches of the tester model. If the `others` list is empty, it indicates that the observation is beyond the specification's producible behavior, so the tester should reject the SUT.
(3) When the tester wants to observe a packet *from* itself, it generates a packet and sends it to the SUT in Line 16.

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

```
Notation tE := (clientE ⊕ genE ⊕ exceptE).

CoFixpoint backtrack (current:      itree ntE void)
                     (others: list (itree ntE void))
           : itree tE void :=
  match current with
  | Impure e k ⇒
    match e with
    | (|Or|)          ⇒ b ← trigger GenBool;;
                        backtrack (k b) (k (negb b)::others)
    | (||Throw msg)   ⇒ match others with
                        | other::ot' ⇒ backtrack other ot'
                        | []         ⇒ trigger (Throw msg)
                        end
    | (FromObserver|) ⇒ q ← trigger GenPacket;;
                        trigger (ClientSend q);;
                        let others' := expect FromObserver q others in
                        backtrack (k q) others'
    | (ToObserver|)   ⇒
      oa ← trigger ClientRecv;;
      match oa with
      | Some oa ⇒ let others' := expect ToObserver a others in
                  backtrack (k a) others'
      | None    ⇒
        match others with
        | other::ot' ⇒ backtrack other (ot'++[current]) (* postpone *)
        | []         ⇒ backtrack m     []               (* retry    *)
        end
      end
    end
  | Pure vd ⇒ match vd in void with end
  end.

Definition tester_http: itree tE void :=
  backtrack nondet_tester_http [].
```

FIGURE 3.11. Backtrack execution of nondeterministic tester.

As shown in Figure 3.12, `(expect e r l)` matches every tester in list `l` against the observation `e` that has return value `r`. For each element $\mathtt{m} \in \mathtt{l}$, if `m`'s first observer event `oe` matches the observation `e` (Line 7 and Line 8), then `match_observe` instantiates the tester's continuation function `k` with the observed result `r`. Otherwise, the tester throws an exception in Line 9, indicating that this branch cannot explain the observation because they performed different events.

```
CoFixpoint match_observe {R} (e: observeE R) (r: R)
                             (m: itree ntE (V * void))
           : itree ntE (V * void) :=
  match m with
  | Impure (oe|) k ⇒
    match oe, e with
    | FromObserver, FromObserver
    | ToObserver  , ToObserver ⇒ k r
    | _, _ ⇒ trigger (Throw ("Expect " ++ print oe
                               ++ " but observed " ++ print e))
    end
  | Impure (|e0|) k | Impure (||e0) k ⇒
    r0 ← trigger e0;;
    match_observe e r (k r0)
  | Pure (_, vd) ⇒ match vd in void with end
  end.

Definition expect {R} (e: observeE R) (r: R)
  : list (itree ntE (V * void)) → list (itree ntE (V * void))
  := map (match_observe e r).
```

FIGURE 3.12. Matching tester model against existing observation.

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

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

Notice that if the SUT keeps silent, then the tester will starve but won't reject, because (i) such silence is indistinguishable from the SUT sending a packet but delayed by the environment, and (ii) the SUT hasn't *exhibited* any violations against the specification. The starvation issue is addressed in Subsection 3.4.2.

The backtracking algorithm also explains the network composition design in Figure 3.7, where the server model is scheduled at a higher priority than the network model: Suppose the SUT has produced some invalid output, then every branch of the tester should reject its observation by throwing an exception. However, the network model is always ready to absorb packets. Evaluating the network model lazily prevents

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

FIGURE 3.13. Interpreting ITree tester to IO monad.

the composed symbolic model from having infinitely many absorbing branches. This allows the derived tester to converge to rejection upon violation.

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

3.4.2. From ITree model to IO program. The deterministic tester model derived in Figure 3.11 is an ITree program that never returns (its result type `void` has no elements). It represents a client program that keeps interacting with the SUT until it reveals a violation and throws an exception.

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

As shown in Figure 3.13, the `execute` function takes an argument `fuel` that indicates the remaining depth to explore in the ITree. If the execution ran out of fuel (Line 3), then the test accepts; If the tester model throws an exception (Line 8), then the test rejects. Otherwise, it translates the ITree's primitive events into IO computations in Coq [LYS: Cite Li-yao's SimpleIO library], which are eventually extracted into OCaml programs that can be compiled into executables that can communicate with the SUT over the operating system's network stack.

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

So far I have answered "how to tell compliant implementations from violating ones". The next chapter will answer "how to generate and shrink test input that reveal violations effectively", and unveil the techniques behind `gen_packet` in Line 13 of Figure 3.13.

CHAPTER 4

Test Harness Design

A tester consists of a validator and a test harness. Chapters 2 and 3 have explained the validator's theory and practices. This chapter presents a language-based design for test harnesses. I'll show how to generate and shrink test inputs effectively, addressing inter-execution nondeterminism.

Section 4.1 provides a brief overview of how test harnesses work. Section 4.2 explains how to write heuristics to generate interesting test inputs. Section 4.3 then shows how to keep the test inputs interesting among different executions in the shrinking process.

4.1. Overview

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

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

As shown in Figure 4.1, when the validator wants to observe a sent request, the harness generates the request and encodes it into bytes to send. Conversely, when the validator wants to observe a received response, the harness receives bytes from the environment and decodes it into abstract messages.

The generator is a randomized program that produces test inputs. One example is the `gen_packet` function in Figure 3.13. The HTTP packets generator can be naïvely implementation as shown in Figure 4.2. It fills in the request's fields with arbitrary

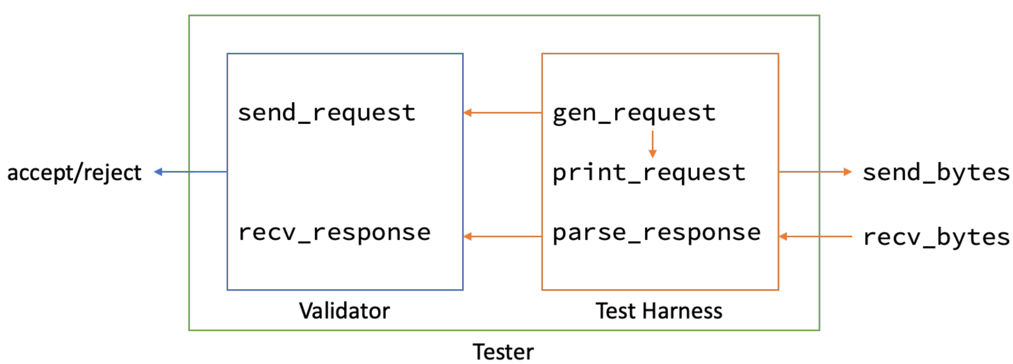


Figure 4.1. Tester architecture outline.

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

FIGURE 4.2. Naïve generator for HTTP requests.

values, and has limited coverage of the SUT's behavior. This is because the request target and ETags are both generated randomly, and thus unlikely to hit the server's resource, resulting in 404 Not Found and 412 Precondition Failed in almost all cases.

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

4.2. Heuristics for Test Generation

This section implements heuristics for generating test inputs. I'll use the HTTP tester as an example to show how to make requests more interesting, by parameterizing them over the model state (Subsection 4.2.1) and the trace (Subsection 4.2.2).

4.2.1. State-based heuristics. The model state may instruct the test generator to produce more interesting test inputs. For example, consider the `random_path` generator in Line 4 of Figure 4.2. One way to improve it is to generate more paths that have corresponding resources on the server:

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

Here I modify the server model's state type σ from `(path → resource)` in Figure 3.8 into `(list (path * resource))`, which has the same expressiveness but allows the generator to access the list of all `paths` in the server state. The generator chooses from these existent paths in 90% of the cases, as assigned by the `freq` combinator. The remaining 10% are still generated randomly, to discover how the SUT handles nonexistent paths.

For the `gen_packet` generator in Figure 4.2, replacing its `random_path` with the improved `gen_path` would generate more interesting request targets. This requires the `gen_packet` to carry the server state to instantiate `gen_path`.

As shown in Figure 3.11, the `GenPacket` generator is triggered when the tester wants to observe a packet from itself to the SUT. `fig:symbolic-observer` then shows that such `FromObserver` expectation happens when the symbolic model `Emit`s a packet. Such `Emit` event only happens when the server wants to receive a packet in Figure 3.6. The `Recv` events are triggered by the server model in Figure 3.8, which iterates over the server state σ.

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

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

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

```
Variant netE: Type → Type :=
  Emit  : packet → netE unit
| Absorb: σ       → netE packet.

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

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

Definition gen_packet: σ → IO concrete_packet.
```

As a result, when instantiating the `(GenPacket state)` event in Figure 3.13, we can feed the `gen_packet` function with argument `state`, so that `gen_path` can generate interesting paths based on the server state.

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

To improve the generator in Figure 4.2, we can generate interesting ETags based on the trace produced during execution. The trace is a list of packets sent and received by the tester, and the packets' payloads may include responses that have an ETag field. The `gen_etag` function emphasizes ETags that were observed in the trace, which are more likely to match those generated by the SUT:

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

To utilize this improved generator for ETags, the tester needs to record the trace of packets sent and received. This is done by modifying the `execute` function in Figure 3.13, adding an accumulator as the recursion parameter:

```
Fixpoint execute (fuel: nat) (trace: list concrete_packet)
                 (m: itree tE void) : IO bool :=
  match fuel with
  | S fuel' ⇒
    match m with
    | Impure e k ⇒
      match e with
      | (ClientSend q|) ⇒ client_send q;;
                         execute fuel' (trace ++ [q]) (k tt)
      | (ClientRecv|)   ⇒ oa ← client_recv;;
                         let trace' := match oa with
                                       | Some a ⇒ trace ++ [a]
                                       | None   ⇒ trace
                                       end in
                         execute fuel' trace' (k oa)
      | (|GenPacket state|) ⇒ pkt ← gen_packet state trace;;
                             execute fuel' trace (k pkt)
      ... (* similar to Figure 3.13 *)
```

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

```
Definition gen_packet (state: σ) (trace: list concrete_packet) :=
  target ← gen_path state;;
  etag   ← gen_etag trace;;
  ... (* same as Figure 4.2 *)
```

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

4.3. Shrinking Interactive Tests

Suppose we have generated a test input that has caused invalid observations of the SUT. The generated counterexample consists of (1) *signals* that are essential to

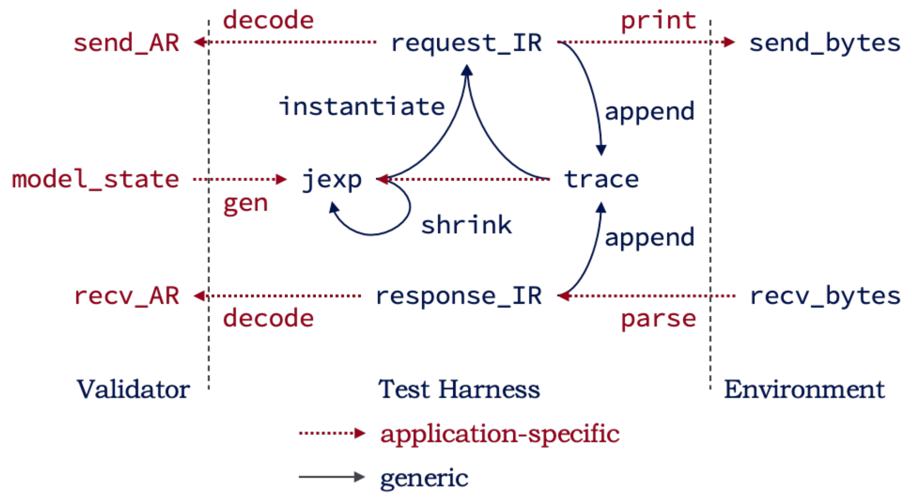


FIGURE 4.3. Test harness architecture.

triggering violations, and (2) *noises* that do not contribute to revealing such violations. We need to shrink the counterexample by removing its noises and keeping its signals.

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

To prevent turning signals into noises when rerunning the test, I shrink the heurestics instead of shrinking the generated test input. Subsection 4.3.1 introduces the architecture for interactive shrinking, then Subsection 4.3.2 explains the language design beneath that addresses inter-execution nondeterminism.

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

For example, when generating a timestamp, instead of producing concrete value *e.g.* "Wed, 30 Mar 2022 10:06:20 GMT", the generator returns an abstract representation that says "use the timestamp observed in the last response". When rerunning the test, the timestamp is computed from the new trace *e.g.* " Thu, 31 Mar 2022 10:06:20 GMT".

The test generation and shrinking framework is shown in Figure 4.3. It refines the test harness box in Figure 4.1, and involves four languages, from right to left:

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

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

FIGURE 4.4. Intermediate representation and J-expression.

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

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

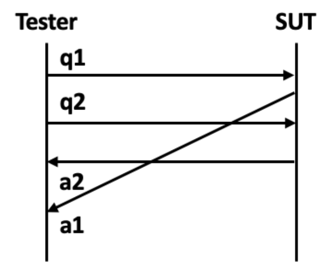


FIGURE 4.5. Labelled trace example.

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

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

The intermediate representation and J-expression will be further explained in Subsection 4.3.2.

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

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

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

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

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

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

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

FIGURE 4.6. IR, Jpath, and heuristics function example.

The J-expression is an extension of JSON that can encode trace-based heuristics. As shown in Figure 4.4, a Jexp may include syntax (*label*.Jpath.*function*) that represents trace-based heuristics, specified as:

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

    For example, consider the trace in Figure 4.5: If a trace-based heuristics is interested in `q2`'s response `a2`, then it can be encoded as "compute the test input based on message labelled 4":

    ```
    Context get_label: labelT → traceT → IR.

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

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

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

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

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

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

For example, suppose request `q2` lists files in a directory using the POSIX `ls` command, and its response `a2` is encoded as the IR shown in Figure 4.6. The response IR is a JSON object whose `"files"` field is an array of objects, each has a `"name"` and a `"mode"` field. A heuristics can refer to the second file's mode bits by Jpath (`this@"files"#2@"mode"`), which will guide the test harness to locate its corresponding value:

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

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

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

Consider the mode bits located in the previous example: If the heuristics wants to add write permission to the mode bits, it can do so with the `mode_add_write` function in Figure 4.6, which produces mode 700. Some heuristics might use the sub-IR 500 as-is, using the identity function `id`.

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

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

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

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

For example, given the runtime trace in Figure 4.5, with `a2` is defined in Figure 4.4, the the above Jexp is instantiated into the following request:

```
(* instantiate e5 labelled_trace = *)
{
  "command": "chmod",
  "args": [ 700, "bar" ]
}
```

However, when rerunning the test, the `new_trace` has a different response associated with label 4. The new response `a2'` might have fewer than 2 files in its payload.

Moreover, the response `a2'` might have not appeared in the trace, due to delays in the environment.

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

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

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

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

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

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

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

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

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

Definition t2: traceT :=
  [(3,q1); (4,j3); (5,q2); (6,a2)].
```

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

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

So far I have shown how to generate and shrink interactive test inputs and address inter-execution nondeterminism. In the next chapter, I'll combine this test harness design with the validator practice in Chapter 3, and evaluate these techniques by testing real-world systems like HTTP servers and file synchronizers.

[LYS: Under construction:]

CHAPTER 5

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.

5.1. Experiment Setup

5.1.1. Systems Under Test (SUTs). We ran the tests against Apache HTTP Server [3], 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.

5.1.2. 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).

5.2. Results

5.2.1. 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 [6]. 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).

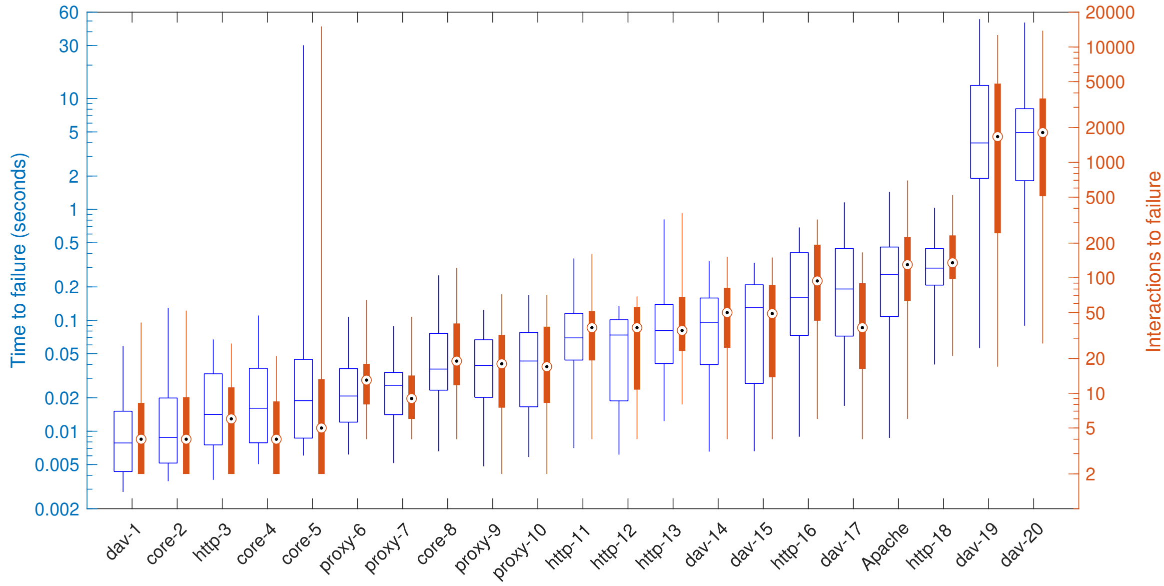


FIGURE 5.1. Cost of detecting bug in each server/mutant. The left box with median line is the tester's execution time before rejecting the server, which includes interacting with the server and checking its responses. The right bar with median circle is the number of HTTP/1.1 messages sent and received by the tester before finding the bug. Results beyond 25%–75% are covered by whiskers.

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 [7]. These results show that our tester is capable of finding bugs in server implementations, including those we're unaware of.

5.2.2. Performance. As shown in Figure 5.1, 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

[LYS: Duplicates with Figure 1.3 and Figure 1.4.]

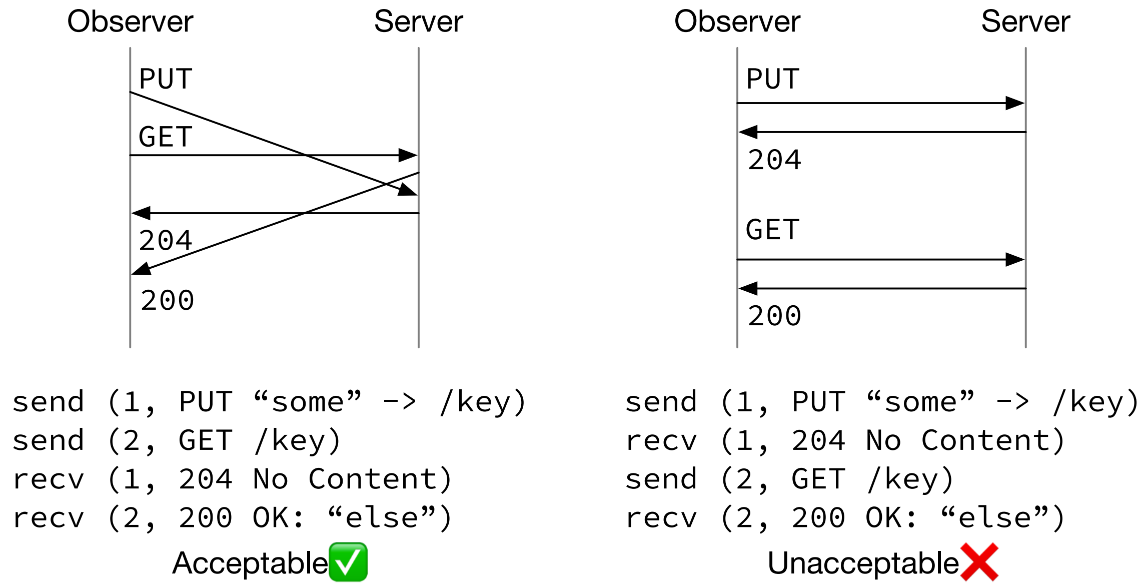


FIGURE 5.2. The trace on the left does not convince the tester that the server is buggy, because there exists a certain network delay that explains why the PUT request was not reflected in the 200 response. When the trace is ordered as shown on the right, the tester cannot imagine any network reordering that causes such observation, thus must reject the server.

with full confidence, these observations must be made in a certain order, as shown in Figure 5.2.

- Mutant 18 is similar to the bug in vanilla Apache: the server should have responded with 304 Not Modified, but sent back 200 OK instead. To reveal such violation, a minimal counterexample consists of 4 messages: (1) GET request, (2) 200 OK response with some ETag `x`, (3) GET request conditioned over `If-None-Match: x`, and (4) 200 OK response, indicating that the ETag `x` did not match itself. Notice that (2) must be observed before (3), otherwise the tester will not reject the server, with a similar reason as Figure 5.2.
- 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.

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.

CHAPTER 6

Related Work

6.1. 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 [14] implemented a test generation tool for symbolic transition systems called TorXakis, which has been used for testing Dropbox [14].

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.

6.2. Reasoning about Network Delays

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

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

CHAPTER 7

Discussions

CHAPTER 8

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] 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.

[4] 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.

[5] 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.

[6] 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.

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

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

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

[10] 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.

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

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

[13] Jan Tretmans. "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.

[14] 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.

[15] 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.

[16] 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

Mathematical Proof of Derived Validators' Correctness

A.1. Forward preservation lemma for rejection soundness

Hypothesis RejSound-Step

$$\begin{aligned}
&\forall (p : \mathsf{Prog})(q, c, a : \mathbb{Z})(s_0, s' : \sigma)(v : \beta),\\
&\mathsf{sstep}_p(q, c, s_0) = (a, s') \wedge v \sim s_0\\
&\implies \exists v' : \beta, \mathsf{vstep}_p(q, a, v) = \mathsf{Some}\ v' \wedge v' \sim s'
\end{aligned}$$

The invariant $v \sim s_0$ tells us that v contains a validation state that reflects the server state s_0:

$$\exists ((vs_0, cs_0) \in v)(asgn_0 : \mathsf{var} \to \mathbb{Z}), asgn_0\ \mathsf{satisfy}\ cs_0 \wedge {vs_0}^{asgn_0} \equiv s_0$$

The corresponding validator step is constructed by analyzing the server step, and proving small-step bisimulation for each derivation rule in Section 2.3.

Write. The server writes some expression ($e : \mathsf{SExp}$) to an address $!dst$. According to Rule 1, the validator creates a fresh variable x_e for address $!dst$, and constraints that $(\#x_e \equiv e^{vs})$.[6]

We need to show that:

$$\begin{aligned}
&\forall vs, cs, s, asgn,\\
&asgn\ \mathsf{satisfy}\ cs \wedge vs^{asgn} \equiv s\\
&\implies \forall d, e,\ \begin{array}{lll} \mathsf{let}\ s' & = s[d \mapsto e^s] & \mathsf{in}\\ \mathsf{let}\ (vs', cs') & = \mathsf{write}(d, e, (vs, cs)) & \mathsf{in}\\ \exists asgn', & asgn'\ \mathsf{satisfy}\ cs' \wedge vs'^{\ asgn'} \equiv s' & \end{array}
\end{aligned}$$

Proof. Based on the definition of write, we need to show that:

$$\begin{aligned}
&\mathsf{let}\ x_e = \mathsf{fresh}\ (vs, cs)\ \mathsf{in}\\
&\mathsf{let}\ vs' = vs[d \mapsto x_e]\ \mathsf{in}\\
&\mathsf{let}\ cs' = cs \cup \{x_e \equiv e^{vs}\}\ \mathsf{in}\\
&\exists asgn', \quad asgn'\ \mathsf{satisfy}\ cs' \wedge vs'^{\ asgn'} \equiv s'
\end{aligned}$$

Let:

$$asgn' = asgn[x_e \mapsto e^s]$$

In order to prove ($asgn'$ $\mathsf{satisfy}$ cs'), we introduce some generic lemmas to show that ($asgn'$ $\mathsf{satisfy}$ cs) and (${x_e}^{asgn'} \equiv (e^{vs})^{asgn'}$):

[6]If unspecified, (vs, cs) represents the pre-small-step validator state, and s represents the pre-small-step server state.

Lemma A.1 (Fresh variable preserves satisfaction).

$$
\begin{array}{l}
\forall(cs : \textsf{set constraint})(asgn : \textsf{var} \to \mathbb{Z}),\\
asgn\ \textsf{satisfy}\ cs\\
\quad \Longrightarrow \quad \begin{array}{lll}
\multicolumn{3}{l}{\forall(z : \mathbb{Z}), vs,}\\
\textsf{let}\ x & = \textsf{fresh}\ (vs, cs) & \textsf{in}\\
\textsf{let}\ asgn' & = asgn[x \mapsto z] & \textsf{in}\\
\multicolumn{3}{l}{asgn'\ \textsf{satisfy}\ cs}
\end{array}
\end{array}
$$

Proof. Since x is fresh in cs, we have:

$$\forall(e_1\ cmp\ e_2) \in cs, \quad {e_1}^{asgn'} = {e_1}^{asgn} \wedge {e_2}^{asgn'} = {e_2}^{asgn}$$

Thus:

$$
\begin{array}{l}
\forall(e_1\ cmp\ e_2) \in cs, \quad {e_1}^{asgn'}\ cmp\ {e_2}^{asgn'}\\
i.e.\ \ asgn'\ \textsf{satisfy}\ cs
\end{array}
$$

□

Lemma A.2 (Fresh variable preserves evaluation).

$$
\begin{array}{lll}
\multicolumn{3}{l}{\forall(vs : \mathbb{N} \to \textsf{var})(asgn : \textsf{var} \to \mathbb{Z})(z : \mathbb{Z}), cs,}\\
\textsf{let}\ x & = \textsf{fresh}\ (vs, cs) & \textsf{in}\\
\textsf{let}\ asgn' & = asgn[x \mapsto z] & \textsf{in}\\
\multicolumn{3}{l}{\forall(e : \textsf{SExp}), (e^{vs})^{asgn'} = (e^{vs})^{asgn}}
\end{array}
$$

Proof. Assume to the contrary that:

$$(e^{vs})^{asgn'} \neq (e^{vs})^{asgn}$$

Since $asgn'$ is the same as $asgn$ except for variable $\#x$, we know that $(e^{vs} : \textsf{VExp})$ must involve $\#x$. Therefore, e must involve some address $!k$ such that $(vs!k = x)$. This contradicts the fact that x is fresh in vs. □

Lemma A.3 (Symbolization preserves evaluation).

$$
\begin{array}{c}
\forall(vs : \mathbb{N} \to \textsf{var})(asgn : \textsf{var} \to \mathbb{Z})(s : \mathbb{N} \to \mathbb{Z}),\\
vs^{asgn} \equiv s \implies \forall(e : \textsf{SExp}), \quad (e^{vs})^{asgn} \equiv e^{s}
\end{array}
$$

Proof. Based on the definition of symbolization and evaluation:

- e^{vs} substitutes all occurences of $!k$ in e with $\#(vs!k)$;
- $(e^{vs})^{asgn}$ substitutes all occurences of $\#(vs!k)$ to $asgn!(vs!k)$;
- e^{s} substitutes all occurences of $!k$ in e with $(s!k)$.

From the hypothesis that $(vs^{asgn} \equiv s)$, we have:

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

Therefore, we know that all occurences of $!k$ were mapped to the same value between the two evaluation paths. □

Based on Lemma A.2, we have:

$$(e^{vs})^{asgn'} = (e^{vs})^{asgn}$$

Also, since x_e is free in vs, and $asgn'$ is the same as $asgn$ except for x_e, we have:

$$\forall k, asgn'!(vs'!k) = \begin{cases} asgn'!x_e = e^s = (s'!k) & k \textsf{ is } d \\ asgn!(vs!k) = (s!k) = (s'!k) & \text{otherwise} \end{cases}$$

Therefore:

$$vs'^{\ asgn'} \equiv s'$$

□

Havoc. When the server writes some internal choice c to address d, according to Rule 3, the validator creates a fresh variable for address $!d$.

We need to show that:

$$\begin{aligned}
&\forall vs, cs, s, asgn, \\
&asgn \textsf{ satisfy } cs \land vs^{asgn'} \equiv s \\
&\implies \forall d, c, \ \begin{array}{lll} \textsf{let } s' & = s[d \mapsto c] & \textsf{in} \\ \textsf{let } x_c & = \textsf{fresh } (vs, cs) & \textsf{in} \\ \multicolumn{3}{l}{\exists asgn', \quad asgn' \textsf{ satisfy } cs \land vs^{asgn'} \equiv s'} \end{array}
\end{aligned}$$

Proof. Let:

$$asgn' = asgn[x_c \mapsto c]$$

Since x_c is free in cs, we have □

A.2. Backward preservation lemma for rejection completeness

Hypothesis RejComplete-Step

APPENDIX B

Unstructured contents

B.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 [4] and message forwarding [5]—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."

B.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 [15].

Subsection B.2.1 shows how to handle network nondeterminism. Subsection 3.3.1 then expands the model to address internal nondeterminism.

B.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. [9].

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 3.5 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 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 3.2.2, we compose the server and network models to yield an observer-side specification for testing purposes.

B.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 **??** 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 **??**.

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

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

B.3.1. Interpreting Interaction Trees.

B.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 3.1, 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 3.2.2, addressing network nondeterminism.

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 B.1, our derivation framework allows passing the programs' internal state as the events' parameters, so the test case generator can utilize the

```
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 B.1. Embedding programs' internal state into the events. By expanding the events' parameters, we enrich the test case generator's knowledge along the interpretations.

states in all intermediate interpretation phases, and apply heuristics to emphasise certain bug patterns.

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 Section 5.2. 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 B.2. 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 **??**, the packet is appended to the outgoing buffer `bo` until absorbed by `net` in Line 9, and eventually emitted to the client in Line 16. Conversely, packets sent by clients are absorbed by `net` in Line 10, emitted to the application's incoming buffer `bi` in **??**, until `srv` consumes it in **??**.

B.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 B.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 3.6, 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 3.5 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 27. When the network wants to `recv` some packet in **??**, 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 3.5, 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.