Language Design for Asynchronous Test

Thesis Proposal

Yishuai Li

January 30, 2022

1 Introduction

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

Rigorous testing requires a rigorous specification of the protocol that we expect the server to obey. Protocol specifications can be written as (i) a *server model* that describes *how* valid servers should handle messages, or (ii) a *property* that defines *what* server behaviors are valid. From these specifications, we can conduct (i) *model-based testing* [6] or (ii) *property-based testing* [9], respectively.

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

(1) *Within* the server, correct behavior may be underspecified. For example, to handle HTTP conditional requests [8], a server generates strings called entity tags (ETags), but the RFC specification does not limit what values these ETags should be. Thus, to create test messages containing ETags, the tester must remember and reuse the ETags it has been given in previous messages from the server.

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

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

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

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

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

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

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

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

This claim will be supported by the following publications:

1. *From C to Interaction Trees: Specifying, Verifying, and Testing a Networked Server* [13], with Nicolas Koh, Yao Li, Li-yao Xia, Lennart Beringer, Wolf Honoré, and William Mansky, where I developed a tester program based on the swap server's ITree specification, and evaluated the tester's effectiveness by mutation testing.

2. *Verifying an HTTP Key-Value Server with Interaction Trees and VST* [20], 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* [15], 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. A theory for model-based asynchronous testing, explaining how to specify protocols using abstract model implementations, and how to guarantee the soundness and completeness of the validator logic derived from the abstract model.

5. An experience report on test harness design, introducing an intermediate representation for specifying test input generation heuristics, and a generic framework for running the tests and shrinking the counterexamples.

This proposal is structured as follows: Section 2 describes the challenges caused by internal and network nondeterminism. Section 3 lists related works in testing network protocols. Section 4 and 5 introduce my previous experiments and developed theories. Section 6 and 7 discuss my plan towards finishing this thesis, with technical details in Appendix A.

2 Challenges

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

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

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

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

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

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

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

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

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

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

Figure 1: Ad hoc tester for HTTP/1.1 conditional requests, demonstrating how tricky it is to write the logic by hand. The checker determines whether a one-client-at-a-time `trace` is valid or not. The trace is represented as a stream (infinite linked list, constructed by "`::`") of HTTP messages sent and received. `PUT(k,t,v)` represents a PUT request that changes `k`'s value into `v` only if its ETag matches `t`; `GET(k,t)` is a GET request for `k`'s value only if its ETag does not match `t`; `OK(t,v)` indicates the request target's value is `v` and its ETag is `t`. The tester maintains three sorts of knowledge about the server: `data` stored for each content, what some ETag `is` known to be equal to, and what some ETag `is_not` equal to.

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 shows a hand-written tester for checking this bit of ETag functionality; we hope the reader will agree that this testing logic is not straightforward to derive from the informal "server's eye" specifications.

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

3 Related Work

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.

Language of Temporal Ordering Specification (LOTOS) [5] 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 [18] implemented a test generation tool for symbolic transition systems called TorXakis, which has been used for testing Dropbox [18].

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. [4] 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.

Reasoning about Network Delays For property-based testing against distributed applications like Dropbox, Hughes et al. [12] have introduced "conjectured events" to represent uploading and downloading events that nodes may perform at any time invisibly.

Sun et al. [16] 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.

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

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

Definition trace E := list (event E)

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

Figure 2: Interaction trees and their traces of events.

4 Prior Practices in Asynchronous Testing

4.1 Specification Languages

Property-based specification with QuickChick My first formal specification of HTTP/1.1 was written as QuickChick [14] properties, which takes a trace of requests, and determines whether the traces is valid per protocol specification, like that shown in Figure 1. The specification implemented a constraint solving logic by hand, making it hard to scale when the protocol becomes more complex, as discussed in Section 2

Model-based specification with ITrees To write specifications for protocols' rich semantics, I employed "interaction tree" (ITree), a generic data structure for representing interactive programs, introduced by Xia et al. [19]. ITree enables specifying protocols as monadic programs that model valid implementations' possible behavior. The model program can be interpreted into a tester program, to be discussed in Subsection 4.2.

Figure 2 defines the type `itree E R`. The definition is *coinductive*, so that it can represent potentially infinite sequences of interactions, as well as divergent behaviors. The parameter `E` is a type of *external interactions*—it defines the interface by which a computation interacts with its environment. `R` is the *result* of the computation: if the computation halts, it returns a value of type `R`.

There are three ways to construct an ITree. The `Ret r` constructor corresponds to the trivial computation that halts and yields the value `r`. The `Tau t` constructor corresponds to a silent step of computation, which does something internal that does not produce any visible effect and then continues as `t`. Representing silent steps explicitly with `Tau` allows us, for example, to represent diverging computation without violating Coq's guardedness condition [7]:

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

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

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

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

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

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

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

Figure 3: Linear specification of the swap server. In the `linear_spec'` loop, the parameter `conns` maintains the list of open connections, while `last_msg` holds the message received from the last client (which will be sent back to the next client). The server repeatedly chooses between accepting a new connection or doing a receive and then a send on some existing connection picked in the list `conns`. The linear specification is initialized with an empty set of connections and a message filled with zeros.

4.2 From Specification to Test

From an ITree specification, I conducted "offline" testing, which takes a trace and determines its validity [13], and "online" testing, where the specification is derived into a client program that validates the system under test interactively [15].

Offline testing of swap server I started with testing a simple "swap server" [13], specified in Figure 3. The specification says that the server can either accept a connection with a new client (`obs_connect`) or else receive a message from a client over some established connection (`obs_msg_to_server c`), send back the current stored message (`obs_msg_from_server c last_msg`), and then start over with the last received message as the current state.

To test this swap server, I wrote a client program that interacts with the server and produces a trace of requests and responses, and a function that determines whether the trace t is a trace of the linear specification s *i.e.* whether `is_trace s t` in Figure 2 holds.

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

Online testing of HTTP To test protocols with internal nondeterminism (*e.g.* HTTP) effectively, I introduced a symbolic representation for the server's invisible choices, as shown in Figure 4. I then defined a TCP network model in Figure 5. Combining the server and network models produces a model program that exhibits all valid observations, considering both internal and network nondeterminism.

From the server and network models, I derived a tester client that interacts with servers over the network, and validates the observations against the protocol specification, as shown in Figure 6.

Using this automatically derived tester program, I have found violations against HTTP/1.1 in the latest version of both Apache and Nginx. More details are explained in Li et al. [15].

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

5 Theories Developed for Interactive Testing

During the testing practice in Section 4, the tester's quality was evaluated by mutation testing, *i.e.* running the tester against buggy implementations to see if it rejects. To formally prove that the tester is good, I develop a theory for reasoning on testers' good properties.

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

Figure 4: Symbolic model handling conditional PUT request. The model maintains two states: `data` that maps keys to their values, and `xtag` that maps keys to symbolic variables that represent their corresponding ETags. Upon receiving a PUT request conditioned over "If-Match: `t`", the server should decide whether the request ETag `matches` that stored in the server. Upon matching, the server processes the PUT request, and represents the updated value's ETag as a fresh variable.

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

Figure 5: Network model for concurrent TCP connections. The model maintains a `buffer` of all packets en route. In each cycle, the model may nondeterministically branch to either absorb `or` emit a packet. Any absorbed packet is appended to the end of buffer. When emitting a packet, the model may choose a connection and send the oldest packet in it.

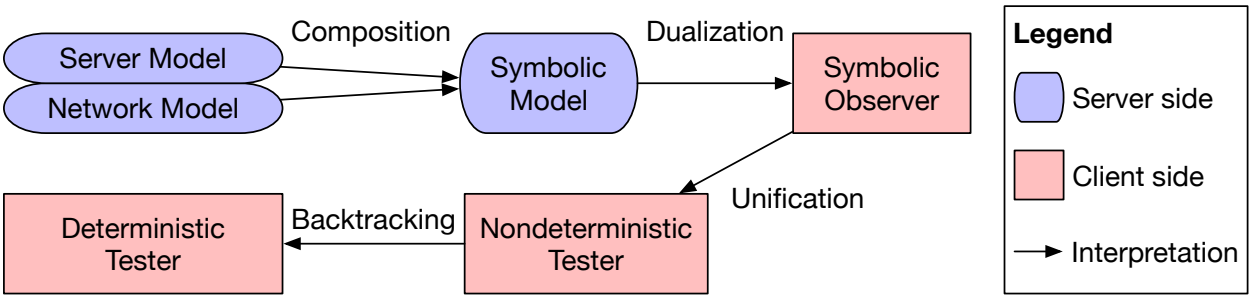


Figure 6: Deriving tester program from specification

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

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

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

My goal is to formalise the techniques in Li et al. [15] into a generic theory for asynchronous testing.

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

The test harness needs to produce counterexamples effectively, and provide good coverage of test cases. The goal is to locate unknown bugs within a fixed budget, which is more practical than theoretical, and will be discussed in Subsection 6.2. The test theory in this dissertation focuses on guaranteeing the soundness and completeness of the validator logic.

5.1 Synchronous Testing Language

Before facing the complexity in asynchronous testing, I start by formalising synchronous testing theory, which describes how to build a sound and exhaustive tester, and will expand the theory to asynchronous testing as described in Subsection 6.1.

Server specifications and validator implementations Synchronous testing can be viewed as a *server* interacting with a single client, producing a *trace* of queries and responses, and a validator determing whether the trace is acceptable or not.

Definition 1 (Server model)**.** Let Q be the query type, A be the response type, S be some server state, then a deterministic server starts from an initial state, takes a query, computes the response based on its current state, computes the next server state, and loops over:

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

This definition is pronounced "A deterministic server has some type S that represents its internal state. It is specified by a step function of type $Q \times S \to A \times S$, and an initial state of type S."

In general, servers have *internal nondeterminism* that allows the responses and transitions to depend on internal choices that are invisible to the testers, *e.g.* timestamps and hash algorithms. Let C be the space of internal choices, then a nondeterministic server is specified as:

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

Here $\mathbf{pack}\ S = \sigma\ \mathbf{with}\ (\mathbf{sstep}, state)$ specifies a server whose internal state type is σ, and has step function $(\mathbf{sstep} : Q \times C \times \sigma \to A \times \sigma)$ and initial state $(state : \sigma)$.

Definition 2 (Validator)**.** Let V be some validator state, then a validator starts from an initial state, takes a query and its corresponding response, determines whether these interactions are valid, and computes the next validator state upon valid:

$$\begin{aligned}
&\mathbf{Validator} \triangleq \{\exists V, (Q \times A \times V \to \mathbf{option}\ V) \times V\} \\
&\mathbf{stepValidator} : Q \times A \times \mathbf{Validator} \to \mathbf{option\ Validator} \\
&\mathbf{stepValidator}(q, a, \mathbf{pack}\ V = \beta\ \mathbf{with}\ (\mathbf{vstep}, state)) \\
&\quad \triangleq \begin{cases} \mathbf{Some}\ (\mathbf{pack}\ V = \beta\ \mathbf{with}\ (\mathbf{vstep}, state')) & \mathbf{vstep}(q, a, state) = \mathbf{Some}\ state' \\ \varnothing & \mathbf{vstep}(q, a, state) = \varnothing \end{cases}
\end{aligned}$$

Deriving validator from server definition While it is possible to write server specifications and validators manually, Li et al. [15] have shown that internal nonderminism makes the validator difficult to implement and reason about, and demonstrated how to automatically derive a validator from a server specification. That paper has evaluated the derived validator's effectiveness by finding bugs in real world programs, but did not provide a mathematical proof. To construct a generic derivation process, I start with a simplified derivation model for synchronous tests.

Definition 3 (Server and validator of a program)**.** A program $p \in \mathbf{Prog}$ is a representation that can be "instantiated" into a server model:

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

A program can also be "interpreted" into a validator:

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

Soundness and completeness A tester is *sound* if any valid implementation passes it *i.e.* no false rejection. A tester is *exhaustive* if any invalid implementation fails it (given sufficient time) *i.e.* no false acceptance. [1]

A sound tester requires a sound validator, and a faithful test harness that correctly interprets its observed behavior into the validator's input; An exhaustive tester requires a *complete* validator, a faithful test harness, plus an exhaustive test input generator that reveals all possible behavior of the SUT. Here I focus on the soundness and completeness of the validator.

Definition 4 (Trace validity)**.** A trace is valid if there exists an execution of the server model that *yields* it. Let the trace be a list of query-response pairs ($\mathbf{list}\ (Q \times A)$), then:

1. Any server can step to itself, yielding an empty trace:

$$\forall s \in \mathbf{Server}, s \xrightarrow{\varepsilon} s$$

2. A server yields a non-empty trace if it can yield the head of the trace and step into a server that yields the tail of the trace:

$$\begin{aligned}&\forall s, s_2 \in \mathbf{Server}, \forall l \in \mathbf{list}\ (Q \times A), \forall q \in Q, \forall a \in A,\\ &(\exists s_1 \in \mathbf{Server}, \exists c \in C \mid s \xrightarrow{l} s_1 \wedge \mathbf{stepServer}(q, c, s_1) = (a, s_2))\\ &\implies s \xrightarrow{l+(q,a)} s_2\end{aligned}$$

Definition 5 (Trace acceptance)**.** A trace is *accepted* by a validator if the validator consumes the entire trace and steps into a nonempty state:

1. Any validator accepts an empty trace and steps to itself:

$$\forall v \in \mathbf{Validator}, v \xrightarrow{\varepsilon} v$$

2. A validator accepts a non-empty trace if it can consume the head of the trace and step into a validator that accepts the end of the trace:

$$\begin{aligned}&\forall v, v_2 \in \mathbf{Validator}, \forall l \in \mathbf{list}\ (Q \times A), \forall q \in Q, \forall a \in A,\\ &(\exists v_1 \in \mathbf{Validator} \mid v \xrightarrow{l} v_1 \wedge \mathbf{stepValidator}(q, a, v_1) = \mathbf{Some}\ v_2)\\ &\implies v \xrightarrow{l+(q,a)} v_2\end{aligned}$$

Definition 6 (Soundness)**.** A validator is *sound* per server specification if it accepts all valid traces of the specification:

$$v\ \mathbf{sound}_s \triangleq \forall l \in \mathbf{list}\ (Q \times A), (\exists s' \in \mathbf{Server} \mid s \xrightarrow{l} s') \implies \exists v' \in \mathbf{Validator} \mid v \xrightarrow{l} v'$$

Definition 7 (Completeness)**.** A validator is *complete* per server specification if it only accepts valid traces of the specification:

$$v\ \mathbf{complete}_s \triangleq \forall l \in \mathbf{list}\ (Q \times A), (\exists v' \in \mathbf{Validator} \mid v \xrightarrow{l} v') \implies \exists s' \in \mathbf{Server} \mid s \xrightarrow{l} s'$$

[1] Concepts like "sound", "complete", "false positive", and "false negative" have opposite meanings in different contexts. To avoid further confusion, I'll use "reject" and "accept" in place of "positive" and "negative". The soundness and completeness definitions here is applied from Tretmans [17].

5.2 Reasoning on Synchronous Testers

Given a server **pack** $S = S$ **with** $(\textbf{sstep}, s_0)$ and a validator **pack** $V = V$ **with** $(\textbf{vstep}, v_0)$, to prove its soundness and completeness, we need to show properties over the step functions and the initial states.

The validator's soundness says: For any trace producable by the server specification, that trace can be accepted by the validator. The completeness says: For any trace accepted by the validator, there exists an execution of the server specification that yields this trace.

Since the server and validator are both loops, I introduce an invariant $\{v \sim s \mid v \in V, s \in S\}$, which is preserved by the loops' step functions.

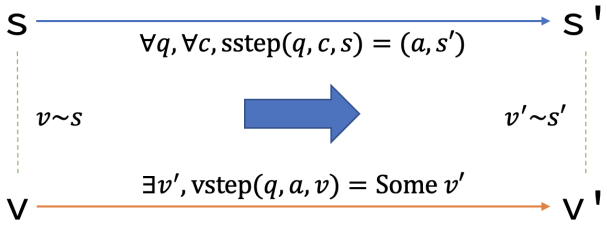
Proving soundness The soundness proof is by forward induction of the server execution path, and construct the corresponding validation path based on the following assumptions:

1. The initial server state reflects the initial validator state:

$$v_0 \sim s_0$$

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

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



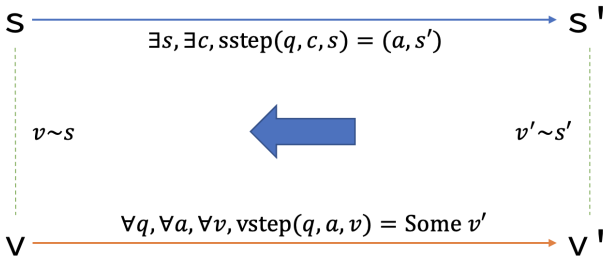
Proving completeness The completeness proof is by backward induction of the validation path, and construct the corresponding server execution path based on the following assumptions:

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

$$\begin{aligned}\forall q \in Q, \forall a \in A, \forall v, v' \in V, \textbf{vstep}(q, a, v) = \textbf{Some } v'\\ \implies \exists s' \in S, v' \sim s'\end{aligned}$$

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

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



3. The initial validator state only reflects the initial server state:

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

5.3 Case Study: Single Instruction Server

In this experiment, I target a simple server language where the program is a single instruction. I'll show how to instantiate the program into a server model, and how derive the program into a validator that is sound and complete. That is, defining **serverOf** and **validatorOf** functions for **Prog** such that:

$$\begin{aligned}\forall p : \mathbf{Prog}, \mathbf{let}\ s = \mathbf{serverOf}(p)\ \mathbf{in}\\ \mathbf{let}\ v = \mathbf{validatorOf}(p)\ \mathbf{in}\\ v\ \mathbf{sound}_s \wedge v\ \mathbf{complete}_s\end{aligned}$$

Server language The server stores a key-value mapping $K \to \mathbb{N}$. The server program is a three-address instruction that performs an arithmetic operation:

$$\mathbf{Prog} \triangleq \{!dst \leftarrow !src_1\ op\ !src_2 \mid op \in \{+, -, \times, \div\}; dst, src_1, src_2 \in K\}$$

A program in this language is instantiated into a server step as follows: Upon receiving a query $q \in Q$, the server writes a choice $c \in C$ to key 1, and writes the query to key 0. The server then runs the instruction, and sends back the value stored in key 0:

$$\begin{aligned}&Q \triangleq \mathbb{N} \qquad C \triangleq \mathbb{N} \qquad A \triangleq \mathbb{N}\\ &\mathbf{serverOf}(p) \triangleq \mathbf{pack}\ S = (K \to \mathbb{N})\ \mathbf{with}\ (\mathbf{sstep}(p), (\_ \mapsto 0))\\ &\text{where}\\ &\mathbf{sstep}(!dst \leftarrow !src_1\ op\ !src_2)(q, c, s_0) \triangleq\\ &\quad \mathbf{let}\ s_1 = s_0[k_1 \mapsto c]\ \mathbf{in}\\ &\quad \mathbf{let}\ s_2 = s_1[k_0 \mapsto q]\ \mathbf{in}\\ &\quad \mathbf{let}\ s_3 = s_2[dst \mapsto s_2(src_1)\ op\ s_2(src_2)]\ \mathbf{in}\\ &\quad (s_3(k_0), s_3)\end{aligned}$$

To check a trace against this server model, the validator introduces symbolic variables $x \in X$ to represent the values stored in each key. The validator also maintains a set of constraints that reflect the server's computations:

$$
\begin{aligned}
&E \triangleq \mathbb{N} \cup \{!x \mid x \in X\} \cup \{!x_1 \ op \ !x_2 \mid op \in \{+, -, \times, \div\}; x_1, x_2 \in X\} \\
&\textbf{Constraint} \triangleq \{e_1 = e_2 \mid e_1, e_2 \in E\} \\
&\beta \triangleq (K \to X) \times \textbf{list Constraint} \\
&\textbf{validatorOf}(p) \triangleq \textbf{pack } V = \beta \textbf{ with } (\textbf{vstep}(p), ((\_ \mapsto x_0), (0 = !x_0))) \\
&\text{where} \\
&\textbf{vstep}(p)(q, a, v_0) \triangleq \\
&\quad \textbf{let } v_1 \quad = \textbf{havoc}(k_1, v_0) \quad \textbf{in} \\
&\quad \textbf{let } v_2 \quad = \textbf{write}(k_0, q, v_1) \quad \textbf{in} \\
&\quad \textbf{let } (vs_3, cs_3) \quad = \textbf{exec}(p, v_2) \quad \textbf{in} \\
&\quad \textbf{let } cs_4 \quad = cs_3 + (!vs_3(k_0) = a) \quad \textbf{in} \\
&\quad \textbf{if solvable}(cs_4) \textbf{ then Some } (vs_3, cs_4) \textbf{ else } \varnothing \\
&\textbf{write}(k, q, (vs, cs)) \triangleq \\
&\quad \textbf{let } x = \textbf{fresh}(vs, cs) \textbf{ in} \\
&\quad (vs[k \mapsto x], cs + (q = !x)) \\
&\textbf{havoc}(k, (vs, cs)) \triangleq \\
&\quad \textbf{let } x = \textbf{fresh}(vs, cs) \textbf{ in} \\
&\quad (vs[k \mapsto x], cs) \\
&\textbf{exec}((!dst \leftarrow !src_1 \ op \ !src_2), (vs, cs)) \triangleq \\
&\quad \textbf{let } x = \ \textbf{fresh}(vs, cs) \textbf{ in} \\
&\quad (vs[dst \mapsto x], cs + (!vs(src_1) \ op \ !vs(src_2) = !x)) \\
&\textbf{fresh} : \beta \to X \\
&\textbf{solvable} : \textbf{list Constraint} \to \mathbb{B}
\end{aligned}
$$

When the **sstep** function modifies the server state, the **vstep** function updates the validator's view of the server correspondingly: When the server stores a query to key 0, the **write** function maps key 0 to a **fresh** variable, and adds a constraint that the variable's value is equal to the query. When the server stores an internal choice to key 1, the **havoc** function maps key 1 to a fresh variable with no constraints, because its exact value is invisible to the validator. When the server runs a three-address instruction, the **exec** function maps the destination to a fresh variable, and adds a constraint that the variable's value should reflect the three-address computation.

After updating its symbolic view of the server, the validator expects the observed response to be explainable by its symbolic view. Such expectation is checked by a constraint solver, that determines whether the constraints can be satisfied by some assignment of the variables:

$$
\begin{aligned}
\textbf{Assignment} &\triangleq X \to \mathbb{N} \\
\textbf{evaluate} &: \textbf{Assignment} \times E \to \mathbb{N} \\
\textbf{satisfy}(asgn, cs) &\triangleq \forall (e_1 = e_2) \in cs, \\
&\qquad \textbf{evaluate}(asgn, e_1) = \textbf{evaluate}(asgn, e_2) \\
\forall cs \in \textbf{list Constraint}, &\ \textbf{solvable}(cs) \\
&\iff \exists \ asgn \in \textbf{Assignment}, \textbf{satisfy}(asgn, cs)
\end{aligned}
$$

To prove the validator's completeness, the reflection relation is defined as:

$$
\begin{aligned}
((vs, cs) \sim s) \triangleq \exists \ asgn, &\ \textbf{satisfy}(asgn, cs) \\
&\wedge \forall k \in K, asgn(vs(k)) = s(k)
\end{aligned}
$$

This reflection definition is sufficient for proving the validator's completeness, as it satisfies all the assumptions in Subsection 5.2:

1. Any accepting validator step has solved the post-validation constraints cs', which implies an assignment $asgn$ that satisfies cs'. Let vs' be the post-validation key-variable mapping, then $s' = asgn \circ vs'$ reflects the post-validation state (vs', cs') by definition.

2. The **vstep** function monotonically increases the list of constraints, and thus monotonically narrows the space of satisfying assignments. Let s' be the post-execution server state that reflects the post-validation state (vs', cs'), then $asgn$ is guaranteed to satisfy the pre-validation constraints cs. Therefore, by analysing the execution of $\mathbf{vstep}(q, a, (vs, cs))$, we can construct the pre-execution server state $s = asgn \circ vs$ and the server's internal choice $c = asgn(\mathbf{fresh}(vs, cs))$ that satisfy $\mathbf{sstep}(q, c, s) = (a, s')$ and $(vs, cs) \sim s$.

3. From the definition of the server and validator's initial state, we can show that the initial server state is the only state that reflects the initial validator state.

The soundness proof is pending, but I've proven the soundness of validators for a simpler server language.

6 Research Plan

6.1 Theories of Interactive Testing

Synchronous Testing in General The server language shown in Subsection 5.3 consists of a single instruction. I plan to extend that language with sequential combinators and If-branches, for modelling more realistic server programs. For example:

```
Inductive Prog: Type :=
  Singleton (i: Instruction)
| Sequence  (p1 p2: Prog)
| Branch    (condition: Expression) (thenP elseP: Prog).
```

This language covers a subset of C programs, excluding loops with indefinite iterations (which is not common in server programs' control flow). It can also model various implementation choices, using `Branch c p1 p2` where the condition is an unbound variable.

Asynchronous Testing The server model in Subsection 5.2 has separated the computations from the interactions. To model asynchronous interactions of the server, I plan to merge the send/receive interactions into the server language, between the lines of computations. For example:

```
Inductive Prog  : Type :=
  Recv (handler : Message → Prog)
| Send (response: Message)
| Singleton ...
| Sequence ...
| Branch ...
| ...
```

6.2 Test Input Generation and Shrinking Techniques

To make a sound and exhaustive tester useful in software engineering practices, we need to (1) generate test inputs that can reveal potential bugs quickly, and (2) report minimal counterexamples to debuggers, rather than thousand-line logs.

One challenge for random testing is that, to find failing examples efficiently, it is important to generate fairly large test cases. But then, after a failing test case has been found, it is important to *minimize* the test input to eliminate the parts that are not relevant to provoking the failure. Tools like QuickCheck therefore provide automatic methods for minimizing, or *shrinking*, failing tests, by incrementally generating smaller variants until they reach a local minimum—a counterexample that cannot be reduced any further.

Traditional shrinking doesn't work well for impure programs, which might mutate state or interact with their environments, because an interesting test input in one execution might become trivial or irrelevant in another run. In particular, when testing an interactive program, interesting test inputs might depend on existing observations of the program's behavior.

To generate and shrink inputs that are interesting among different executions, Hughes [11] introduced a method that relates inputs to runtime observations. In this framework, the tester interacts with the system

under test (SUT) via function calls, and maintains a runtime state that is updated after each call returns. Test inputs are represented in an abstract language that depends on the runtime state *e.g.* "call function `foo(x)`, where `x` is the return value of the 10th function call". Such abstract representation is instantiated into a concrete input during runtime, by remembering the 10th call's return value `a` to build the actual function call `foo(a)`.

This methodology has achieved good results in testing state machines [10], but the specification language requires the interactions to be synchronous *i.e.* to return immediately upon call. However, when testing networked systems, requests and responses might be delayed or reordered *en route*. Such a scenario requires a different specification language that allows asynchronous interactions [15], introducing new challenges in minimizing test inputs, *e.g.* when a dependent response has not arrived.

I'll propose a generic method for generating and minimizing asynchronous test interactions, by combining the idea of abstract input representation from Hughes [11] with the asynchronous specification language of Li et al. [15]. The tester records a *trace* of network packets sent and received, and uses the trace to instantiate abstract inputs into concrete requests to send. The key inovation is an intermediate representation (IR) between structured application messages and bytes. The IR allows the generated abstract input to refer to specific fields in the trace, and can handle situations where the referred field is absent (caused by internal or network nondeterminism).

A preliminary design is shown in Appendix A.

7 Evaluation Criteria

As mentioned in Section 1, this thesis is to propose language designs for (i) validation logic and (ii) test case generation, and provide an asynchronous testing methodology with (i) theoretical guarantee of soundness and completeness, and (ii) practicality in testing real-world applications.

7.1 Practices: Testing More Complex Network Applications

To demonstrate the specification language's expressiveness and the derived tester's effectiveness, I'll apply my test framework to a web application on top of HTTP/1.1.

Protocol specification I'll develop a simple exchange where users can trade assets by placing orders. The application specification for the exchange's functionality will be composed with the underlying HTTP/1.1 specification, to show that specifications written in my language are reusable and can be developed modularly.

Test case generation The exchange server will communicate with clients using JSON, a common message format for web applications. My tester will generate requests using flexible heuristics, that can refer to fields in JSON's data strucutre which can be arbitrarily deep.

Research questions

1. For testing nontrivial protocols, is the specification easy to write and easy to understand?
2. Can the derived tester effectively produce minimal counterexamples and help debugging server implementations?

7.2 Theories: Verified Testing Framework

To show the methodology of formally reasoning of derived testers, I'll extend the server language **Prog** as proposed in Subsection 6.1, such that a nontrivial application can be written in this language. I'll then implement the (**validatorOf** : **Prog** $\rightarrow$ **Validator**) algorithm, and prove its soundness and completeness.

The experiments with HTTP/1.1 are conducted with ITrees, whereas the **Prog** is a tree of imperative instructions. While bridging the language gap in between is beyond the scope of this thesis, I'd expect the reasoning methods here can inspire test developers of other languages.

Research questions

1. As the **Prog** language extends, how to adjust the verification framework to prove validators' soundness and completeness?

2. Can the proof technique for **Prog**-based validators be applied to other specification languages, *e.g.* interaction trees?

References

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

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

[3] Thomas Arts et al. "Testing Telecoms Software with Quviq QuickCheck". In: *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*. ERLANG '06. Portland, Oregon, USA: Association for Computing Machinery, 2006, 2–10. ISBN: 1595934901. DOI: 10.1145/1159789.1159792. URL: https://doi.org/10.1145/1159789.1159792.

[4] 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.

[5] Tommaso Bolognesi and Ed Brinksma. "Introduction to the ISO specification language LOTOS". In: *Computer Networks and ISDN Systems* 14.1 (1987), pp. 25 –59. ISSN: 0169-7552. DOI: https://doi.org/10.1016/0169-7552(87)90085-7. URL: http://www.sciencedirect.com/science/article/pii/0169755287900857.

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

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

[8] 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.

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

[10] John Hughes. "Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane". In: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Ed. by Sam Lindley et al. Cham: Springer International Publishing, 2016, pp. 169–186. ISBN: 978-3-319-30936-1. DOI: 10.1007/978-3-319-30936-1_9. URL: https://doi.org/10.1007/978-3-319-30936-1_9.

[11] John Hughes. "QuickCheck Testing for Fun and Profit". In: *Practical Aspects of Declarative Languages*. Ed. by Michael Hanus. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 1–32. ISBN: 978-3-540-69611-7.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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, 79–89. ISBN: 9781450350761. DOI: 10.1145/3092703.3092706. URL: https://doi.org/10.1145/3092703.3092706.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

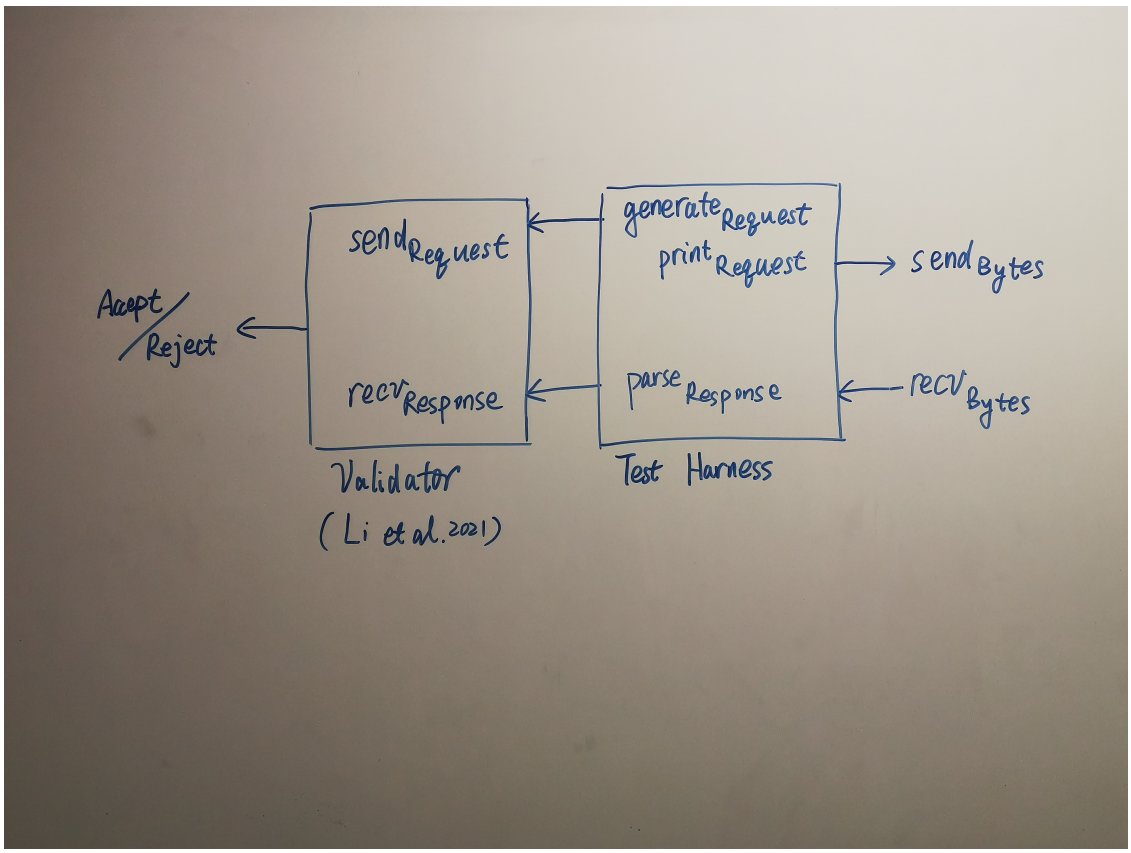


Figure 7: Test framework overview

A Language Design for Test Generation

A.1 Overview

As shown in Figure 7, the test framework consists of a *validator* that determines whether the SUT's behavior satisfies the specification, and a *test harness* that provides test inputs for the validator.

A *validator* is a client-side model program that observes messages sent and received and decides whether these interactions are conformant to the specification. For example, a simple validator for echo server is written as:

```
let validateEcho =
  request := sendRequest();
  response := recvResponse();
  if response ≠ request
  then reject
  else validate
```

Notice that the `sendRequest` event does not take the request to be sent as argument, but in stead returns the request actually sent. The validator only describes the logic that checks messages sent and received, while the test harness computes what requests to send.

The *test harness* takes a validator and turns it into an executable program that performs network interactions. It handles the validator's send and receive events, and generates the requests to be sent. A simple test harness for the validator above is written as:

```
let execute(v) =
  match v with
  | x := sendRequest(); v'(x) =>
    request := arbitraryRequest();
    sendBytes(print(request));
    execute(v'(request))
  | x := recvRequest(); v'(x) =>
    responseBytes := recvBytes();
```

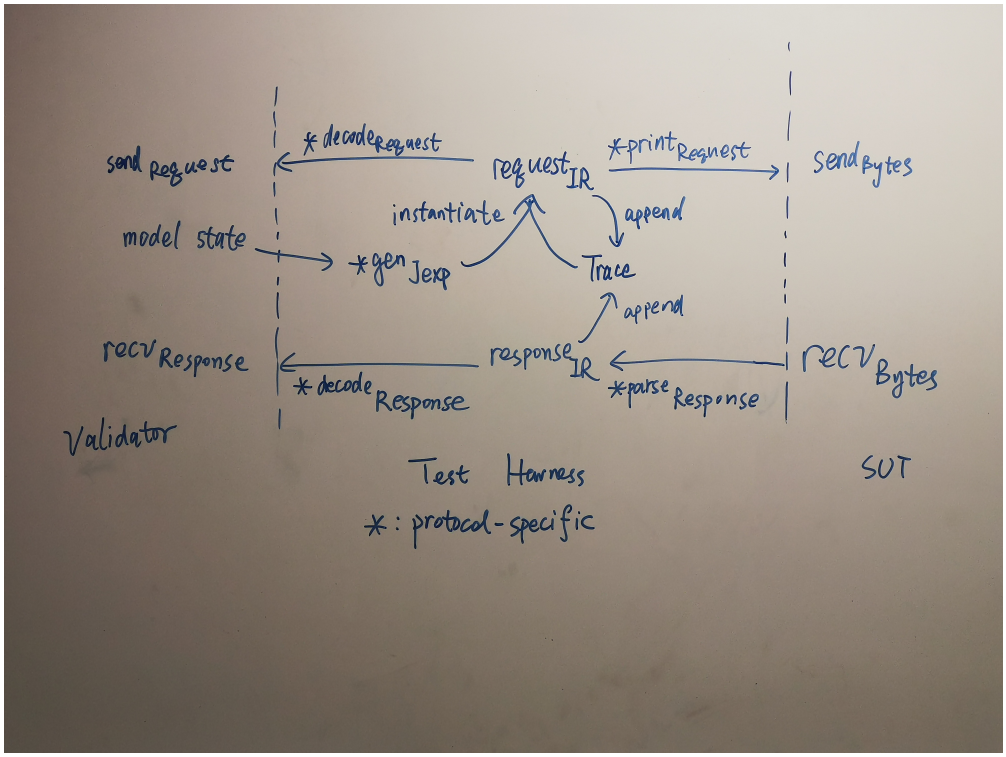


Figure 8: Test harness mechanism

```
    execute(v'(parse(responseBytes)))
  | reject => reject
  end in
execute(validateEcho)
(* ... is equivalent to ... *)
let executeValidateEcho =
  request := arbitraryRequest();
  sendBytes(print(request));
  responseBytes := recvBytes();
  if parse(responseBytes) ≠ request
  then reject
  else executeValidateEcho
```

The `arbitraryRequest` generator here produces requests randomly. To generate requests that depend on previously observed messages, my framework will extend the test harness in Figure 7 that records a trace of messages.

A.2 Architecture

As shown in Figure 8, the test harness records a trace of requests and responses, using an generic intermediate representation. Requests are generated as "J-expressions" that can be instantiated into a request IR based on the trace. For each protocol to test, the developers need to specify how to generate J-expressions that represent requests. Developers also need to define how to interpret among the IR, bytes, and application messages.

A.3 Intermediate representation language

The purpose of introducing an IR in this framework is to enable a generic method for generating requests that refer to specific fields in the trace. For example, when testing conditional HTTP requests, the generator wants to include "a precondition that uses the ETag field of a previous response"; when testing an online store, the generator wants to provide "an order ID that the server has mentioned before".

I choose JSON as the intermediate representation. Since JSON itself is widely used in web applications, the IR should provide the flexibility for developers to refer to any field in a JSON message, which might

```
Example response1 : http_response :=
  Response (Status (Version 1 1) 200 (Some "OK"))
           [Field "ETag" "tag-foo";
            Field "Content-Length" "11"]
           (Some "content-bar").

Example response2 : store_response :=
  Response__ListOrders [(233, (12, 100, 34, 500));
                        (996, (56, 400, 78, 20))].
```

```
{
  "version": {
    "major": 1,
    "minor": 1
  },
  "code": 200,
  "reason": "OK",
  "fields": {
    "ETag": "tag-foo",
    "Content-Length": "11"
  },
  "body": "content-bar"
}
```

```
{
  "code": 200,
  "orders": [
    {
      "ID": 233,
      "BuyerID": 12,
      "BuyAmount": 100,
      "SellerID": 34,
      "SellAmount": 500
    },
    {
      "ID": 996,
      "BuyerID": 56,
      "BuyAmount": 400,
      "SellerID": 78,
      "SellAmount": 20
    }
  ]
}
```

Figure 9: Application message example for HTTP and online store protocols, and their corresponding intermediate representation

involve arbitrarily deep path down the syntax tree. Figure 9 shows the intermediate representation for two protocols.

To represent the correspondence between requests and responses, the trace labels each message, and the request-response pair have the same label. Figure 10 shows a trace of messages sent and received by the tester client.

Based on this unified structure of messages, I introduce a symbolic language for representing requests, called "J-expressions", defined in Figure 11. The J-expression is similar to JSON, except that it allows a pointer notation that refers to the trace. For example, the following J-expression represents a `takeOrder` request whose order ID is equal to the second order's ID in the message labelled 30:

```
{
  "method": "takeOrder",
  "user": 2,
  "order": ref 30 (this@"orders"#2@"ID") id
}
```

Notice the `ref` in the final line: The first argument 30 is the message label in the trace. The following **this**@"orders"#2@"ID" is a path for the generator to search in the IR, pronounced "the 'ID' field in the 2nd entry of the array named 'orders' ". The last argument is the transformer function that maps the found field to the generated request, here `id` is the identical function, meaning the found order ID is used in the result request verbatim.

Suppose a trace contains a message labelled 30, and its payload is the second example in Figure 9, then the request generator can find the corresponding order ID in the request *i.e.* 996, and construct the following request IR based on the J-expression above:

```
{
```

```
[
  {
    "label": 10,
    "message": {
      "method": "GET",
      "path": "index.html"
    }
  },
  {
    "label": 20,
    "message": {
      "method": "DELETE",
      "path": "index.html"
    }
  },
  {
    "label": 20,
    "message": {
      "code": 204,
      "reason": "No Content",
    }
  },
  {
    "label": 10,
    "message": {
      "code": 410,
      "reason": "Gone"
    }
  }
]
```

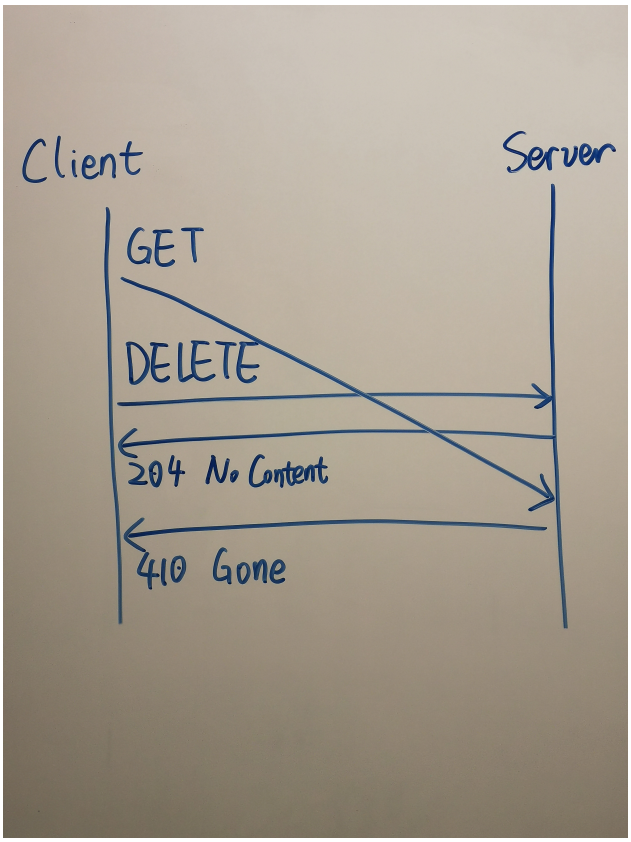



Figure 10: Example client-side trace and its corresponding IR

l trace labels
f functions
s string literals
i integers

jexp	::=		J-expressions
	|	**ref** $l\ p\ f$	
	|	$\{object\}$	
	|	$[array]$	
	|	s	
	|	i	
	|	**true**	
	|	**false**	
	|	**null**	
object	::=		
	|		
	|	"s" : $jexp$, $object$	
array	::=		
	|		
	|	$jexp$, $array$	

p	::=		J-paths
	|	**this**	
	|	$p\#i$	
	|	$p@s$	

Figure 11: Formal definition of J-expressions

```
  "method": "takeOrder",
  "user": 2,
  "order": 996
}
```

A.4 Instantiating J-expression into request IR

Suppose the test developer has generated some J-expression to represent request to send, the tester harness needs to instantiate the J-expression into the request IR. In particular, **ref** expression referrs to a field in the trace, so the test harness should locate its corresponding value.

When testing networked systems, response packets might be delayed by the network. As a result, when the tester wants to send a request that depends on a response labelled l, it is possible that the response hasn't arrived yet. In this case, to avoid blocking the program, the test harness should find some fallback options to instantiate the J-expression. A useful solution is to ignore the label and find if any message in the trace has a field of the given J-path.

Even if the labelled response has arrived, considering internal nondeterminism, the response might not have a field of the given J-path *e.g.* the J-expression doesn't have a second entry in the `"orders"` array. In this case, the test harness can ignore the index and pick any entry in the array as fallback.

A.5 Completeness of test suite

Definition 8 (Specification, implementation, and test suite)**.** An implementation $i : I$ is a software that performs some observable behavior. A specification $s : S$ is a language of valid behavior. A test suite $t : S \to I \to \mathbb{B}$ is a program that executes an implementation and computes whether its observed behavior is included in the specification.

Definition 9 (**passes** and **implements** relations)**.** An implementation **passes** a test suite if the test suite determines that its behavior is included in the specification:

$$i \textbf{ passes } t_s \triangleq t_s(i) = \textbf{true}$$

An implementation **implements** a specification if all its observable behavior is included in the specification:

$$i \textbf{ implements } s \triangleq \forall t, t_s(i) = \textbf{true}$$

Definition 10 (Soundness, exhaustiveness, and completeness)**.** A test suite is **sound** if all implementations that **implements** the specification **passes** it:

$$t \textbf{ sound}_{\triangleq} \forall s, \forall i, i \textbf{ implements } s \implies i \textbf{ passes } t_s$$

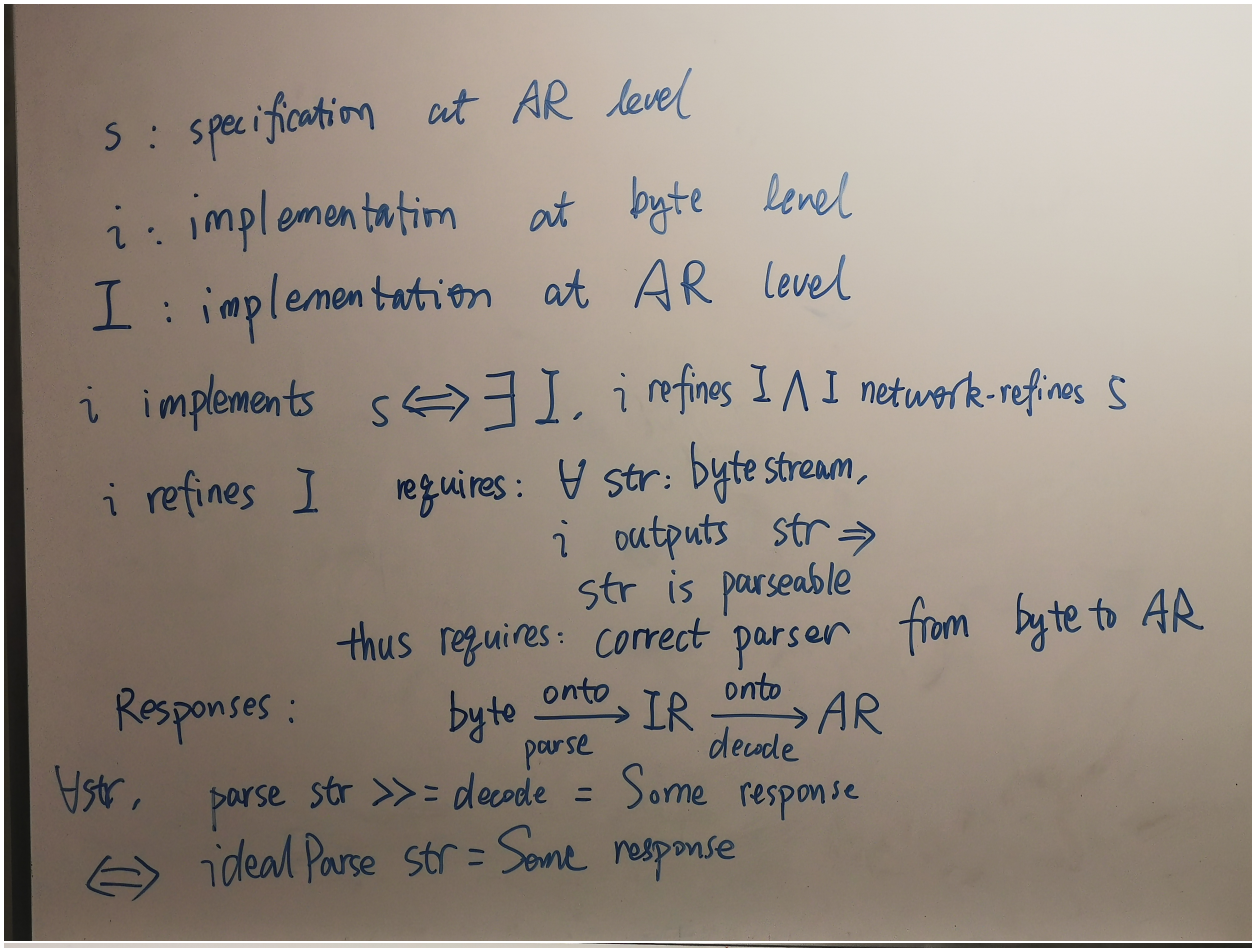
A test suite is **exhaustive** if all implementations that **passes** it **implements** the specification:

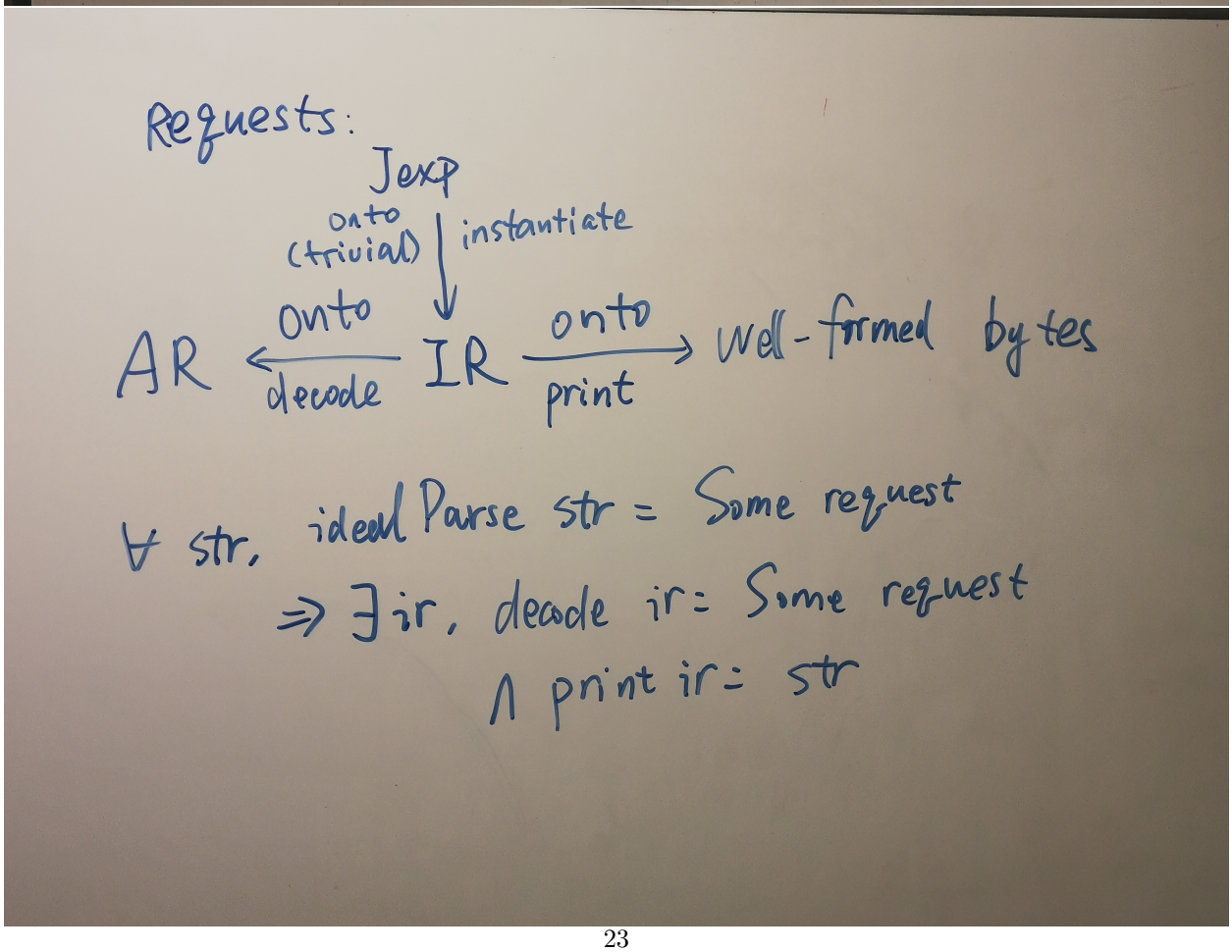
$$t \textbf{ exhaustive} \triangleq \forall s, \forall i, i \textbf{ passes } t_s \implies i \textbf{ implements } s$$

A test suite is **complete** if it's both **sound** and **exhaustive**:

$$t \textbf{ complete}_{\triangleq} t \textbf{ sound}_{\wedge} t \textbf{ exhaustive}$$

A.6 Requirements for IR design





```
(** Generate request line. *)
Definition gen_line (ss : server_state exp) : IO request_line :=
  p <- gen_path ss;; (* Emphasise paths known in the server state. *)
  m <- io_choose [Method__GET; Method__PUT];;
  ret (RequestLine m (RequestTarget__Origin p None) (Version 1 1)).

(** Find ETags in the trace. *)
Definition find_etag (tr : traceT) : IO jexp :=
  tags <- io_or (ret $ findpath (this@"fields"@"ETag") id tr) (ret []);;
  io_choose_ (ret $ Jexp__Object []) tags.

(** Generate precondition from the trace. *)
Definition gen_condition (tr : traceT) : IO jexp :=
  condition <- io_choose ["If-Match"; "If-None-Match"];;
  etag <- find_etag tr;;
  ret (jkv condition etag).

(** Generate J-expression representing the request. *)
Definition gen_request (ss : server_state exp) (tr : traceT) : IO jexp :=
  l <- gen_line ss;;
  f <- gen_condition tr;;
  match request__method l with
  | Method__PUT =>
    content <- gen_string;;
    ret (xencode l +
         jkv "fields" (host_localhost + f + content_length content) +
         jobj "body" content)
  | _ => ret $ xencode l + jkv "fields" (host_localhost + f)
  end.
```

Figure 12: Generating J-expressions for HTTP/1.1 requests

A.7 Experiment

To evaluate the effectiveness of this framework in shrinking test input, I applied it to the HTTP/1.1 experiment in Li et al. [15]. The IR encoding for HTTP was illustrated in Figure 10. Printing and parsing between the IR and HTTP is straightforward.

Figure 12 shows how to generate HTTP requests as J-expressions. This algorithm employs two heuristics: (1) When generating target path, produce known paths more frequently; (2) When generating ETags for preconditions, use ETags that exist in the trace.

After implementing the codec and the request generator, the framework interprets the ITree validator (derived from HTTP specification) into an IO program, that shrunk the initial counterexample of 11 requests into a minimal example of 2 requests.

A.8 Discussion

The experiment has revealed the problem of over-shrinking: The shrink algorithm over J-expressions is protocol-independent, so it cannot tell which expressions are interesting. For example, shrinking `HTTP/1.1` into `HTTP/0.1` makes the server always return `400 Bad Request`. My framework should employ the concepts of "positive testing" (only produce test cases that should work well) and "negative testing" (include semantically bad test cases), introduced in Quviq QuickCheck by Arts et al. [3].

A.9 Next Step

So far I've studied HTTP. To demonstrate the generality of my testing method, I plan to use this framework to test Unison.

Main property to test Unison claims to "protect both its internal state and the state of the replicas" at any point of the synchronization process. To validate this claim, the tester should launch the synchronization

process, and at any point of the process, interrupt it and/or examine its internal state and the replicas.

Observations and specifications Compared to HTTP that interacts with arbitrary clients over the network, Unison interacts with either another Unison process over the network or directly the file system. This difference of observation interface introduces a question of how to specify the space of valid observations. The HTTP specification says "even upon the most malicious client, the server should behave properly". Should we construct a malicious remote Unison to run against an implementation under test (IUT)? or should we use the should-be-good IUT on both sides, and only introduce various test cases like conflicts and interrupts?