Specifying the DeepSpec Web Server

Lennart Beringer, Joachim Breitner, Olek Gierczak, Wolf Honore, Nicolas Koh, Yao Li, Yishuai Li, William Mansky, Benjamin C. Pierce, Stephanie Weirich, Li-Yao Xia, Steve Zdancewic

DeepSpec Workshop @ PLDI
June, 2018
Specifying the DeepSpec Web Server

Lennart Beringer, Joachim Breitner, Olek Gierczak, Wolf Honore, Nicolas Koh, Yao Li, Yishuai Li, William Mansky, Benjamin C. Pierce, Stephanie Weirich, Li-Yao Xia, Steve Zdancewic

DeepSpec Workshop @ PLDI
June, 2018
Goal: A “single QED” encompassing the entire software and hardware stack

HTTP(S) spec
Web server
POSIX API
CertiKOS
RISC-V ISA
RISC-V
Transistors

Executable high-level specification of HTTP(S) protocols and web services
System call interface specification
Instruction-set specification
RTL description of circuit behaviors
The DeepSpec Web Server

• Based on popular GNU libmicrohttpd library
  • Clean separation between core HTTP-level functionality and the specifics of particular web services

• Aimed at embedded web servers (E.g. IoT device controllers)

• Current implementation (in C)
  • Parsing / printing of core HTTP formats; basic GET functionality

• Current specification (in Coq): ditto, plus
  • PUT functionality
  • ETag support for bandwidth conservation

• Later:
  • Broader coverage of HTTP standard documents
  • TLS authentication
  • Support for database-backed web services
What metalanguage? How to make it testable? How to integrate with VST?

Executable high-level specification of HTTP(S) protocols and web services

Hoare-style specification of system calls
Running Example
A “Swap Server”

```plaintext
CONNECT 'A'

CONNECT 'B'

CONNECT 'C'

CONNECT 'B'
```
Concrete swap server in C

```c
#include "macros.h"
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/select.h>
#include <sys/socket.h>

typedef int socket_fd;

enum state {
    RECVING,
    SENDING,
    DONE,
    DELETED,
};

struct connection {
    socket_fd fd;
    ssize_t send_size;
    char *send_buffer;
    struct connection* next;
}```
Concrete swap server in C

```c
#include <sys/socket.h>
#include <sys/select.h>
#include <netinet/in.h>
#include <unistd.h>
#include <string.h>
#include "macros.h"

int main(int argc, char *argv[]) {
    int fd, r, port = 8000;
    int maxsock = INVALID_SOCKET;
    struct connection *head, *pos;  
    socklen_t addr_len = sizeof(struct sockaddr_in);
    struct sockaddr_in addr;
    memset(&addr, 0, sizeof(struct sockaddr_in));
   _FD_ZERO(&rs);
    FD_ZERO(&ws);
    FD_ZERO(&es);
    maxsock = INVALID_SOCKET;
    port = atoi(argv[1]);
    if (argc > 1) {
        fd = socket(AF_INET, SOCK_STREAM, 0);
        memset(&addr, 0, sizeof(struct sockaddr_in));  
        addr.sin_port = htons(port);
        addr.sin_family = AF_INET;
        addr_len = sizeof(struct sockaddr_in);
        if (-1 == r)
            r = bind(fd, (struct sockaddr*)&addr, addr_len);
        if (-1 == r)
            r = listen(fd, SOMAXCONN);
        if (-1 == r)
            return -1;
    }
    else 
        return -1;
    sock_fd max_fd = NULL;
    for (pos = head; NULL != pos; pos = pos->next) {
        if (INVALID_SOCKET == pos->fd)
            s = pos->fd;
        while (NULL != pos)
            accept_connection(socket, &head);
        if (FD_ISSET(socket, &rs))
            continue;
        if (num_ready < 0)
            num_ready = select(maxsock + 1, &rs, &ws, &es, &timeout);
    }
    break;
  
  case DELETED:
  case DONE:
  r = add_to_fd_set(pos->fd, &ws, &maxsock, FD_SETSIZE);
  break;
  case SENDING:
  r = add_to_fd_set(pos->fd, &rs, &maxsock, FD_SETSIZE);
  break;
  case RECVING:
  {
    switch (pos->st)
      {
    for (pos = head; NULL != pos; pos = pos->next)
        continue;
    if (0 != r)
        r = add_to_fd_set(socket, &rs, &maxsock, FD_SETSIZE);
    
    FD_ZERO(&es);
    FD_ZERO(&ws);
    FD_ZERO(&rs);
    maxsock = INVALID_SOCKET;
    
    while (1)
        curr_size = recv_size;
        curr_buffer[0] = '0';
        conn_read(conn);
        if (read_ready)
            continue;
        if (num_ready < 0)
            num_ready = select(maxsock + 1, &rs, &ws, &es, &timeout);
    break;
  
  case DELETED:
  case DONE:
  case SENDING:
  break;
  conn->st = SENDING;
  curr_size   = recv_size;
  curr_buffer = recv_buffer;
  conn->send_size   = curr_size;
  conn->send_buffer = curr_buffer;
  free(conn->send_buffer);
  if (NULL != conn->send_buffer)
    return -1;
  free(recv_buffer);
  
  if (recv_size < 0)
    recv_size = recv(conn->fd, recv_buffer, BUFFER_LENGTH, 0);
  memset(recv_buffer, 0, BUFFER_LENGTH * sizeof(char));
  return -1;
  if (NULL == recv_buffer)
    recv_buffer = malloc(BUFFER_LENGTH * sizeof(char));
  case RECVING:
  {
    switch (conn->st)
      {
    
    if (DONE == pos->st)
    for (pos = head; NULL != pos; pos = pos->next)
        continue;
    if (NULL == conn)
        conn = malloc(sizeof(struct connection));
    head = NULL;
    conn->next = *head;
    *head = conn;
    conn->st = RECVING;
    conn->fd = fd;
    memset(conn, 0, sizeof(struct connection));
    
    
    if (INVALID_SOCKET == fd)
    fd = accept(socket, NULL, NULL);
    
    FD_SET(fd, set);
    return -1;
    if (fd > *max_fd || INVALID_SOCKET == *max_fd)
    {
        if (NULL != max_fd && INVALID_SOCKET != fd)
            *max_fd = fd;
    
    if (NULL == conn)  
        return -1;
    conn->st = DELETED;
    free(conn->send_buffer);
    if (NULL != conn->send_buffer)
        return -1;
    close(fd);
    conn->send_size   = curr_size;
    conn->send_buffer = curr_buffer;
    free(conn->send_buffer);
    if (NULL != conn->send_buffer)
        return -1;
    switch (conn->st)
      {
    
    conn->st = DELETED;
    conn->send_size   = curr_size;
    conn->send_buffer = curr_buffer;
    free(conn->send_buffer);
    if (NULL != conn->send_buffer)
        return -1;
    close(fd);
    return -1;
    
    if (r < 0)
        return -1;
    r = send(conn->fd, conn->send_buffer, conn->send_size, 0);
    
    case SENDING:
    {
      switch (conn->st)
        {
    
    conn->st = SENDING;
    curr_size   = recv_size;
    curr_buffer = recv_buffer;
    conn->send_size   = curr_size;
    conn->send_buffer = curr_buffer;
    free(conn->send_buffer);
    if (NULL != conn->send_buffer)
        return -1;
    free(recv_buffer);
    
    if (recv_size < 0)
        recv_size = recv(conn->fd, recv_buffer, BUFFER_LENGTH, 0);
    memset(recv_buffer, 0, BUFFER_LENGTH * sizeof(char));
    return -1;
    if (NULL == recv_buffer)
        recv_buffer = malloc(BUFFER_LENGTH * sizeof(char));
    case RECVING:
    {
      switch (conn->st)
        {
    
      conn->st = RECVING;
      curr_size   = recv_size;
      curr_buffer = recv_buffer;
      conn->send_size   = curr_size;
      conn->send_buffer = curr_buffer;
      free(conn->send_buffer);
      if (NULL != conn->send_buffer)
          return -1;
      free(recv_buffer);
      
      if (recv_size < 0)
          recv_size = recv(conn->fd, recv_buffer, BUFFER_LENGTH, 0);
      memset(recv_buffer, 0, BUFFER_LENGTH * sizeof(char));
      return -1;
      if (NULL == recv_buffer)
          recv_buffer = malloc(BUFFER_LENGTH * sizeof(char));
      ```}
Concrete swap server in C

Functional model in Gallina

Inductive connection_state : Type :=
  RECVING | SENDING | DONE | DELETED.

Record connection : Type :=
  {
    conn_id : connection_id;
    conn_request : string;
    conn_response : string;
    conn_response_bytes_sent : Z;
    conn_state : connection_state
  }.

Definition upd_conn_response (conn : connection) (response : string)
  : connection :=
  {
    conn_id := conn_id conn;
    conn_request := conn_request conn;
    conn_response := response;
    conn_response_bytes_sent := conn_response_bytes_sent conn;
    conn_state := conn_state conn
  }.

Definition upd_conn_response_bytes_sent
  (conn : connection) (response_bytes_sent : Z)
  : connection :=
  {
    conn_id := conn_id conn;
    conn_request := conn_request conn;
    conn_response := conn_response conn;
    conn_response_bytes_sent := response_bytes_sent;
    conn_state := conn_state conn
  }.

Definition upd_conn_state (conn : connection) (state : connection_state)
Concrete swap server in C

Functional model in Gallina
Concrete swap server in C

Functional model in Gallina

Reference implementation (in Gallina)

CoFixpoint loop (last: byte) : M
  networkE void :=
  c <- open_conn;;
  msg <- read_byte c;;
  write_byte last;;
  loop msg
Concrete swap server in C

Functional model in Gallina

Reference implementation (in Gallina)
Early results: Testing stock web servers

- **Apache**
  - Nonstandard responses:
    - For GET requests that expect 200 OK, Apache sometimes closes the connection before sending the full response.
    - For GET requests that expect 404 Not Found, Apache sometimes responds 403 Forbidden.
  - **Wrong behavior:**
    1. Unconditional PUT, return 204 No Content
    2. Unconditional GET, return 200 OK with ETag
    3. Conditional If-Match PUT with ETag from 2, return 412 Precondition Failed
    4. Unconditional GET, return 200 OK with content from 3

- **Nginx**
  - One similar (but less serious) wrong behavior found so far.

I.e., The server said it was rejecting our PUT, but actually executed it!
A Common Metalanguage
Too many metalanguages!

- Network-level HTTP spec
  - Functional program in Gallina

- Web server implementation
  - CompCert “observation traces”

- VST C verification tool
  - Hoare triples in separation logic

- CertiKOS
  - “Layer interfaces”
Interaction Trees

• “Abstract syntax trees” for computations

• Nodes labeled with constructors drawn from some set of observable effects and branching corresponding to possible results of observations

• Can be given a variety of semantics by supplying different interpretations for the effects

• Effects can be varied to model different views of a computation
  • e.g., “OS view” vs “network view”
Interaction Trees

An $M E X$ is the denotation of a program as a possibly infinite (coinductive) tree, parameterized over a type $Event$ of observable events where:

- **leaves** correspond to final **results** labeled with $X$,
- **internal nodes** node are either
  - **internal events** (labeled $\text{Tau}$), or
  - **observable events** (labeled $\text{Vis}$, with a child for every element of the event’s result type $Y$).

CoInductive $M (Event : Type -> Type) X :=$
| Ret $(x:X)$
| Tau $(k: M Event X)$.
| Vis $\{Y: Type\} (e : Event Y) (k : Y -> M Event X)$
**Network events**

Inductive networkE : Type -> Type :=  
| OpenConn : networkE connection 
| ReadByte : connection -> networkE (option byte) 
| WriteByte : connection -> byte -> networkE unit.

Definition embed : forall {E X}, E X -> M E X :=  
  fun E X e => Vis e (fun x => Ret x).

Definition open_conn : M networkE connection :=  
  embed OpenConn.
Definition read_byte conn : M networkE (option byte) :=  
  embed (ReadByte conn).
Definition write_byte conn b : M networkE unit :=  
  embed (WriteByte conn b).
Network events

Inductive networkE : Type -> Type :=
  | OpenConn : networkE connection
  | ReadByte : connection -> networkE (option byte)
  | WriteByte : connection -> byte -> networkE unit.

Definition embed : forall {E X}, E X -> M E X :=
  fun E X e => Vis e (fun x => Ret x).

Definition open_conn : M networkE connection :=
  embed OpenConn.
Definition read_byte conn : M networkE (option byte) :=
  embed (ReadByte conn).
Definition write_byte conn b : M networkE unit :=
  embed (WriteByte conn b).
Network events

Inductive networkE : Type -> Type :=
  | OpenConn : networkE connection
  | ReadByte : connection -> networkE (option byte)
  | WriteByte : connection -> byte -> networkE unit.

Definition embed : forall {E X}, E X -> M E X :=
  fun E X e => Vis e (fun x => Ret x).

Definition open_conn : M networkE connection :=
  embed OpenConn.

Definition read_byte conn : M networkE (option byte) :=
  embed (ReadByte conn).

Definition write_byte conn b : M networkE unit :=
  embed (WriteByte conn b).
“Reference implementation” of the swap server as an interaction tree

CoFixpoint loop (last: byte) : M networkE void :=
c <- open_conn;;
msg <- read_byte c;;
write_byte last;;
loop msg
Testing over the Network
Where to observe?
One view

• A program interacts with the outside world via the operating system.

• The observations we can make of it are the OS calls it makes (together with their results)

• But what if we want to verify the whole stack?
  • program + OS + hardware??
Another view

- A program running on an OS running on some hardware interacts with the outside world via its I/O ports
- Put a probe on the ethernet cable coming out of the box
- But what’s actually on the wire?
  - electrical or optical waves?
  - raw seething bits?
  - IP packets?
  - TCP packets?
  - (HTTP messages?)
Yet another view

• Put some client machines on the other end of the wire
• On each, run a browser on top of a standards-compliant protocol stack
• Put a probe into the internals of each client, at the level where TCP connections are decoded into bytestreams
• Specify OS primitives like send and recv in terms of the effects they have on externally visible bytestreams (network events)
Inductive testingE : Type -> Type :=
| Connect : testingE connection
| ToServer : connection -> testingE byte
| FromServer : connection -> testingE byte
| Fail : testingE void.

Definition Tester := M testingE void.

Definition connect : M testingE connection :=
  embed Connect.
Definition toserver c : M testingE byte :=
  embed (ToServer c).
Definition fromserver c : M testingE byte :=
  embed (FromServer c).
Definition fail {X} : M testingE X :=
  Vis Fail (fun v : void => match v with end).
Inductive testingE : Type -> Type :=
| Connect : testingE connection
| ToServer : connection -> testingE byte
| FromServer : connection -> testingE byte
| Fail : testingE void.

Definition Tester := M testingE void.

Definition connect : M testingE connection :=
    embed Connect.
Definition toserver c : M testingE byte :=
    embed (ToServer c).
Definition fromserver c : M testingE byte :=
    embed (FromServer c).
Definition fail {X} : M testingE X :=
    Vis Fail (fun v : void => match v with end).
CoFixpoint swap_tester (last : byte) : Tester :=
c <- connect;;
rq <- toserver c;;
rs <- fromserver c;;
if rs = last ?
   then swap_tester rq
else fail.
Inductive Event : Type :=
| ConnectE : connection -> Event
| ToServerE : connection -> byte -> Event
| FromServerE : connection -> byte -> Event.

Definition Trace := Trace.
descramble

all server-side traces that could plausibly have generated the observed one

observed trace
(what the tester saw)
Fixpoint check (ds: Trace) (t: Tester) : bool :=
  (* Check that trace ds is accepted by t... *)

Definition descramble (es: Trace) : list Trace :=
  (* Return all "server-side" traces that could
    have generated "client-side" observations es... *)

Definition acceptable (spec: Tester) (es: Trace) : bool :=
  existsb (fun ds => check ds spec) (descramble es).
acceptable (swap_spec 'A')
[ConnectE 0;
  ToServerE 0 'B';
  FromServerE 0 'A';
ConnectE 1;
  ToServerE 1 'C';
  FromServerE 1 'B']
= true

acceptable (swap_spec 'A')
[ConnectE 0;
  ToServerE 0 'B';
  FromServerE 0 'A';
ConnectE 1;
  ToServerE 1 'C';
  FromServerE 1 'A']
= false
more interestingly...

acceptable (swap_spec 'A')
[ConnectE 0;
 ConnectE 1;
 ToServerE 0 'B';
 ToServerE 1 'C';
 FromServerE 1 'B';
 FromServerE 0 'A']
= true

acceptable (swap_spec 'A')
[ConnectE 0;
 ConnectE 1;
 ToServerE 0 'B';
 ToServerE 1 'C';
 FromServerE 1 'A';
 FromServerE 0 'A']
= false
Further Steps

• Test! (Use QuickChick to generate test cases)

• Test faster!
  • Annotate Testers with hints describing how to generate requests to send to the server
  • Incrementalize / interleave testing and test-case generation
    • … so that we can see hints when we need them
    • … to manage combinatorial explosion of descrambling

• Test real servers (some more)!

• Prove!
Interaction Trees and VST
Interaction Trees in VST specifications

C program

{ ALLOWED(whole_server_itree) ; SOCKAPI(...) ; ... }
Interaction Trees in VST specifications

```
{  ALLOWED(whole_server_itree) ;  SOCKAPI(...);  ...  }
```

C program

```
{  ALLOWED(null_itree) ;  SOCKAPI(...);  ...  }
```
Interaction Trees in VST specifications

Remaining behavior

\[
\{ \texttt{ALLOWED(whole\_server\_itree)} ; \texttt{SOCKAPI(...)} ; \ldots \}
\]

C program

\[
\{ \texttt{ALLOWED(null\_itree)} ; \texttt{SOCKAPI(...)} ; \ldots \}
\]
Interaction Trees in VST specifications

Remaining behavior

```c
{ ALLOWED(whole_server_itree) ; SOCKAPI(…) ; … }
```

C program

```c
{ ALLOWED(null_itree) ; SOCKAPI(…) ; … }
```

Remaining behavior

after this bit of C code runs
**VST Specification for Posix recv system call**

**Definition**  
recv_msg_spec (T : Type) :=

DECLARE _recv_msg

WITH t : SocketMonad T, k : option string -> SocketMonad T,
    client_conn : connection_id,
    st : SocketMap, (* A SocketMap maps file descriptors to socket states *)
    fd: sockfd,
    buf_ptr: val, alloc_len: Z, sh: share

PRE [ 1 OF tint, 2 OF (tptr tuchar), 3 OF tuint, 4 OF tint ]

PROP ( lookup_socket st fd = ConnectedSocket client_conn; writable_share sh ;
    trace_incl (msg <- recv client_conn (Z.to_pos alloc_len);; k msg) t
)

LOCAL (temp 1 (Vint (Int.repr (descriptor fd)));
    temp 2 buf_ptr;
    temp 3 (Vint (Int.repr alloc_len));
    temp 4 (Vint (Int.repr 0))
)

SEP ( SOCKAPI st; (* API memory contains some representation of st *)
    ALLOWED t;
    data_at_ sh (tarray tuchar alloc_len) buf_ptr
)

*postcondition on next slide...*
POST [ tint ]

EX result : unit + option string, (* result is either a failure, EOF, or received message *)
EX st' : SocketMap,
EX r : Z, EX contents: list val,
PROP ( (0 <= r <= alloc_len) \ / r = -1; Zlength contents = alloc_len ;
    r > 0 ->
    (exists msg, result = inr (Some msg) \ /
        Zlength (val_of_string msg) = r \ /
        sublist 0 r contents = (val_of_string msg) \ /
        sublist r alloc_len contents =
            list_repeat (Z.to_nat (alloc_len - r)) Vundef) \ / (st' = st) ;
    r = 0 -> (result = inr None \ / contents = list_repeat (Z.to_nat alloc_len) Vundef \ /
        st' = update_socket_state st fd OpenedSocket) ;
    r < 0 -> (result = inl tt \ /
        contents = list_repeat (Z.to_nat alloc_len) Vundef \ /
        st' = st)
)
LOCAL ( temp ret_temp (Vint (Int.repr r)) )
SEP ( SOCKAPI st' ;
    ALLOWED ( match result with inl tt => t | inr msg_opt => k msg_opt end ) ;
    data_at sh (tarray tuchar alloc_len) contents buf_ptr
).

Status

• Swap server correctness proof nearly complete
  • Still experimenting with various ways of refactoring the specifications

Next steps

• Prove that CertiKOS implementation of POSIX socket API satisfies the axioms
• Scale proofs up to web server…
More slides
<table>
<thead>
<tr>
<th>Component</th>
<th>Approximate LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common: axioms for socket API</td>
<td>500</td>
</tr>
<tr>
<td>C code for echo server</td>
<td>140</td>
</tr>
<tr>
<td>Interaction tree for echo server</td>
<td>100</td>
</tr>
<tr>
<td>Hoare triples for functions in echo server</td>
<td>200</td>
</tr>
<tr>
<td>VST proofs of C-to-OS-level-spec</td>
<td>400</td>
</tr>
<tr>
<td>Coq proofs of OS-level-to-network-level</td>
<td>1000-2000 ?</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1500-2500ish</strong></td>
</tr>
<tr>
<td>C code for web server</td>
<td>2880</td>
</tr>
<tr>
<td>Interaction tree for web server</td>
<td>2000?</td>
</tr>
<tr>
<td>Hoare triples for web server</td>
<td>4000?</td>
</tr>
<tr>
<td>VST proofs of C-to-OS-level-spec</td>
<td>8000?</td>
</tr>
<tr>
<td>Coq proofs of OS-level-to-network-level</td>
<td>20-40k?</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>30-50k ???</strong></td>
</tr>
</tbody>
</table>
Exercising the HTTP specification from both sides
User-level spec

HTTP-based application (running in browser)

HTTP(S) + web service spec

Web service

Stock web servers
Questions?