Programming Language Abstractions for Modularly Verified Distributed Systems

\[ \vdash \{ P \} \ c \ \{ Q \} \]

James R. Wilcox    Zach Tatlock    Ilya Sergey

PLSE

PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING

UCL
Distributed Systems
Distributed *Infrastructure*
Distributed Applications
Verified Distributed Systems
Verified Distributed Systems

holds(Φ, S, \(\leadsto_1\)) \implies holds(\text{transfer}(\Phi), T(S), \leadsto_2)
Verified Distributed *Infrastructure*

\[
\text{holds}(\Phi, S, \sim_1) \rightarrow \\
\text{holds}(\text{transfer}(\Phi), T(S), \sim_2)
\]
Verified Distributed Infrastructure

holds(\Phi, S, \sim_1) \rightarrow
holds(\text{transfer}(\Phi), T(S), \sim_2)
Verified Distributed Applications

\[
\text{holds}(\Phi, S, \rightsquigarrow_1) \rightarrow \\
\text{holds}(\text{transfer}(\Phi), T(S), \rightsquigarrow_2)
\]
Verified Distributed Applications

\[ \text{holds}(\Phi, S, \sim_1) \rightarrow \text{holds}(\text{transfer}(\Phi), T(S), \sim_2) \]

Veri-

Iron

Wow

-Cert
Verified Distributed Applications

Challenging to verify apps in terms of infra. **verify clients by starting over!**

Indicates deeper problems with composition **one node’s client is another’s server!**
Challenging to verify apps in terms of infra. verify clients by starting over!

Indicates deeper problems with composition one node’s client is another’s server!

(Make it possible to) verify clients verify clients without starting over!

Will also enable more general composition
Composition: A way to make proofs harder
Composition: A way to make proofs harder

When distracting language issues are removed and the underlying mathematics is revealed, compositional reasoning is seen to be of little use.
Approach

*Distributed Hoare Type Theory*

\[ \vdash \{ P \} \; c \; \{ Q \} \]
Distributed Interactions

Servers and Clients

Optimizations
gcc -O3

Combining Services

Horizons
Cloud Compute

C → S

C

S

21
Cloud Compute

C

{3,7}

S

21
Cloud Compute
while True:
    (from, n) <- recv
    send (n, factors(n)) to from
Cloud Compute: Server

```python
while True:
    (from, n) <- recv
    send (n, factors(n)) to from
```

Traditional specification:
messages from server have correct factors

Proved by finding an invariant of the system
Cloud Compute: Server
Cloud Compute: Client
Cloud Compute: Client

send 21 to server
(_, ans) <- recv
assert ans == {3, 7}
Cloud Compute: Client

send 21 to server
(_, ans) <- recv
assert ans == {3, 7}

Expand system to include clients

Need to reason about client-server interaction

introduce protocol
Protocols

C \rightarrow S
\{3,7\}

S \rightarrow C
21
Protocols
Protocols make it possible to verify clients!
Protocols
Protocols

State:
abstract state of each node
Protocols

State:
abstract state of each node

Transitions:
allowed sends and receives
Cloud Compute Protocol

State:

Transitions:
Cloud Compute Protocol

State:

permissions: Set<Msg>

Transitions:
Cloud Compute Protocol

State:

permissions: Set<Msg>

Transitions:

Send Req
Recv Req
Send Resp
Recv Resp
Cloud Compute: Protocol

State:

Transitions:

perm: Set<Msg>

Effect:

add (from, n) to perm

Recv Request n

Send Req

Recv Req

Send Resp

Recv Resp
Cloud Compute: Protocol

Send Response \((n,l)\)

Requires:
\[
l == \text{factors}(n)
\]
\[
(n, to) \text{ in perm}
\]

Effect:
\[
\text{removes} \ (n, to) \text{ from perm}
\]
Cloud Compute: Protocol

Recv Response \( l \)

Ensures:

\[
l == \text{factors}(n)
\]

\[
(n, to) \in \text{perm}
\]
Cloud Compute: Protocol

State:

permissions: Set<Msg>

Transitions:

Send Req
Recv Req
Send Resp
Recv Resp
Cloud Compute: Protocol

Protocols make it possible to verify clients!
From Protocols to Types

\[ \inf \{ P \} \quad c \quad \{ Q \} \]
From Protocols to Types

\[
\left\{ \begin{array}{l}
\text{send } m \text{ to } h
\end{array} \right\}
\]
From Protocols to Types

$\vdash \{ \{ \} \text{ send}_t m \text{ to } h \} \{ \}$
From Protocols to Types

\[ t \in \{ \{ \text{send}_t \} \} \]
From Protocols to Types

\[ t \in \text{pre} \quad P \Rightarrow Pre^t \]

\[ \vdash \{ P \} \text{send}^t m \text{ to } h \{ \text{ } \} \]
From Protocols to Types

\( t \in \{P\} \quad P \Rightarrow Pre^{t} \)

\( \vdash \{P\} \text{send}^{t} m \text{to} h \{sent^{t}(m,h)\} \)
Cloud Compute: Client

send 21 to server
(_, ans) <- recv
assert ans == {3, 7}
Cloud Compute: Client

send 21 to server
(_, ans) <- recv
assert ans == \{3, 7\}

recv ensures correct factors
Cloud Compute: More Clients

```
send 21 to server₁
send 35 to server₂
(_, ans₁) <- recv
(_, ans₂) <- recv
assert ans₁ ∪ ans₂ == {3, 5, 7}
```
Cloud Compute: More Clients

`send 21 to server_1`

`send 35 to server_2`

`(_, ans_1) <- recv`

`(_, ans_2) <- recv`

`assert ans_1 \cup ans_2 == \{3, 5, 7\}`

Same protocol enables verification
Cloud Compute: More Clients

send 21 to server_1
send 35 to server_2
(_, ans_1) <- recv
(_, ans_2) <- recv
assert ans_1 \cup ans_2 == \{3, 5, 7\}

Same protocol enables verification
Cloud Compute: Server

while True:
    (from, n) <- recv
    send (n, factors(n)) to from
Cloud Compute: Server

```python
while True:
    (from, n) <- recv
    send (n, factors(n)) to from
```

Precondition on `send` requires correct factors
cache = {}
while True:
    (from, n) <- recv
    ans = if n ∈ cache then cache[n]
          else factors(n)
    cache[n] = ans
    send (n, ans) to from
Cloud Compute: More Servers

cache = {}
while True:
    (from, n) <- recv
    ans = if n ∈ cache then cache[n]
    else factors(n)
    cache[n] = ans
    send (n, ans) to from

Still follows protocol!
while True:
    (from, n) <- recv
send n to backend
(_, ans) <- recv
send (n, ans) to from
while True:
    (from, n) <- recv
    send n to backend
    (_, ans) <- recv
    send (n, ans) to from

Still follows protocol!
while True:
    (from, n) <- recv
    send n to backend
    (_, ans) <- recv
    send (n, ans) to from

Any combination of transitions follows protocol

*Well-typed programs don’t go wrong!*

One node’s client is another’s server!
Horizons

Sophisticated protocol composition
  *e.g.* computation uses separate database

Adding other effects
  *e.g.* mutable heap, threads, failure…

Fault tolerance
  *what do Verdi’s VSTs look like here?*
Verified Distributed Applications

\[
\begin{align*}
\text{holds}(\Phi, S, \sim_1) & \rightarrow \\
\text{holds}(\text{transfer}(\Phi), T(S), \sim_2)
\end{align*}
\]
Verified Distributed Applications

$\text{holds}(\Phi, S, \sim_1) \rightarrow$
$\text{holds}(\text{transfer}(\Phi), T(S), \sim_2)$
Verified Distributed Applications

Challenging to verify apps in terms of infra. *verify clients by starting over!*

Indicates deeper problems with composition: *one node’s client is another’s server!*
Challenging to verify apps in terms of infra. *verify clients by starting over!*

Indicates deeper problems with composition *one node’s client is another’s server!*

Protocols make it possible to verify clients *reason about client-server interaction*

Also enable more general composition

Any combination of transitions follows protocol *Well-typed programs don’t go wrong!*
Protocols make it possible to verify clients reason about client-server interaction

Also enable more general composition

Any combination of transitions follows protocol

Well-typed programs don’t go wrong!
Protocols make it possible to verify clients
reason about client-server interaction

Also enable more general composition

Any combination of transitions follows protocol
Well-typed programs don’t go wrong!

Composition is hard
but important for infrastructure

Achieve with types
syntactic theory of composition