Programming and Proving with Distributed Protocols

Disel: Distributed Separation Logic

\[ \models \{ P \} \mathcal{C} \{ Q \} \]

http://distributedcomponents.net

Ilya Sergey  James R. Wilcox  Zach Tatlock
Distributed Systems
Distributed *Infrastructure*
Distributed Applications
Verified Distributed Systems

holds(\Phi, S, \rightsquigarrow_1) \rightarrow
holds(\text{transfer}(\Phi), T(S), \rightsquigarrow_2)
Verified Distributed *Infrastructure*

holds(\(\Phi, S, \sim_1\)) \rightarrow
holds(transfer(\(\Phi\)), \(T(S), \sim_2\))
Verified Distributed *Infrastructure*

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

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

holds(\Phi, S, \sim_1) \rightarrow
holds(\text{transfer}(\Phi), T(S), \sim_2)
Challenging to verify apps in terms of infra.  
starting from scratch is unacceptable

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

- Client reasoning
- Invariants
- Separation
**Challenges**
- Client reasoning
- Invariants
- Separation

**Solutions**
- Protocols
  - **WithInv** rule
  - **Frame** rule/Hooks

Disel: ![Diagram](disel.png) \(\vdash \{P\} \ c \ \{Q\}\)
Outline

Protocols and running example

Logical mechanisms

- programming with protocols
- invariants
- framing and hooks

Implementation and future work
Cloud Compute
Cloud Compute

C

21

S
Cloud Compute
Cloud Compute
Cloud Compute: Server

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

while true:
    (from, n) <- recv Req
    send Resp(n, factors(n)) to from

Traditional specification:
messages from server have correct factors

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

C \xleftrightarrow{21} S

\{3,7\}
Cloud Compute: Client
Cloud Compute: Client

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

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

Start over with clients in system?
Cloud Compute: Client

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

Start over with clients in system?

In Disel: use protocol to describe client interface
Protocols
Protocols
Protocols

A protocol is an **interface** among nodes
Protocols

A protocol is an **interface** among nodes

Enables compositional verification
Cloud Compute Protocol

Messages:
Cloud Compute Protocol

Messages:

State:
Cloud Compute Protocol

Messages:

State:

Transitions:
Cloud Compute Protocol

Messages:

State:

Transitions:
  Sends: precondition and effect
Cloud Compute Protocol

Messages:

State:

Transitions:
  Sends: precondition and effect
  Receives: effect
Cloud Compute Protocol

Messages:

State:

Transitions:
  Sends:
  Receives:
Cloud Compute Protocol

Messages: \( \text{Req}(n) \ | \ \text{Resp}(n,s) \)

State:

Transitions:
- Sends:
- Receives:
Cloud Compute Protocol

Messages: $\text{Req}(n) \mid \text{Resp}(n,s)$

State: outstanding: $\text{Set}<\text{Msg}>$

Transitions:
Sends:
Receives:
Cloud Compute Protocol

Messages: Req(n) | Resp(n,s)

State: outstanding: Set(Msg)

Transitions:
Sends:
- Req
- Resp

Receive:
- Req
- Resp
Cloud Compute

Send \text{Req}(n)

Precondition: none

Effect: none
Cloud Compute

Receive Req(n)
Effect: add (from, n) to out

\[
\text{Req}(21) \rightarrow \{(C, 21)\}
\]
Cloud Compute

Send Resp\((n, l)\)

Requires: \(l = \text{factors}(n)\)
\((n, to) \text{ in } \text{out}\)

Effect: removes \((n, to)\) from \text{out}
Cloud Compute

Recv Resp(n, l)

Effect: none
Cloud Compute Protocol

**Messages:**  \( \text{Req}(n) \ | \ \text{Resp}(n,s) \)

**State:** \textit{outstanding}: Set\(<\text{Msg}>\)

**Transitions:**

- **Sends:**
  - Req
  - Resp

- **Receives:**
  - Req
  - Resp
Outline

Protocols and running example

Logical mechanisms

- programming with protocols
- invariants
- framing and hooks

Implementation and future work
Cloud Compute: Server

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

while true:
    (from, n) <- recv Req
    send Resp(n, factors(n)) to from

Precondition on send requires correct factors
Cloud Compute: Server

while true:
  (from, n) <- recv Req
  send Resp(n, factors(n)) to from

Precondition on send requires correct factors
Cloud Compute: Server

while true:
    (from, n) <- recv Req
    send Resp(n, factors(n)) to from

Precondition on send requires correct factors
Cloud Compute: Server

\[ t \in \text{Cloud Compute} \]

\[
\{ \text{send} t m \text{ to } h \} \]

while true:
  (from, n) <- recv Req
  send Resp(n, factors(n)) to from

Precondition on send requires correct factors
Cloud Compute: Server

\[ t \in \text{server} \]

\[
\begin{array}{c}
\text{while true:} \\
(\text{from, n}) \leftarrow \text{recv} \ \text{Req} \\
\text{send} \ \text{Resp}(n, \text{factors}(n)) \ \text{to} \ \text{from}
\end{array}
\]

Precondition on \textbf{send} requires correct factors.
Cloud Compute: Server

\[ t \in \text{Server} \]

\[
\vdash \{ Pre_t \} \text{send}_t m \text{ to } h \{ sent_t (m, h) \}
\]

while true:
  (from, n) <- recv Req
  send Resp(n, factors(n)) to from

Precondition on send requires correct factors
Cloud Compute: Client

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

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

recv doesn’t ensure correct factors
Cloud Compute: Client

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

recv doesn’t ensure correct factors
Cloud Compute: Client

\[
\begin{align*}
\exists t & \in \text{client} \\
\end{align*}
\]

\[
\begin{align*}
\vdash \{T\} \text{recv}_t m \{\text{recvd}(m)\}
\end{align*}
\]

send Req(21) to server

(_, ans) <- recv Resp

assert ans == \{3, 7\}

recv doesn’t ensure correct factors
Protocol Invariants

\[
\vdash \{ P \} \ c \ \{ Q \} \quad I \text{ inductive}
\]

\[
\vdash' \{ P \wedge I \} \ c \ \{ Q \wedge I \}
\]
Protocol Invariants

\[ \{P\} \text{ c } \{Q\} \quad I \text{ inductive} \]

\[ \{P \land I\} \text{ c } \{Q \land I\} \]

Protocol where every state satisfies \( I \)
Cloud Compute: Client

\[ t \in \{ \text{recv} \} \]

\[ \text{send } \text{Req}(21) \text{ to server} \]
\[ (_, \text{ans}) \leftarrow \text{recv} \text{ Resp} \]
\[ \text{assert } \text{ans} == \{3, 7\} \]

Now \text{recv} ensures correct factors
Cloud Compute: More Clients

\[
\text{send } \text{Req}(21) \text{ to server}_1 \\
\text{send } \text{Req}(35) \text{ to server}_2 \\
(_, \text{ ans}_1) \leftarrow \text{recv} \text{ Resp} \\
(_, \text{ ans}_2) \leftarrow \text{recv} \text{ Resp} \\
\text{assert } \text{ans}_1 \cup \text{ans}_2 = \{3, 5, 7\}
\]
Cloud Compute: More Clients

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

Same protocol enables verification
Frame rule

\[
\begin{array}{c}
\{ P \} \ c \ \{ Q \} \quad R \ \text{stable} \\
\hline
\{ P \ast R \} \ c \ \{ Q \ast R \}
\end{array}
\]
Frame rule

\[
\begin{array}{c}
\{ P \} \ c \ \{ Q \} \quad R \ \text{stable} \\
\hline \\
\{ P R \} \ c \ \{ Q R \} \\
\end{array}
\]

Reuse invariants from component protocols
Frame rule

\[
\begin{array}{c}
\text{\textbf{Re}use invariants from component protocols} \\
\end{array}
\]
Frame rule: Hooks

\[
\allowdisplaybreaks
\begin{align*}
\Gamma & \vdash \{P\} \ c \ \{Q\} \quad R \text{ stable} \\
\hline
\Gamma & \vdash \{P \ast R\} \ c \ \{Q \ast R\}
\end{align*}
\]
Frame rule: Hooks

\[ \vdash \{P\} \; c \; \{Q\} \quad R \text{ stable} \]

\[ \vdash \{P \ast R\} \; c \; \{Q \ast R\} \]

Allows one protocol to restrict another
Outline

Protocols and running example

Logical mechanisms
  programming with protocols
  invariants
  framing and hooks

Implementation and future work
Implementation

Shallowly embedded in Coq with full power of functional programming

Executable via extraction to OCaml via trusted shim to implement semantics

Case study: two-phase commit exercises all features of the logic
Related and Future Work

Concurrent separation logics

Iris, FCSL, CAP, …

Adding other effects

e.g. mutable heap, threads, failure…
Composition: A way to make proofs harder
Composition: A way to make proofs harder

“In 1997, the unfortunate reality is that engineers rarely specify and reason formally about the systems they build. It seems unlikely that reasoning about the composition of open-system specifications will be a practical concern within the next 15 years.”
Verified Distributed Applications

- Client reasoning
- Invariants
- Separation

Solutions

- Protocols
- WithInv rule
- Frame rule/Hooks

Disel: \[ \frac{}{\{P\} \text{ c } \{Q\}} \]