Behavioural Type-Based Static Verification Framework for GO

Julien Lange  Nicholas Net  Bernardo Toninho  Nobuko Yoshida
A static verification framework for message passing in Go using behavioural types

JANUARY 25, 2018

tags: Concurrency, Programming Languages

A static verification framework for message passing in Go using behavioural types. Lange et al., ICSE'18

With thanks to Alexis Richardson who first forwarded this paper to me.

We're jumping ahead to ICSE'18 now, and a paper that has been accepted for publication there later this year. It fits with the theme we've been exploring this week though, so I thought I'd cover it now. We've seen verification techniques applied in the context of Rust and JavaScript, looked at the integration of linear types in Haskell, and today it is the turn of Go!
Go programming language @ Google (2009)

- Message-Passing based multicore PL, successor of C
- Do not communicate by shared memory; instead, share memory by communicating
  Go Lang Proverb
- Explicit channel-based concurrency
  - Buffered I/O communication channels
  - Lightweight thread spawning - goroutines
  - Selective send/receive

Fun
- Dropbox, Netflix, Docker, CoreOS
GO has a runtime deadlock detector

How can we detect partial deadlock and channel errors for realistic programs?

Use behavioural types in process calculi

e.g. [ACM Survey, 2016] 185 citations, 6 pages

Dynamic channel creations, unbounded thread creations, recursions, ...

Scalable (synchronous/asynchronous) Modular, Refinable
- GO has a runtime deadlock detector
- How can we detect partial deadlock and channel errors for realistic programs?
- Use behavioural types in process calculi
e.g. [ACM Survey, 2016] 185 citations, 6 pages

- Dynamic channel creations, unbounded thread creations, recursions, ...

- Scalable (synchronous/asynchronous) Modular, Refinable
- Go has a runtime deadlock detector
- How can we detect partial deadlock and channel errors for realistic programs?
- Use behavioural types in process calculi
  e.g. [ACM Survey, 2016] 185 citations, 6 pages
- Dynamic, online, unbounded thread creations, recursions, ...
- Scalable (synchronous/asynchronous) Modular, Refinable
- Go has a runtime deadlock detector.
- How can we detect partial deadlock and channel errors for realistic programs?
- Use behavioural types in process calculi.
  e.g. [ACM Survey, 2016] 185 citations, 6 pages.
- Channel creations, unbounded thread creations, recursions, ...
- Scalable (synchronous/asynchronous) Modular, Refinable.
- GO has a runtime deadlock detector
- How can we detect partial deadlock and channel errors for realistic programs?
- Use behavioural types in process calculi
  e.g. [ACM Survey, 2016] 185 citations, 6 pages
- Dynamic channel creations, unbounded thread creations, ...
- Scalable (synchronous/asynchronous) Modular, verifiable
  Understandable
Our Framework

**STEP 1** Extract Behavioural Types
- (Most) Message passing features of **GO**
- Tricky primitives: selection, channel creation

**STEP 2** Check Safety/Liveness of Behavioural Types
- Model-Checking (Finite Control)

**STEP 3**
- Relate Safety/Liveness of Behavioural Types and **GO** Programs
  - 3 Classes [POPL’17]
  - Termination Check
Our Framework

**STEP 1** Extract Behavioural Types
- (Most) Message passing features of **GO**
- Tricky primitives: selection, channel creation

**STEP 2** Check Safety/Liveness of Behavioural Types
- Model-Checking (Finite Control)

**STEP 3**
- Relate Safety/Liveness of Behavioural Types and **GO** Programs
  - 3 Classes [POPL’17]
  - Termination Check
Static verification framework for Go 🐧

Overview

1. Type inference

Behavioural Types

SSA IR

Go source code

Transform and verify

2. Model checking

mCRL2 model checker

Check safety and liveness

3. Termination checking

KITTeL termination prover

Address type ↔ program gap
Concurrency in Go 🚆

Goroutines

```go
func main() {
    ch := make(chan string)
    go send(ch)
    print(<-ch)
    close(ch)
}

func send(ch chan string) {
    ch <- "Hello Kent!"
}
```

go keyword + function call

- Spawns function as goroutine
- Runs in parallel to parent
Concurrency in Go 🐘

Channels

Create new channel
- Synchronous by default

Receive from channel

Close a channel
- No more values sent to it
- Can only close once

Send to channel

```go
func main() {
    ch := make(chan string)
    go send(ch)
    print(<-ch)
    close(ch)
}

func send(ch chan string) {
    ch <- "Hello Kent!"
}
```
Concurrency in Go 🌊

Channels

```go
func main() {
    ch := make(chan string)
    go send(ch)
    print(<-ch)
    close(ch)
}

func send(ch chan string) {
    ch <- "Hello Kent!"
}

Also select-case:
- Wait on multiple channel operations
- switch-case for communication
```
Concurency in Go

Behavioural type inference

Model checking behavioural types

Termination checking

Summary

Concurrency in Go 🐸

Deadlock detection

```
1 func main() {
2   ch := make(chan string)
3   go send(ch)
4   print(<-ch)
5   close(ch)
6 }
7
8 func send(ch chan string) {
9   ch <- "Hello Kent!"
10 }
```

- Send message thru channel
- Print message on screen

Output:

```
$ go run hello.go
Hello Kent!
$
```
Concurrenty in Go 🐧

Deadlock detection

- Only one (main) goroutine
- Send without receive - blocks

Output:

```
$ go run deadlock.go
fatal error: all goroutines are asleep - deadlock!
```

```
// import _ "net"
func main() {
    ch := make(chan string)
    send(ch) // Oops
    print(<-ch)
    close(ch)
}

func send(ch chan string) {
    ch <- "Hello Kent!"
}
```
Concurrency in Go 🛑

Deadlock detection

Go’s runtime deadlock detector

- Checks if all goroutines are blocked (‘global’ deadlock)
- Print message then crash
- Some packages disable it (e.g. net)

```go
// import _ "net"

func main() {
    ch := make(chan string)
    send(ch) // Oops
    print(<-ch)
    close(ch)
}

func send(ch chan string) {
    ch <- "Hello Kent!"
}
```
Concurrency in Go 🧪

Deadlock detection

Missing 'go' keyword

```go
import _ "net" // unused
func main() {
    ch := make(chan string)
    send(ch) // Oops
    print(<-ch)
    close(ch)
}

func send(ch chan string) {
    ch <- "Hello Kent"
}
```

Import unused, unrelated package
Concurrency in Go

Deadlock detection

- Only one (main) goroutine
- Send without receive - blocks

Output:

$ go run deadlock2.go

Hangs: Deadlock **NOT** detected
Our goal

Check liveness/safety properties in addition to global deadlocks

- Apply process calculi techniques to Go
- Use model checking to statically analyse Go programs
Concurrency in Go

Behavioural type inference

Abstract Go communication as Behavioural Types

1. Type inference
   SSA IR
   Go source code

2. Model checking
   mCRL2 model checker
   Check safety and liveness

3. Termination checking
   KITTeL termination prover
   Address type ↔ program gap

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida
A Static Verification Framework for Message Passing in Go using Behavioural Types

mrg.doc.ic.ac.uk
Infer Behavioural Types from Go program

**Go source code**

```go
func main() {
    ch := make(chan int)
    go send(ch)
    print(<-ch)
    close(ch)
}

func send(c chan int) {
    c <- 1
}
```

**Behavioural Types**

Types of CCS-like [Milner ’80] process calculus

- Send/Receive
- new (channel)
- parallel composition (spawn)

**Go-specific**

- Close channel
- Select (guarded choice)
Infer Behavioural Types from Go program

Go source code

```go
func main() {
    ch := make(chan int)
    go send(ch)
    print(<-ch)
    close(ch)
}

func send(c chan int) {
    c <- 1
}
```

Inferred Behavioural Types

```
main() = (new ch);
    (send(ch) | ch;
    close ch),

send(ch) = __ch
```

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida

A Static Verification Framework for Message Passing in Go using Behavioural Types
Infer Behavioural Types from Go program

Go source code

```
func main() {
    ch := make(chan int)
    go send(ch)
    print(<-ch)
    close(ch)
}

func send(c chan int) {
    c <- 1
}
```

Inferred Behavioural Types

```
create channel

main() = (new ch);
spawn
(send\( \langle ch\rangle \) | 
receive
ch;

send \( ch \) = \overline{ch}

close

close ch),
```

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida
A Static Verification Framework for Message Passing in Go using Behavioural Types
Infer Behavioural Types from Go program

```go
func main() {
    ch := make(chan int) // Create channel
    go sendFn(ch)        // Run as goroutine
    x := recvVal(ch)     // Function call
    for i := 0; i < x; i++ {
        print(i)
    }
    close(ch)            // Close channel
}

func sendFn(c chan int) { c <- 3 } // Send to c
func recvVal(c chan int) int { return <-c } //Recv from c
```
Infer Behavioural Types from Go program

package main

func main.main()

entry

0

\[ t0 = \text{make chan int 0:int} \]
\[ \text{go sendFn(t0)} \]
\[ t1 = \text{recvVal(t0)} \]
\[ \text{jump 3} \]

3

\[ t5 = \text{phi[0: 0:int, 1: t3] } \#i \]
\[ t6 = t5 < t1 \]
\[ \text{if t6 goto 1 else 2} \]

1

\[ t2 = \text{print(t5)} \]
\[ t3 = t5 + 1: int \]
\[ \text{jump 3} \]

2

\[ t4 = \text{close(t0)} \]
\[ \text{return} \]

func main.sendFn(c)

entry

0

\[ \text{send c <- 42:int} \]
\[ \text{return} \]

func main.recvVal(c)

entry

0

\[ t0 = <-c \]
\[ \text{return t0} \]

Analyse in

**Static Single Assignment**

SSA representation of input program

- Only inspect **communication** primitives
- Distinguish between unique channels
Model checking behavioural types

From behavioural types to model and property specification

1. Type inference
   SSA IR
   Go source code

2. Model checking
   mCRL2 model checker
   Check safety and liveness

3. Termination checking
   KITTeL termination prover
   Address type ↔ program gap
Model checking behavioural types

\[ M \models \phi \]

- **LTS model**: inferred type + type semantics
- **Safety/liveness properties**: \( \mu \)-calculus formulae for LTS
- Check with mCRL2 model checker
  - mCRL2 constraint: \textit{Finite control} (no spawning in loops)
- Global deadlock freedom
- Channel safety (no send/\texttt{close} on closed channel)
- Liveness (partial deadlock freedom)
- Eventual reception
Behavioural Types as LTS model

Standard CS semantics, i.e.

\[
\begin{align*}
\overline{a}; T & \xrightarrow{a} T \\
T \mid S & \xrightarrow{\tau a} T' \mid S'
\end{align*}
\]

Send on channel \( a \)

Synchronise on \( a \)

Receive on channel \( a \)
Behavioural Types as \textbf{LTS} model

Standard CS semantics, i.e.

\[
\begin{align*}
\bar{a}; T & \xrightarrow{\bar{a}} T \\
T & \xrightarrow{a} T' \\
S & \xrightarrow{a} S' \\
T \parallel S & \xrightarrow{\tau a} T' \parallel S'
\end{align*}
\]

Send on channel \( a \) \hspace{1cm} \textbf{Synchronise on} \( a \) \hspace{1cm} Receive on channel \( a \)
Specifying properties of model

Barbs (predicates at each state) describe property at state
- Concept from process calculi [Milner ’88, Sangiorgi ’92]
- $\mu$-calculus properties specified in terms of barbs

Barbs ($T \downarrow_o$)
- Predicates of state/type $T$
- Holds when $T$ is ready to fire action $o$
Specifying properties of model

\[ \overline{a}; T \downarrow a \]

- Ready to send

\[ \frac{T \downarrow \overline{a}}{T \mid T' \downarrow \tau a} \]

- Ready to synchronise

\[ a; T \downarrow a \]

- Ready to receive

Barbs \((T \downarrow_o)\):

- Predicates of state/type \(T\)

- Holds when \(T\) is ready to fire action \(o\)
Specifying properties of model

\[
\bar{a}; T \Downarrow \bar{a} \quad T \Downarrow \bar{a} \quad T' \Downarrow a \quad T' \Downarrow a \\
\]

Ready to send \quad Ready to synchronise \quad Ready to receive

Barbs \((T \Downarrow o)\):

- Predicates of state/type \( T \)
- Holds when \( T \) is ready to fire action \( o \)
Mi Go | Liveness / Safety

Channel Safety

- Channel is closed at most once.
- Can only input from a closed channel (default value).
- Others raise an error and crash.

Barb
[Milner 8
Sangiorgi 92]
Channel Safety

- Channel is closed at most once.
- Can only input from a closed channel (default value).
- Others raise an error and crash.

\[ P \text{ is channel safe if } P \rightarrow^* (v \bar{c}) Q \text{ and } Q \Downarrow \text{close}(a) \]

\[ \neg (Q \Downarrow \text{end}(a)) \land \neg (Q \Downarrow \bar{a}) \]

never closing

never send
Migo Liveness / Safety

- Liveness

All reachable actions are eventually performed

\[ P \text{ is live if } P \rightarrow^{* (v \epsilon)} Q \]

\[ Q \downarrow a \Rightarrow Q \downarrow z \text{ at } a \]

\[ Q \downarrow \overline{a} \Rightarrow Q \downarrow z \text{ at } a \]
Select

\[ P_1 = \text{select} \{ a!, \ b? , \ z.P \} \]

\[ P_2 = \text{select} \{ a!, \ b? \} \]

\[ R_1 = a? \]
Select

\[ P_1 = \text{select} \{ a!, b?, z \cdot P \} \]

\[ P_2 = \text{select} \{ a!, b? \} \quad R_1 = a? \]

Time Out

if \( P \) is live

\( P_1 \) is live
Select

\[ P_1 = \text{select} \{ a!, b? \} \]

\[ P_2 = \text{select} \{ a!, b? \} \quad R_1 = a? \]

Time Out

if \( P \) is live
\( P_1 \) is live
\( P_2 \) is not live
\( P_2 | R_2 \) is
Select

\[ P_1 = \text{select } \{ a!, b? \} \]

\[ P_2 = \text{select } \{ a!, b? \} \]

\[ R_1 = a? \]

Barb \( \downarrow \bar{a} \)

\[ \pi_i \downarrow Q_i \]

select \( \{ \pi_i, P_i \} \downarrow \bar{a} \)

\[ P \downarrow \bar{a} \]

\[ Q \downarrow \bar{a} \Rightarrow Q \downarrow \bar{2} \text{ at } \bar{a} \]

Time Out

if \( P \) is live

\( P_1 \) is live

\( P_2 \) is not live

\( P_2 \mid R_2 \) is
Specifying **properties** of model

Given

- **LTS model** from inferred behavioural types
- **Barbs** of the LTS model

Express **safety/liveness properties**

- As \(\mu\)-calculus formulae
- In terms of the **model** and the **barbs**

- Global deadlock freedom
- Channel safety (no send/\texttt{close} on closed channel)
- Liveness (partial deadlock freedom)
- Eventual reception
Property: Global deadlock freedom

\[ \left( \bigwedge_{a \in \mathcal{A}} \downarrow a \lor \downarrow \bar{a} \right) \implies \langle \mathcal{A} \rangle \text{true} \]

If a channel \( a \) is ready to receive or send, then there must be a next state (i.e. not stuck)

\( \mathcal{A} = \text{set of all initialised channels} \quad \mathcal{A} = \text{set of all labels} \)

\[ \Rightarrow \text{Ready receive/send} = \text{not end of program.} \]
Property: Global deadlock freedom

\[(\bigwedge_{a \in A} \downarrow a \lor \downarrow \overline{a}) \implies \langle A \rangle \text{true}\]

- **Send** (\(\downarrow \text{ch}: \text{line 10}\))
- No synchronisation
- No more reduction

```go
import _ "net" // unused
func main() {
    ch := make(chan string)
    send(ch) // Oops
    print(<-ch)
    close(ch)
}

func send(ch chan string) {
    ch <- "Hello Kent"
}
```
Property: Channel safety

\[(\bigwedge_{a \in A} \downarrow a^*) \implies \neg (\downarrow \bar{a} \lor \downarrow \text{clo } a)\]

Once a channel \(a\) is closed (\(a^*\)),
it will not be sent to, nor closed again (\(\text{clo } a\))
Property: Channel safety

\[ \bigwedge_{a \in A} \downarrow a^* \implies \neg (\downarrow \neg a \lor \downarrow \text{clo } a) \]

```go
func main() {
    ch := make(chan int)
    go func(ch chan int) {
        ch <- 1 // is ch closed?
    }(ch)
    close(ch)
    <-ch
}
```

- `\downarrow \text{clo } ch` when `close(ch)`
- `\downarrow \text{ch}^*` fires after closed
- Send (`\downarrow \text{ch}:` line 4)
Property: Liveness (partial deadlock freedom)

Liveness for Send/Receive

\[(\bigwedge_{a \in A} \downarrow a \lor \downarrow \bar{a}) \implies \text{eventually } (\langle \tau_a \rangle \text{true})\]

If a channel is ready to receive or send, then eventually it can synchronise \((\tau_a)\)

(i.e. there’s corresponding send for receiver/recv for sender)
Property: Liveness (partial deadlock freedom)

Liveness for Send/Receive

\[
\left( \bigwedge_{a \in A} \downarrow a \lor \downarrow \overline{a} \right) \implies \text{eventually} \left( \langle \tau_a \rangle \text{true} \right)
\]

where:

\[
\text{eventually} \ (\phi) \ \overset{\text{def}}{=} \ \mu y. \ (\phi \lor \langle A \rangle y)
\]

If a channel is ready to receive or send, then for some reachable state it can synchronise \((\tau_a)\)
Property: Liveness (partial deadlock freedom)

Liveness for Select

\[
( \bigwedge_{\tilde{a} \in \mathcal{P}(A)} \downarrow \tilde{a} ) \implies \text{eventually} ( \langle \{ \tau_a \mid a \in \tilde{a} \} \rangle \text{true} )
\]

If one of the channels in `select` is ready to receive or send, then `eventually` it will synchronise `\( \tau_a \)`
Property: Liveness (partial deadlock freedom)

Liveness for Select

\[(\bigwedge_{\tilde{a} \in \mathcal{P}(A)} \downarrow \tilde{a}) \implies \text{eventually } \langle \{ \tau_a | a \in \tilde{a} \} \rangle \text{true} \]

\[P_1 = \text{select}\{\overline{a}, b, \tau.P\}\]

\[P_2 = \text{select}\{\overline{a}, b\}\]

\[R_1 = a\]

\[P_1\text{ is live if } P\text{ is } \checkmark\]

\[P_2\text{ is not live } \times\]

\[(P_2 | R_1)\text{ is live } \checkmark\]
Property: Liveness (partial deadlock freedom)
Liveness for Select

\[
( \bigwedge_{\tilde{a} \in \mathcal{P}(A)} \downarrow \tilde{a} ) \implies \text{eventually} (\langle \{\tau_a \mid a \in \tilde{a}\} \rangle \text{true})
\]

\[
P_1 = \text{select}\{\overline{a}, \ b, \ \tau.P\}
\]
\[
P_2 = \text{select}\{\overline{a}, \ b\}
\]
\[
R_1 = a
\]

\[
P_1 \text{ is live if } P \text{ is } \checkmark
\]
\[
P_2 \text{ is not live } \times
\]
\[
(P_2 | R_1) \text{ is live } \checkmark
\]
Concurrency in Go

Behavioural type inference

Model checking behavioural types

Termination checking

Summary

Property: Liveness (partial deadlock freedom)

\((\bigwedge_a \downarrow a \lor \downarrow \bar{a}) \implies \text{eventually} (\langle \tau_a \rangle_{\text{true}})\)

\((\bigwedge_{\bar{a}} \downarrow \bar{a}) \implies \text{eventually} (\langle \{\tau_a \mid a \in \bar{a}\} \rangle_{\text{true}})\)

```go
1 func main() {
2     ch := make(chan int)
3     go looper() // !!!
4     <-ch // No matching send
5 }
6 func looper() {
7     for {
8     }
9 }
```

✗ Runtime detector: Hangs

✓ Our tool: NOT live

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida
A Static Verification Framework for Message Passing in Go using Behavioural Types
mrg.doc.ic.ac.uk
Property: Liveness (partial deadlock freedom)

\[
\left( \bigwedge_{a \in A} \downarrow a \lor \downarrow \overline{a} \right) \implies \text{eventually}\left( \langle \tau_a \rangle \text{true} \right)
\]

\[
\left( \bigwedge_{\tilde{a} \in \mathcal{P}(A)} \downarrow \tilde{a} \right) \implies \text{eventually}\left( \langle \{ \tau_a \mid a \in \tilde{a} \} \rangle \text{true} \right)
\]

```go
func main() {
    ch := make(chan int)
    go loopSend(ch)
    <-ch
}
func loopSend(ch chan int) {
    for i := 0; i < 10; i-- {
        // Does not terminate
    }
    ch <- 1
}
```

What about this one?

- Type: Live
- Program: NOT live

Needs additional guarantees
Property: Eventual reception

\( \left( \bigwedge_{a \in A} \downarrow a^\bullet \right) \implies \text{eventually} (\langle \tau_a \rangle \text{true}) \)

If an item is sent to a buffered channel \((a^\bullet)\),
Then \textbf{eventually} it can be consumed/synchronised \((\tau_a)\)

(i.e. no orphan messages)
Termination checking

Addressing the program-type abstraction gap

1. Type inference

SSA IR
Go source code

2. Model checking

mCRL2 model checker
Check safety and liveness

3. Termination checking

KITTeL termination prover
Address type ↔ program gap
Termination checking with KITTeL

Type inference does not consider *program data*

- Type liveness ≠ Program liveness if program non-terminating
- Especially when involving iteration
  ⇒ Check for loop termination
- If terminates, type liveness = program liveness

<table>
<thead>
<tr>
<th></th>
<th>Program terminates</th>
<th>Program does not terminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type live</td>
<td>✓ Program live</td>
<td>?</td>
</tr>
<tr>
<td>Type not live</td>
<td>✗ Program not live</td>
<td>✗ Program not live</td>
</tr>
</tbody>
</table>

Julien Lange, Nicholas Ng, Bernardo Toninho, Nobuko Yoshida
A Static Verification Framework for Message Passing in Go using Behavioural Types
Tool: Godel-Checker

https://github.com/nickng/gospal
https://bitbucket.org/MobilityReadingGroup/godel-checker

Understanding Concurrency with Behavioural Types
GolangUK Conference 2017
Conclusion

Verification framework based on Behavioural Types

- Behavioural types for Go concurrency
- Infer types from Go source code
- Model check types for safety/liveness
- + termination for iterative Go code
In the paper

See our paper for omitted topics in this talk:

- Behavioural type inference algorithm
- Treatment of buffered (asynchronous) channels
- The select (non-deterministic choice) primitive
- Definitions of behavioural type semantics/barbs

Table 3: Go programs verified by our framework and comparison with existing static deadlock detection tools.

<table>
<thead>
<tr>
<th>Programs</th>
<th>LoC</th>
<th># states</th>
<th>$\psi_g$</th>
<th>$\psi_I$</th>
<th>$\psi_s$</th>
<th>$\psi_e$</th>
<th>Godel Checker</th>
<th>dingo-hunter [36]</th>
<th>gopherlyzer [40]</th>
<th>Gofner/Gong [30]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Infer</td>
<td>Live</td>
<td>Live+CS</td>
<td>Term</td>
</tr>
<tr>
<td>mismatch</td>
<td>29</td>
<td>53</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>620.7</td>
<td>996.8</td>
<td>996.7</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>fixed</td>
<td>27</td>
<td>16</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>624.4</td>
<td>996.5</td>
<td>996.3</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>fanin</td>
<td>41</td>
<td>39</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>631.1</td>
<td>996.2</td>
<td>996.2</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>sieve</td>
<td>43</td>
<td>$\infty$</td>
<td>$n/a$</td>
<td>$-$</td>
<td>$-$</td>
<td>$n/a$</td>
<td>$n/a$</td>
<td>$n/a$</td>
<td>$n/a$</td>
<td>$n/a$</td>
</tr>
<tr>
<td>philo</td>
<td>41</td>
<td>65</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>6.1</td>
<td>996.5</td>
<td>996.6</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>dinephi13</td>
<td>55</td>
<td>3838</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>645.2</td>
<td>996.4</td>
<td>996.3</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>starvphel13</td>
<td>47</td>
<td>3151</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>628.2</td>
<td>996.5</td>
<td>996.5</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>sel</td>
<td>22</td>
<td>103</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>4.2</td>
<td>996.7</td>
<td>996.6</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>selfixed</td>
<td>22</td>
<td>20</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>4.9</td>
<td>996.3</td>
<td>996.4</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>jobsched</td>
<td>43</td>
<td>43</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>632.7</td>
<td>996.7</td>
<td>1996.1</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>forselect</td>
<td>42</td>
<td>26</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>623.3</td>
<td>996.4</td>
<td>996.3</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>cond-recur</td>
<td>37</td>
<td>12</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>4.9</td>
<td>996.2</td>
<td>996.2</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>concxsys</td>
<td>118</td>
<td>15</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>$\times$</td>
<td>549.7</td>
<td>996.5</td>
<td>996.4</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>alt-bit</td>
<td>70</td>
<td>112</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>634.4</td>
<td>996.3</td>
<td>996.3</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>prod-cons</td>
<td>28</td>
<td>106</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>4.1</td>
<td>996.4</td>
<td>1996.2</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>nonlive</td>
<td>16</td>
<td>8</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>630.1</td>
<td>996.6</td>
<td>996.5</td>
<td>timeout</td>
</tr>
<tr>
<td>double-close</td>
<td>15</td>
<td>17</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>3.5</td>
<td>996.6</td>
<td>1996.6</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>stuckmsg</td>
<td>18</td>
<td>4</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
<td>3.5</td>
<td>996.6</td>
<td>996.6</td>
<td>$\checkmark$</td>
</tr>
</tbody>
</table>
Future and related work

Extend framework to support more safety properties
Different verification approaches
- Godel-Checker model checking [ICSE’18] (this talk)
- Gong type verifier [POPL’17]
- Choreography synthesis [CC’15]
Different concurrency issues (e.g. data races)
Scribble is a language to describe application-level protocols among communicating systems. A protocol represents an agreement on how participating systems interact with each other. Without a protocol, it is hard to do meaningful interaction: participants simply cannot communicate effectively, since they do not know when to expect the other parties to send data, or whether the other party is ready to receive data. However, having a description of a protocol has further benefits. It enables verification to ensure that the protocol can be implemented without resulting in unintended consequences, such as deadlocks.

**Describe**
Scribble is a language for describing multiparty protocols from a global, or endpoint neutral, perspective.

**Verify**
Scribble has a theoretical foundation, based on the Pi Calculus and Session Types, to ensure that protocols described using the language are sound, and do not suffer from deadlocks or livelocks.

**Project**
Endpoint projection is the term used for identifying the responsibility of a particular role (or endpoint) within a protocol.

**Implement**
Various options exist, including (a) using the endpoint projection for a role to generate a skeleton code, (b) using session type APIs to clearly describe the behaviour, and (c) statically verify the code against the projection.

**Monitor**
Use the endpoint projection for roles defined within a Scribble protocol, to monitor the activity of a particular endpoint, to ensure it correctly implements the expected behaviour.
Scribble-Go workflow

1. Write a role-parametric global protocol
2. Select endpoint role variant to implement (e.g. Fetcher)
3. Use Scribble-Go to project and generate Endpoint API
4. Implement endpoint (e.g. Fetcher[3]) using the Endpoint API
Role variant are *unique kinds* of endpoints

\{ M, F[1..n], Server \}

If \( F[1] \) sends an extra request

HTTP HEAD to Server to get total size

Then acts as a normal \( F \)

The role variants are:

\{ M, F[1], F[2..n], Server \}

\( \rightarrow \) \( F[1] \) and \( F[2..n] \) are different endpoints

Inference of role variants (indices): formulated as SMT constraints for Z3
Endpoint API generation and usage

FSMs from local protocols → Message passing API

- Fluent-style
  - Every state is a unique type (struct)
  - Method calls (communication) returns next state
- Type information can be leveraged by IDEs
  - “dot-driven” content assist & auto complete
Behavioural Types for Go

Type syntax

\[
\begin{align*}
\alpha & ::= \overline{u} \mid u \mid \tau \\
T, S & ::= \alpha; T \mid T \oplus S \mid &\{\alpha_i; T_i\}_{i \in I} \mid (T \mid S) \mid 0 \\
& \mid (\text{new } a)T \mid \text{close } u; T \mid t\langle \overline{u} \rangle \mid u \mid buf[u]_{\text{closed}} \\
T & ::= \{t(\tilde{y}_i) = T_i\}_{i \in I} \text{ in } S
\end{align*}
\]

- Types of a CCS-like process calculus
- Abstracts Go concurrency primitives
  - Send/Recv, new (channel), parallel composition (spawn)
  - Go-specific: Close channel, Select (guarded choice)
Semantics of types

\[
\begin{align*}
\text{SND} & \quad \overline{a}; T \xrightarrow{\overline{\overline{a}}} T \\
\text{RCV} & \quad a; T \xrightarrow{\overline{a}} T \\
\text{TAU} & \quad \tau; T \xrightarrow{\tau} T \\
\text{END} & \quad \text{close } a; T \xrightarrow{\text{clo } a} T \\
\text{BUF} & \quad |a|_k^n \xrightarrow{\text{clo } a} \text{buf}[a]_{\text{closed}} \\
\text{CLD} & \quad \text{buf}[a]_{\text{closed}} \xrightarrow{a^*} \text{buf}[a]_{\text{closed}} \\
\text{SEL} & \quad i \in \{1, 2\} \quad T_1 \oplus T_2 \xrightarrow{\tau} T_i \\
\text{BRA} & \quad \alpha_j; T_j \xrightarrow{\alpha_j} T_j \quad j \in I \\
\text{PAR} & \quad T \xrightarrow{\alpha} T' \quad T \mid S \xrightarrow{\alpha} T' \mid S \\
\text{SEQ} & \quad T; S \xrightarrow{\alpha} T'; S \\
\text{TERM} & \quad 0; S \xrightarrow{\tau} S \\
\text{COM} & \quad \alpha \in \{\overline{a}, a^*, a^\bullet\} \\
\text{EQ} & \quad T \equiv_{\alpha} T' \quad T \xrightarrow{\alpha} T'' \\
\text{DEF} & \quad T \{\overline{a}/\overline{x}\} \xrightarrow{\alpha} T' \quad t(\overline{x}) = T \\
\text{CLOSE} & \quad T \xrightarrow{\text{clo } a} T' \quad S \xrightarrow{\text{clo } a} S' \quad T \mid S \xrightarrow{\tau} T' \mid S' \\
\text{IN} & \quad k < n \quad |a|^n_k \xrightarrow{\cdot a} |a|^n_{k+1} \\
\text{OUT} & \quad k \geq 1 \quad |a|^n_k \xrightarrow{\cdot a^*} |a|^n_{k-1}
\end{align*}
\]
Barb predicates for types

\[
\begin{align*}
T \downarrow_a & \quad \text{close } T \downarrow_{\text{cl}o} a \\
\bar{a}; T \downarrow_{\bar{a}} & \quad \text{buf}[a]\text{closed} \downarrow_{a^*} \\
T \downarrow_{\text{o}} & \quad T \downarrow_a T' \downarrow_{\bar{a}} \text{ or } T' \downarrow_{a^*} \\
& \quad T \{\bar{a}/\bar{x}\} \downarrow_{\text{t}} \text{t}(\bar{x}) = T \\
& \quad T \{\bar{a}/\bar{x}\} \downarrow_{\text{o}} \\
& \quad \text{t}(\bar{a}) \downarrow_{\text{o}} \\
T \downarrow_a & \quad \alpha_i \downarrow_{a} \\
\{\alpha_i; S_i\}_{i \in I} \downarrow_{\tau_a} & \quad T \downarrow_{\text{a}} \text{ or } T \downarrow_{a^*} \quad \alpha_i \downarrow_a \\
& \quad \alpha_i \downarrow_{a} \\
k < n & \quad \text{or } k \geq 1 \\
[a]_k^{n} \downarrow_{a} & \quad \alpha_i \downarrow_{a} \\
[a]_{a}^{n} \downarrow_{a^*} & \quad \alpha_i \downarrow_{a} \\
T \downarrow_{a} T' \downarrow_{\tau_a} & \quad \alpha_i \downarrow_{a} \\
& \quad \alpha_i \downarrow_{a} \\
T \downarrow_{\text{o}} a \notin \text{fn(o)} & \quad (\text{new}^n a); T \downarrow_{\text{o}} \\
& \quad T \equiv T' \\
& \quad T \downarrow_{\text{o}} \\
\end{align*}
\]

Figure: Barb predicates for types.