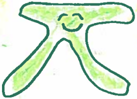
OPEN PROBLEMS OF SESSION TYPES



Us ∈ Mobility Research Group

MobilityReadingGroup

π-calculus, Session Types research at Imperial College

Home | People | Publications | Grants | Talks | Tools | Awards | Kohei Honda

NEWS

Our recent work Fencing off Go: Liveness and Safety for Channel-based Programming was summarised on The Morning Paper blog.

2 Feb 2017

Weizhen passed her viva today, congratulations Dr. Yang!

24 Jan 2017

Mariangiola Dezani-Ciancaglini, a long-term collaborator with our group working on Session Types turns 70 today, more details here.

23 Dec 2016

Rumyana passed her viva today,

SELECTED PUBLICATIONS

2017

Raymond Hu , Nobuko Yoshida : Explicit Connection Actions in Multiparty Session Types. *To appear in* FASE 2017 .

Julien Lange , Nicholas Ng , Bernardo Toninho , Nobuko Yoshida : Fencing off Go: Liveness and Safety for Channel-based Programming. POPL 2017 .

Rumyana Neykova , Nobuko Yoshida : Let It Recover: Multiparty Protocol-Induced Recovery. CC 2017 .

Julien Lange , Nobuko Yoshida : On the Undecidability of Asynchronous Session Subtyping. *To appear in* FoSSaCS 2017 .

http://mrg.doc.ic.ac.uk/

Academic Staff

Nobuko Yoshida

Research Associate

Raymond Hu

Julien Lange

Nicholas Ng

Xinyu Niu

Alceste Scalas

Bernardo Toninho

PhD Student

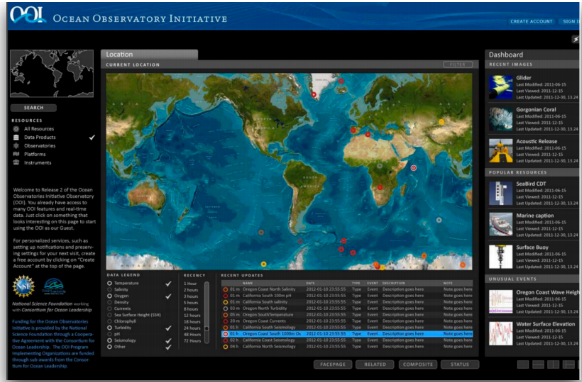
Assel Altayeva

Juliana Franco

Rumyana Neykova

Weizhen Yang

OOI Collaboration



- **TCS'16:** Monitoring Networks through Multiparty Session Types. Laura Bocchi , Tzu-Chun Chen , Romain Demangeon , Kohei Honda , Nobuko Yoshida
- **LMCS'16**: Multiparty Session Actors. Rumyana Neykova, Nobuko Yoshida
- **FMSD'15:** Practical interruptible conversations: Distributed dynamic verification with multiparty session types and Python. Romain Demangeon , Kohei Honda , Raymond Hu , Rumyana Neykova , Nobuko Yoshida
- **TGC'13:** The Scribble Protocol Language. Nobuko Yoshida , Raymond Hu , Rumyana Neykova , Nicholas Ng

www.scribble.org

Home Getting Started Downloads Documentation ▾ Community ▾

Scribble: Describing Multi Party Protocols

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

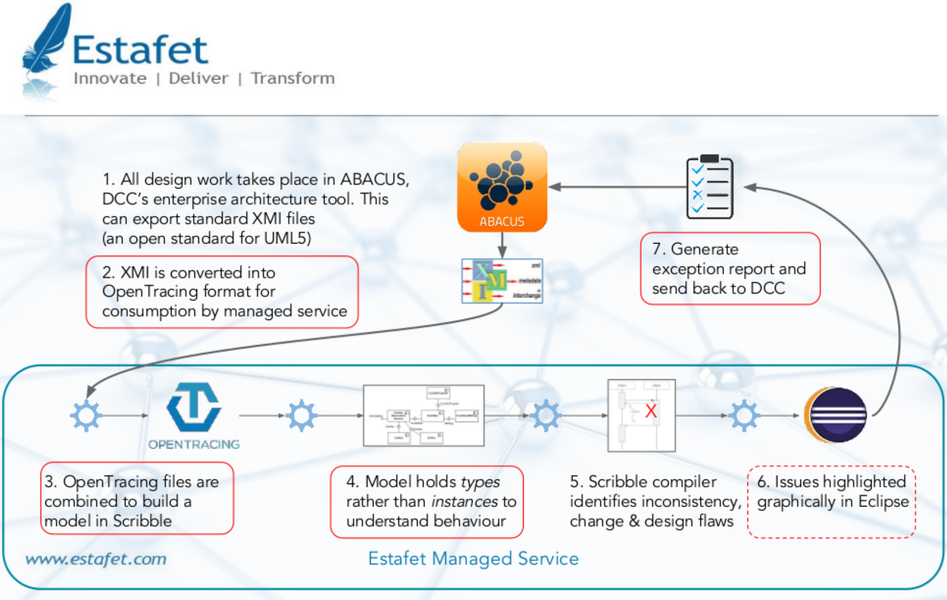
Use the endpoint projection for roles defined within a Scribble protocol, to monitor the activity of a particular endpoint, to ensure it correctly implements the expected behaviour.

Online tool : http://scribble.doc.ic.ac.uk/

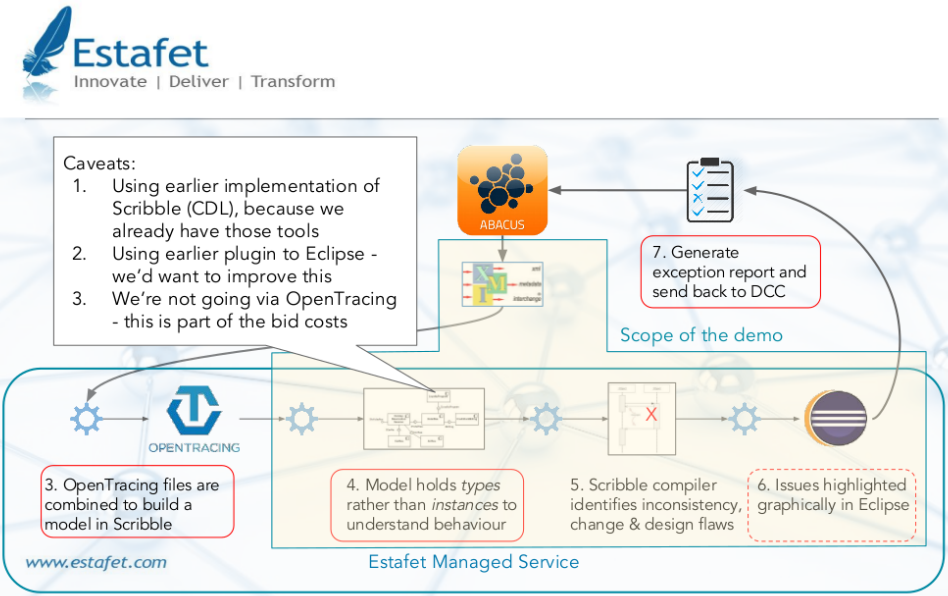
```
module examples;

global protocol HelloWorld(role Me, role World) {
  hello() from Me to World;
  choice at World {
    goodMorning1() from World to Me;
  } or {
    goodMorning1() from World to Me;
  }
}

```

Load a sample | Check | Protocol: examples.HelloWorld | Role: Me | Project | Generate Graph

End-to-End Switching Programme by DCC



End-to-End Switching Programme by DCC



Interactions with Industries

Adam Bowen @adamnbowen · Sep 15
I didn't even know that session types existed an hour ago, but thanks to Nobuko Yoshida's great talk at #pwlconf, I want to learn more.

Nobuko Yoshida
Imperial College, London

DoC researcher to speak at Golang UK conference

by Vicky Kapogianni
20 July 2016

DoC researcher to speak at industry-focused Golang UK conference on results of concurrency research

Click here to add content

.@nicholascwng rocking on @GolangUKconf about static deadlock detection in #golang #gouk16

Interactions with Industries

F#unctional Londoners Meetup Group

6 days ago · 6:30 PM

Session Types with Fahd Abdeljallal

43 Members

Synopsis: Session types are a formalism to codify the structure of a communication, using types to specify the communication protocol used. This formalism provides the... LEARN MORE

Distributed Systems vs. Compositionality

Dr. Roland Kuhn
@rolandkuhn — *CTO of Actyx*

actyx

Current State

- behaviors can be composed both sequentially and concurrently
- effects are not yet tracked
- Scribble generator for Scala not yet there
- theoretical work at Imperial College, London (Prof. Nobuko Yoshida & Alceste Scalas)

Selected Publications 2016/2017

- **[ECOOP'17]** Alceste Scala, Raymond Hu, Ornela Darda, NY: A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming..
- **[COORDINATION'17]** Keigo Imai, NY and Shoji Yuen: Session-ocaml: a session-based library with polarities and lenses.
- **[FoSSaCS'17]** Julien Lange , NY: On the Undecidability of Asynchronous Session Subtyping.
- **[FASE'17]** Raymond Hu , NY: Explicit Connection Actions in Multiparty Session Types.
- **[CC'17]** Rumyana Neykova , NY: Let It Recover: Multiparty Protocol-Induced Recovery.
- **[POPL'17]** Julien Lange , Nicholas Ng , Bernardo Toninho , NY: Fencing off Go: Liveness and Safety for Channel-based Programming.
- **[FPL'16]** Xinyu Niu , Nicholas Ng , Tomofumi Yuki , Shaojun Wang , NY, Wayne Luk : EURECA Compilation: Automatic Optimisation of Cycle-Reconfigurable Circuits.
- **[ECOOP'16]** Alceste Scala, NY: Lightweight Session Programming in Scala
- **[CC'16]** Nicholas Ng, NY: Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis.
- **[FASE'16]** Raymond Hu, NY: Hybrid Session Verification through Endpoint API Generation.
- **[TACAS'16]** Julien Lange, NY: Characteristic Formulae for Session Types.
- **[ESOP'16]** Dimitrios Kouzapas, Jorge A. Pérez, NY: On the Relative Expressiveness of Higher-Order Session Processes.
- **[POPL'16]** Dominic Orchard, NY: Effects as sessions, sessions as effects .

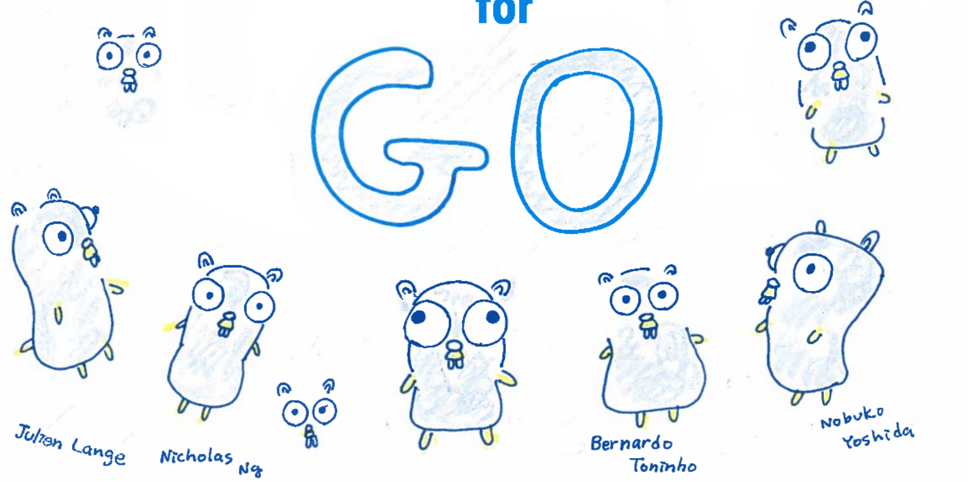
Selected Publications 2016/2017

- [ECOOP'17] Alceste Scala, Raymond Hu, Ornela Darda, NY :A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming.
- [COORDINATION'17] Keigo Imai, NY and Shoji Yuen: Session-ocaml: a session-based library with polarities and lenses.
- [FoSSaCS'17] Julien Lange , NY : On the Undecidability of Asynchronous Session Subtyping.
- [FASE'17] Raymond Hu , NY : Explicit Connection Actions in Multiparty Session Types.
- [CC'17] Rumyana Neykova , NY: Let It Recover: Multiparty Protocol-Induced Recovery.
- [POPL'17] Julien Lange , Nicholas Ng , Bernardo Toninho , NY: Fencing off Go: Liveness and Safety for Channel-based Programming.
- [FPL'16] Xinyu Niu , Nicholas Ng , Tomofumi Yuki , Shaojun Wang , NY, Wayne Luk: EURECA Compilation: Automatic Optimisation of Cycle-Reconfigurable Circuits.
- [ECOOP'16] Alceste Scala, NY: Lightweight Session Programming in Scala
- [CC'16] Nicholas Ng, NY: Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis.
- [FASE'16] Raymond Hu, NY: Hybrid Session Verification through Endpoint API Generation.
- [TACAS'16] Julien Lange, NY: Characteristic Formulae for Session Types.
- [ESOP'16] Dimitrios Kouzapas, Jorge A. Pérez, NY: On the Relative Expressiveness of Higher-Order Session Processes.
- [POPL'16] Dominic Orchard, NY: Effects as Sessions, Sessions as Effects.

HOW to

- deriver theories to practices
- make theories understandable
- meet theoretical challenges (concurrency · distributions)
- communicate people

Behavioural Type-Based Static Verification Framework for Go



Go concurrency verification research at DoC grabs headline

A paper by DoC researchers at POPL on Go concurrency verification was featured in a tech blog and generates a buzz outside of the research community.

A paper by researchers at the department was recently featured in the morning paper, a blog by venture capitalist Adrian Colye, which summarises an important, influential, topical or otherwise interesting paper in the field of computer science every weekday in an easily digestible way by non-researchers. On the 2 Feb 2017 issue of the morning paper, It was highlighted as "the true spirit of POPL (Principles of Programming Languages)".

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 — gorounines
  - Selective send / receive

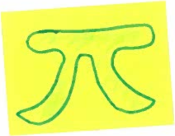
CSP 80'

FUN Dropbox, Netfix, 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

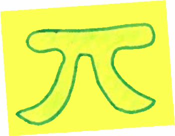
- Dynamic channel creations, unbounded thread creations, recursions, ..
- Scalable (synchronous/asynchronous) Modular, Refinable

- **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

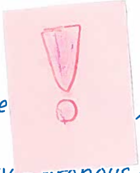


- Dynamic channel creations, unbounded thread creations, recursions, ..
- Scalable (synchronous/asynchronous) Modular, Refinable

- GO has a runtime deadlock detector
- How can we detect partial deadlock and channel errors for realistic programs?

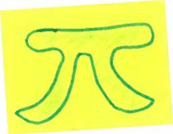


- Use behavioural types in process calculi
  e.g. [ACM Survey, 2016] **185** citations, 6 pages



- Dyn ne , unbounded thread creations, recursions,..
- Scalable (sy chronous/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

186 ??

- hannel 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 creatio… ons, ..
- Scalable (synchronous/asynchronous) Modular, Refinable

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

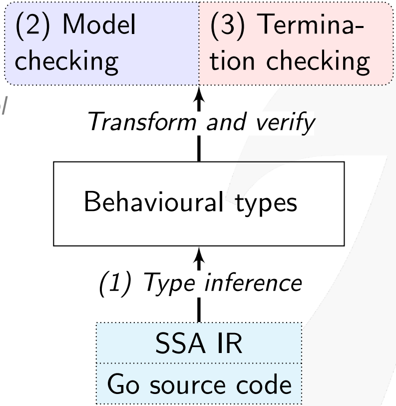
- Relate Safety / Liveness of Behavioural Types and GO Programs
  - 3 Classes [POPL'17]
  - Termination Check

Verification framework for Go

Overview

Check safety and liveness

Create input model and formula



Address type and process gap

Pass to termination prover

Concurrency in Go

```
func main() {
        ch, done := make(chan int), make(chan int)
        go send(ch) // Spawn as goroutine.
        go func() {
                for i := 0; i < 2; i++ {
                        print("Working...")
                }
        }()
        go recv(ch, done)
        go recv(ch, done)             // Who is ch receiving from?
        print("Done:", <-done, <-done) // 2 receivers, 2 replies
}
func send(ch chan int)   { ch <- 1 }    // Send to channel.
func recv(in, out chan int) { out <- <-in } // Fwd in to out.
```

- Send/receive blocks goroutines if channel full/empty resp.
- Close a channel `close(ch)`
- Guarded choice `select { case <-ch:; case <-ch2: }`

Concurrency in Go

Deadlock detection

```
func main() {
        ch, done := make(chan int), make(chan int)
        go send(ch) // Spawn as goroutine.
        go func() {
                for i := 0; i < 2; i++ {
                        print("Working...")
                }
        }()
        go recv(ch, done)
        go recv(ch, done)            // Who is ch receiving from?
        print("Done:", <-done, <-done) // 2 receivers, 2 replies
}
func send(ch chan int)    { ch <- 1 }    // Send to channel.
func recv(in, out chan int) { out <- <-in } // Fwd in to out.
```

Run program:

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

Concurrency in Go

Deadlock detection

```
func main() {
        ch, done := make(chan int), make(chan int)
        go send(ch) // Spawn as goroutine.
        go func() {
                for i := 0; ; i++ { // infini
                        print("Working...")
                }
        }()
        go recv(ch, done)
        go recv(ch, done)               // Who is ch receiving from?
        print("Done:", <-done, <-done) // 2 receivers, 2 replies
}
func send(ch chan int)    { ch <- 1 }    // Send to channel.
func recv(in, out chan int) { out <- <-in } // Fwd in to out.
```

Change to infinite

Concurrency in Go

Deadlock detection

```
func main() {
        ch, done := make(chan int), make(chan int)
        go send(ch) // Spawn as goroutine.
        go func() {
                for i := 0; ;  i++ { // infini
                        print("Working...")
                }
        }()
        go recv(ch, done)
        go recv(ch, done)            // Who is ch receiving from?
        print("Done:", <-done, <-done) // 2 receivers, 2 replies
}
func send(ch chan int)    { ch <- 1 }    // Send to channel.
func recv(in, out chan int) { out <- <-in } // Fwd in to out.
```

Change to infinite

Deadlock **NOT** detected (some goroutines are running)

Concurrency in Go

Deadlock detection

- Go has a runtime deadlock detector, panics (crash) if deadlock
- Deadlock if all goroutines are blocked
- Some packages (e.g. `net` for networking) **disables** it

```
import _ "net"  // Load "net" p
func main() {
        ch := make(chan int)
        send(ch)
        print(<-ch)
}
func send(ch chan int) { ch <- 1 }
```

Add benign import

Concurrency in Go

Deadlock detection

- Go has a runtime deadlock detector, panics (crash) if deadlock
- Deadlock if all goroutines are blocked
- Some packages (e.g. `net` for networking) **disables** it

```
import _ "net"   // Load "net" p
func main() {
        ch := make(chan int)
        send(ch)
        print(<-ch)
}
func send(ch chan int) { ch <- 1 }
```

Add benign import

Deadlock **NOT** detected

Go Programs as Processes

Go Program

$$P, Q \quad := \quad \pi; P \qquad\qquad \pi := u!\langle e\rangle \mid u?(y) \mid \tau$$

Go Programs as Processes

Go Program

$$P, Q \;:=\; \pi; P \;\mid\; \textsf{close}\ u; P \qquad\qquad \pi := u!\langle e\rangle \mid u?(y) \mid \tau$$

Go Programs as Processes

Go Program

$$P, Q \quad := \quad \pi; P \quad | \quad \mathsf{close}\ u; P \quad | \quad \mathsf{select}\{\pi_i; P_i\}_{i \in I}$$

$$\pi := u!\langle e \rangle \mid u?(y) \mid \tau$$

Go Programs as Processes

Go Program

$$
\begin{array}{rcl}
P, Q & := & \pi; P \\
& \mid & \text{close } u; P \\
& \mid & \text{select}\{\pi_i; P_i\}_{i \in I} \\
& \mid & \text{if } e \text{ then } P \text{ else } Q
\end{array}
\qquad
\pi := u!\langle e \rangle \mid u?(y) \mid \tau
$$

Go Programs as Processes

Go Program

$$
\begin{array}{rcl}
P, Q & := & \pi; P \\
& \mid & \text{close } u; P \\
& \mid & \text{select}\{\pi_i; P_i\}_{i \in I} \\
& \mid & \text{if } e \text{ then } P \text{ else } Q \\
& \mid & \text{newchan}(y{:}\sigma); P
\end{array}
$$

$$\pi := u!\langle e \rangle \mid u?(y) \mid \tau$$

Go Programs as Processes

Go Program

$$P, Q \;:=\; \pi; P \;\mid\; \text{close } u; P \;\mid\; \text{select}\{\pi_i; P_i\}_{i \in I} \;\mid\; \text{if } e \text{ then } P \text{ else } Q \;\mid\; \text{newchan}(y{:}\sigma); P \;\mid\; P \mid Q \;\mid\; \mathbf{0} \;\mid\; (\nu c)P$$

$$\pi := u!\langle e\rangle \mid u?(y) \mid \tau$$

Go Programs as Processes

Go Program

$$
\begin{array}{lcl}
P, Q & := & \pi; P \\
& \mid & \mathsf{close}\ u; P \\
& \mid & \mathsf{select}\{\pi_i; P_i\}_{i \in I} \\
& \mid & \mathsf{if}\ e\ \mathsf{then}\ P\ \mathsf{else}\ Q \\
& \mid & \mathsf{newchan}(y{:}\sigma); P \\
& \mid & P \mid Q \mid \mathbf{0} \mid (\nu c)P \\
& \mid & X\langle \tilde{e}, \tilde{u} \rangle \\
D & := & X(\tilde{x}) = P \\
\mathbf{P} & := & \{D_i\}_{i \in I}\ \mathsf{in}\ P
\end{array}
$$

$$\pi := u!\langle e \rangle \mid u?(y) \mid \tau$$

Go Programs as Processes

Go Program

$$
\begin{array}{lcl}
P, Q & := & \pi; P \\
 & \mid & \text{close } u; P \\
 & \mid & \text{select}\{\pi_i; P_i\}_{i \in I} \\
 & \mid & \text{if } e \text{ then } P \text{ else } Q \\
 & \mid & \text{newchan}(y{:}\sigma); P \\
 & \mid & P \mid Q \;\mid\; \mathbf{0} \;\mid\; (\nu c)P \\
 & \mid & X\langle \tilde{e}, \tilde{u} \rangle \\
D & := & X(\tilde{x}) = P \\
\mathbf{P} & := & \{D_i\}_{i \in I} \text{ in } P
\end{array}
$$

$$\pi := u!\langle e \rangle \mid u?(y) \mid \tau$$

Abstracting Go with Behavioural Types

Types

$$
\begin{array}{rcl}
\alpha & := & \overline{u} \mid u \mid \tau \\
T, S & := & \alpha; T \mid T \oplus S \mid \&\{\alpha_i; T_i\}_{i \in I} \mid (T \mid S) \mid \mathbf{0} \\
 & \mid & (\texttt{new}\, a) T \mid \texttt{close}\, u; T \mid \mathbf{t}\langle \tilde{u} \rangle \\
\mathbf{T} & := & \{\mathbf{t}(\tilde{y}_i) = T_i\}_{i \in I} \texttt{ in } S
\end{array}
$$

- 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)

MiGo Liveness / Safety

$P \Downarrow a$

Barb
[Milner & 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

MiGo Liveness / Safety

$P \Downarrow a$

Barb [Milner & 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 is **channel safe** if $P \rightarrow^{*} (\nu \tilde{c}) Q$ and $Q \downarrow close(a)$

a closed

$\neg (Q \downarrow end(a)) \wedge \neg (Q \Downarrow \bar{a})$

never closing

never send

Migo Liveness / Safety

- **Liveness**

All reachable actions are eventually performed

P is **live** if $P \xrightarrow{*}_{(\nu \tilde{c})} Q$

$Q \downarrow a \Rightarrow Q \Downarrow \tau$ at a

$Q \downarrow \bar{a} \Rightarrow Q \Downarrow \tau$ at a

Reduction (tau) at a

Select

P_1 = select $\{a!,\ b?,\ z.P\}$

P_2 = select $\{a!,\ b?\}$

$R_1 = a?$

Time Out

if P is live

P_1 is live

P_2 is <u>not</u> live

$P_2 | R_2$ is

Select

P_1 = select $\{a!,\ b?,\ z.P\}$

P_2 = select $\{a!,\ b?\}$

$R_1 = a?$

Time Out

if P is live
P_1 is live

P_2 is not live
$P_2 | R_2$ is

Barb $\downarrow \tilde{a}$

$$\frac{\pi_i \downarrow a_i}{\text{select}\ \{\pi_i .\ P_i\} \downarrow \tilde{a}} \qquad \frac{P \downarrow \tilde{a} \qquad Q \downarrow \bar{a_i}}{P \mid Q \downarrow [a_i]}$$

Liveness $Q \downarrow \tilde{a} \Rightarrow Q \Downarrow z$ at a_i

Verification framework for Go

Model checking with mCRL2

Generate LTS model and formulae from types

- Finite control (no parallel composition in recursion)
- Properties (formulae for model checker):
  - ✓ Global deadlock
  - ✓ Channel safety (no send/`close` on closed channel)
  - ✗ Liveness (partial deadlock)
  - ✗ Eventual reception
    - Require additional guarantees

the μ-calculus

encoding properties with barbs

Global Deadlock

$$\bigwedge_{a \in C} (\downarrow a \vee \downarrow \bar{a}) \Rightarrow \langle \alpha \rangle T$$

Channel Safety

$$\bigwedge_{a \in C} \downarrow \text{close } a \Rightarrow \neg (\downarrow \bar{a} \vee \downarrow \text{close } a)$$

Liveness

$$\bigwedge_{a \in C} (\downarrow a \vee \downarrow \bar{a}) \Rightarrow \Phi (\langle [a] \rangle T) \wedge$$

$$\bigwedge_{\tilde{a} \in C^n} \downarrow \tilde{a} \Rightarrow \Phi (\bigvee_{a \in \tilde{a}} \langle [a] \rangle T)$$

[Lange & NY TACAS'17]

Verification framework for Go

Termination checking with KITTeL

- Extracted types do not consider *data* in process
- Type liveness != program liveness
  - Especially when involving iteration
  - Check for loop termination
- Properties:
  - ✓ Global deadlock
  - ✓ Channel safety (no send/`close` on closed channel)
  - ✓ Liveness (partial deadlock)
  - ✓ Eventual reception

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

- Type: Live
- Program: NOT live

Relating Programs and Types

Program

$$F(n,i,o) \triangleq i?(x); \text{if } (x\%n \neq 0) \text{ then } o!\langle x\rangle; F\langle n,i,o\rangle \text{ else } F\langle n,i,o\rangle$$

Type

$$\textbf{\textit{filter}}(i,o) \triangleq i; (\overline{o}; \mathbf{t}_F\langle i,o\rangle \oplus \mathbf{t}_F\langle i,o\rangle)$$

- Identify 3 classes (Liveness)
  1. May Terminate
  2. Without infinitely running conditionals
  3. Non-deterministic conditional
- Channel Safety Programs = Types

Relating Programs and Types

Program

$F(n,i,o) \triangleq i?(x); \text{if } (x\%n$ … $o!\langle x\rangle; F\langle n,i,o\rangle \text{ else } F\langle n,i,o\rangle$

Type

$\textbf{filter}(i,o) \triangleq i; (\overline{o}; \mathsf{t}_F\langle i,o\rangle \oplus \mathsf{t}_F\langle i,o\rangle)$

- Identify 3 classes (Liveness)
  1. May Terminate
  2. Without infinitely running conditionals
  3. Non-deterministic conditional

Undecidable but use Termination, Fair check...

- Channel Safety Programs = Types

Verification framework for Go

Termination checking with KITTeL

- Extracted types do not consider *data* in process
- Type liveness != program liveness
  - Especially when involving iteration
  - Check for loop termination
- Properties:
  - ✓ Global deadlock
  - ✓ Channel safety (no send/`close` on closed channel)
  - ✓ Liveness (partial deadlock)
  - ✓ Eventual reception

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

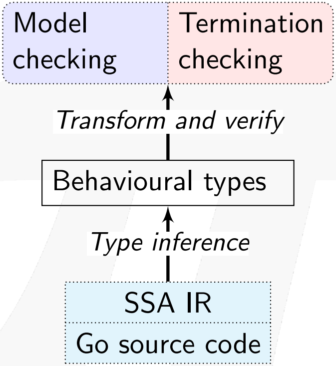
- Type: Live
- Program: NOT live

Tool demo

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



Future work

- Extend framework to support more properties
- Unlimited possibilities!
  - Different verification techniques
    - e.g. [POPL'17], Choreography synthesis [CC'15]
  - Different concurrency issues
    - Other synchronisation mechanisms
    - Race conditions

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

		Godel Checker									dingo-hunter [35]		gopherlyzer [39]		GoInfer/Gong [30]		
	Programs	# states	ψ_g	ψ_l	ψ_s	ψ_e	Infer	Live	Live+CS	Term	Live	Time	DF	Time	Live	CS	Time
1	mismatch [35]	53	×	×	✓	✓	620.68	996.79	996.67	✓	×	639.40	×	3956.41	×	✓	616.78
2	fixed [35]	16	✓	✓	✓	✓	624.41	996.50	996.34	✓	✓	603.18	✓	3166.26	✓	✓	609.95
3	fanin [35, 38]	39	✓	✓	✓	✓	631.12	996.15	996.23	✓	✓	607.98	✓	19.76	✓	✓	696.65
4	sieve [30, 35]	∞			*n/a*		-	-	-	*n/a*	*n/a*	-	*n/a*	-	✓	✓	778.29
5	philo [39]	65	×	×	✓	✓	6.10	996.51	996.56	✓	×	34.23	×	26.99	×	✓	16.84
6	dinephil3 [13, 32]	3838	✓	✓	✓	✓	645.15	996.42	996.31	✓	*n/a*	-	*n/a*	-	✓	✓	13.2 min
7	starvephil3	3151	×	×	✓	✓	628.20	996.50	996.46	✓	*n/a*	-	*n/a*	-	×	✓	3.5 min
8	sel [39]	103	×	×	✓	✓	4.23	996.70	996.61	✓	×	15.31	×	13.04	×	✓	50.46
9	selFixed [39]	20	✓	✓	✓	✓	4.02	996.33	996.39	✓	✓	14.93	✓	3168.32	✓	✓	13.08
10	jobsched [30]	43	✓	✓	✓	✓	632.67	996.69	1996.14	✓	*n/a*	-	✓	4753.56	✓	✓	635.20
11	forselect [30]	26	✓	✓	✓	✓	623.31	996.36	996.38	✓	✓	611.79	*n/a*	-	✓	✓	618.57
12	cond-recur [30]	12	✓	✓	✓	✓	3.95	996.21	996.22	✓	✓	9.40	*n/a*	-	✓	✓	14.74
13	concsys [41]	15	×	×	✓	✓	549.69	996.50	996.40	✓	*n/a*	-	×	5278.59	×	✓	521.26
14	alt-bit [30, 34]	112	✓	✓	✓	✓	634.43	996.34	996.26	✓	*n/a*	-	*n/a*	-	✓	✓	916.81
15	prod-cons	106	✓	×	✓	✓	4.10	996.37	1996.24	✓	×	10.15	×	30.10	×	✓	21.84
16	nonlive	8	✓	✓	✓	✓	630.10	996.55	996.47	timeout	⊗	613.62	*n/a*	-	⊗	✓	613.79
17	double-close	17	✓	✓	×	✓	3.48	996.58	1996.62	✓	⊠	8.68	⊠	11.83	✓	×	9.13
18	stuckmsg	4	✓	✓	✓	×	3.45	996.58	996.60	✓	*n/a*	-	*n/a*	-	✓	✓	7.55
19	dinephil5	~1M	✓	✓	✓	✓	626.45	41194.18	41408.00	✓	*n/a*	-	*n/a*	-	timeout		>48 hrs
20	prod3-cons3	57493	✓	✓	✓	✓	465.09	40859.24	40902.06	✓	*n/a*	-	*n/a*	-	timeout		>48 hrs
21	async-prod-cons	164897	✓	✓	✓	✓	4.29	47720.30	89414.60	✓	*n/a*	-	*n/a*	-	timeout		>48 hrs
22	astranet [26]	1160	✓	✓	✓	✓	2512.54	70399.00	75043.00	✓	*n/a*	-	*n/a*	-	*n/a*		-

CS: Channel Safe, Term: Termination check, DF: Deadlock-free, timeout: Termination check timeout (likely does not terminate), ⊠: False Alarm, ⊗: Undetected liveness error.

most programs use traditional imperative control flow features such as for loops, for-range loops (i.e. loops over a fixed finite data structure) and for-select loops (i.e. an infinite loop with a **select** that can break the loop – the `Consumer` function of Figure 1) instead of recursion; we assume that loop indices are not modified in loop bodies and that no **goto**-like constructs are used in a loop.

Since the analysis only takes into account loop parameters, a loop that indefinitely blocks (e.g. due to communication) may be identified as terminating. However, if our analysis identifies the inferred types as live *and* the termination check validates the program, both termination and program liveness are guaranteed.

6 EVALUATION

Table 3 lists several benchmarks of our tool against other static deadlock detection tools for Go (a detailed comparison of these tools is given in § 7). The benchmarks were run with go1.8.3 on an 8-core Intel i7-3770 machine with 16GB RAM on a 64-bit Linux. The model checker we used was mCRL2 v201707.1.

The results for `Godel Checker` are shown in columns 3–11. Column 3 shows the number of states in the input LTS as a measurement of the relative complexity of each program (proportional to the number of concurrency-related operations rather than the number of lines of code). Columns 4–7 shows the core properties of Figure 6 in § 4, i.e. no global deadlock (ψ_g), liveness (ψ_l), channel safety (ψ_s) and eventual reception (ψ_e). Columns 8–10 list the running time of `Godel Checker`, where Column 8 lists the inference time, Columns 9 and 10 are the model checking times for liveness, and both liveness and channel safety, respectively. The total run time can be obtained by adding Column 8 to Column 9 or 10. Unless otherwise stated, all times are in milliseconds. Column 11 (Term) shows the result of the termination check, which proves the termination of loops in the given program, or times out after 15s. A program that times out is conservatively assumed not to terminate.

Columns 12–13 pertain to the `dingo-hunter` tool from [35]. The time includes both communicating finite state machine extraction and their analysis, but does not include building the global graph and only checks for liveness. Columns 14–15 pertain to the `gopherlyzer` tool [39], which only checks for global deadlock-freedom (most programs had to be manually adjusted in order to be accepted by this tool – see § 7 for the severe practical limitations of the tool). Columns 16–18 refer to the `GoInfer/Gong` tool from [30]. The times include both type inference and analysis stages, which only accounts for liveness and channel safety checks. Most programs in Table 3 are taken either from other papers on the static verification of Go programs [30, 35, 39] or from publicly available source code. Programs 7, and 15–22 are introduced by this work. Programs that are unsupported by a tool are marked with *n/a*.

Programs 1–7 are typical concurrent programs from the literature. The `sieve` program is not finite control (it spawns an infinite number of threads), thus it can only be analysed by `GoInfer/Gong`. Program 6 is a (three) dining philosophers program where the first fork can be released, while Program 7 is the traditional deadlocking version (Program 19 is as Program 6 but with 5 philosophers). `dingo-hunter` does not support Programs 6, 7, and 19 due to dynamically spawned goroutines, while `gopherlyzer` does not support them due to a nested select statement. `GoInfer/Gong` analyses them correctly, but is much slower than `Godel Checker`.

Programs 8–12 consist of idiomatic Go patterns which are all handled correctly and quickly by our tool. Program 13 is a publicly available program which is not live. Program 14 is an implementation of the alternating bit protocol. Program 15 is the Producer-Consumer example from § 1, which is not live. All tools were able to verify this simple program. Program 16 demonstrates the mismatch between type and program liveness, where the type is live but due to an erroneous loop the program does not terminate and causes a partial deadlock. The termination check identifies this as possibly non-terminating, while `GoInfer/Gong` incorrectly identifies it as live. Program 17 closes a channel twice which flags a violation of channel safety in `Godel Checker` and `GoInfer/Gong`. Interestingly, `dingo-hunter` detects a deadlock (a false alarm) due to its representation of channel closure as a message exchange, but not due to the double close. `gopherlyzer` also detects a deadlock incorrectly due to the same reason. Program 18 is a program that violates the *eventual reception* property by sending an asynchronous message that is never received – none of the earlier tools can detect this.