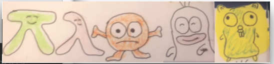
# Behavioural Type-Based Static Verification Framework for Go

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

# Mobility Research Group

π-calculus, Session Types research at Imperial College

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



## NEWS

The paper *Multiparty asynchronous session types* by Kohei Honda, Nobuko Yoshida, and Marco Carbone, published in POPL 2008 has been awarded the ACM SIGPLAN Most Influential POPL Paper Award today at POPL 2018.

» more

10 Jan 2018

Estafet has published a page on their usage of the Scribble language developed in our group with RedHat and other industry partners.

» more

25 Sep 2017

Nick spoke at Golang UK 2017 on applying behavioural types to verify concurrent Go programs.

## SELECTED PUBLICATIONS

2018

Julien Lange , Nicholas Ng , Bernardo Toninho , Nobuko Yoshida : A Static Verification Framework for Message Passing in Go using Behavioural Types. *To appear in* ICSE 2018 .

Bernardo Toninho , Nobuko Yoshida : Depending On Session Typed Process. *To appear in* FoSSaCS 2018 .

Bernardo Toninho , Nobuko Yoshida : On Polymorphic Sessions And Functions: A Talk of Two (Fully Abstract) Encodings. *To appear in* ESOP 2018 .

Rumyana Neykova , Raymond Hu , Nobuko Yoshida , Fahd Abdeljallal : Session Type Providers: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#. *To appear in* CC 2018 .

*Post-docs:*

Simon CASTELLAN

David CASTRO

Francisco FERREIRA

Raymond HU

Rumyana NEYKOVA

Nicholas NG

Alceste SCALAS

*PhD Students:*

Assel ALTAYEVA

Juliana FRANCO

Eva GRAVERSEN

# POPL 2008 MOST INFLUENTIAL PAPER AWARD

# 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/

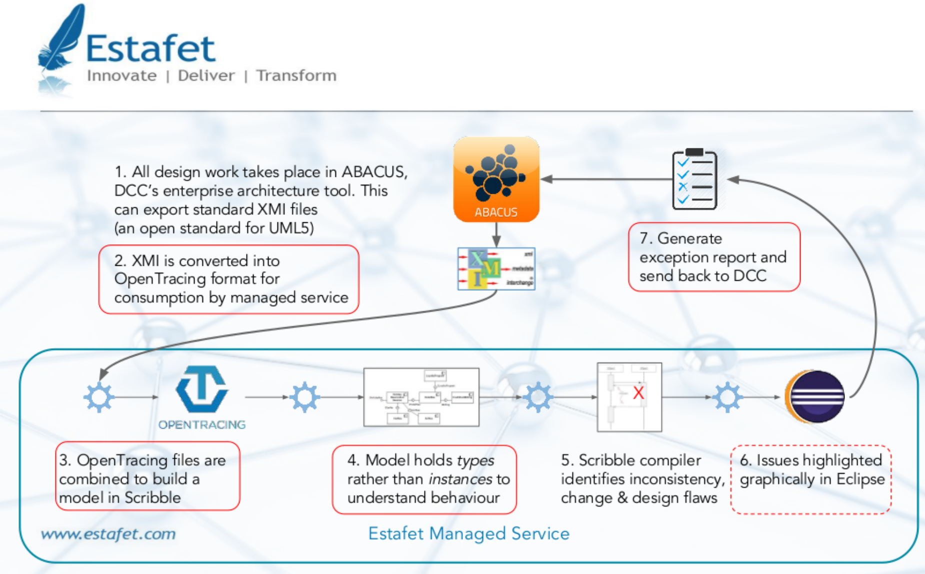
```
module examples;

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

# OOI Collaboration

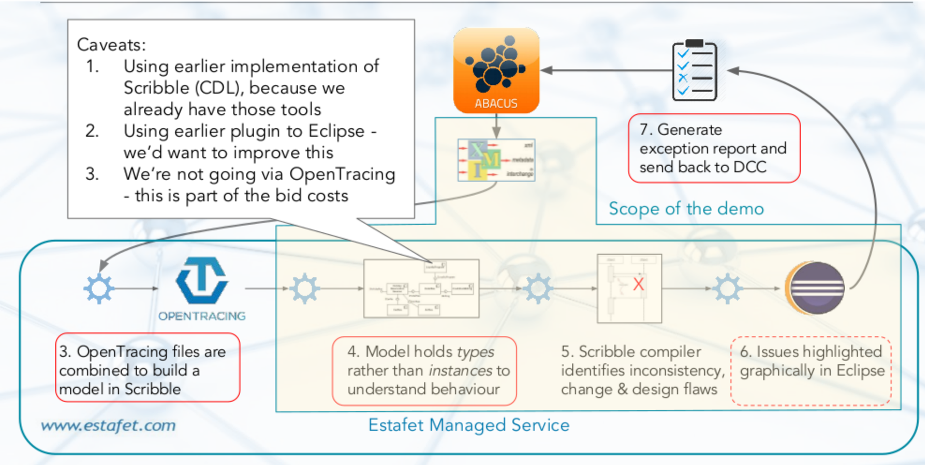

- **TCS'16:** Monitoring Networks through Multiparty Session Types. Laura Bocchi , Tzu-Chun Chen , Romain Demangeon , Kohei Honda , Nobuko Yoshida
- **LMCS'16**: Multiparty Session Actors. Rumyana Neykova, Nobuko Yoshida
- **FMSD'15:** Practical interruptible conversations: Distributed dynamic verification 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

# End-to-End Switching Programme by DCC



# End-to-End Switching Programme by DCC

Estafet
Innovate | Deliver | Transform



# 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

CC'18

**F#unctional Londoners Meetup G**

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

ECOOP'17

**ibuted Systems**
**vs.**
**Compositionality**

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

actyx

ECOOP'16

**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 2017/2018

- **[LICS'18]** Romain Demangeon, NY: Casual Computational Complexity of Distributed Processes.
- **[CC'18]** Rumyana Neykova , Raymond Hu, NY, Fahd Abdeljallal: Session Type Providers: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#.
- **[FoSSaCS'18]** Bernardo Toninho, NY: Depending On Session Typed Process.
- **[ESOP'18]** Bernardo Toninho, NY: On Polymorphic Sessions And Functions: A Talk of Two (Fully Abstract) Encodings.
- **[ESOP'18]** Malte Viering, Tzu-Chun Chen, Patrick Eugster, Raymond Hu , Lukasz Ziarek: A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems.
- **[ICSE'18]** Julien Lange, Nicholas Ng, Bernardo Toninho, NY : A Static Verification Framework for Message Passing in Go using Behavioural Types
- **[ECOOP'17]** Alceste Scala, Raymond Hu, Ornela Darda, NY: A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming..
- **[COORDINATION'17]** Keigo Imai, NY, 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.

# Selected Publications 2017/2018

- **[LICS'18]** Romain Demangeon, NY: Casual Computational Complexity of Distributed Processes.
- **[CC'18]** Rumyana Neykova , Raymond Hu, NY, Fahd Abdeljallal: Session Type Providers: Compile-time API Generation for Distributed Protocols with Interaction Refinements in F#.
- **[FoSSaCS'18]** Bernardo Toninho, NY: Depending On Session Typed Process.
- **[ESOP'18]** Bernardo Toninho, NY: On Polymorphic Sessions And Functions: A Talk of Two (Fully Abstract) Encodings.
- **[ESOP'18]** Malte Viering, Tzu-Chun Chen, Patrick Eugster, Raymond Hu , Lukasz Ziarek: A Typing Discipline for Statically Verified Crash Failure Handling in Distributed Systems.
- **[ICSE'18]** Julien Lange, Nicholas Ng, Bernardo Toninho, NY : A Static Verification Framework for Message Passing in Go using Behavioural Types.
- **[ECOOP'17]** Alceste Scala, Raymond Hu, Ornela Darda, NY: A Linear Decomposition of Multiparty Sessions for Safe Distributed Programming.
- **[COORDINATION'17]** Keigo Imai, NY, 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.

Imperial College London

Home | College and Campus | Science | Engineering | Health | Business | Search here... Go ▸

News

# Go concurrency verification research at DoC grabs headline

POPL'17

**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)".

# the morning paper

ICSE'18

an interesting/influential/important paper from the world of CS every weekday morning, as selected by Adrian Colyer

Home About InfoQ QR Editions Subscribe

## 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!

SUBSCRIBE

never miss an issue! The Morning Paper delivered straight to your inbox.

SEARCH

type and press enter

ARCHIVES

Select Month

MOST READ IN THE LAST FEW DAYS

# 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'

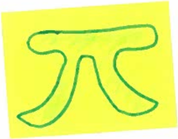
# FUN

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

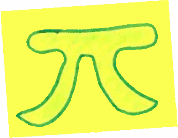
- Dynamic channel creations, unbounded thread creations, recursions, ..
- Scalable (synchronous/asynchronous) Modular, Refinable

- 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?

π

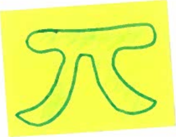
- Use behavioural types in process calculi

  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

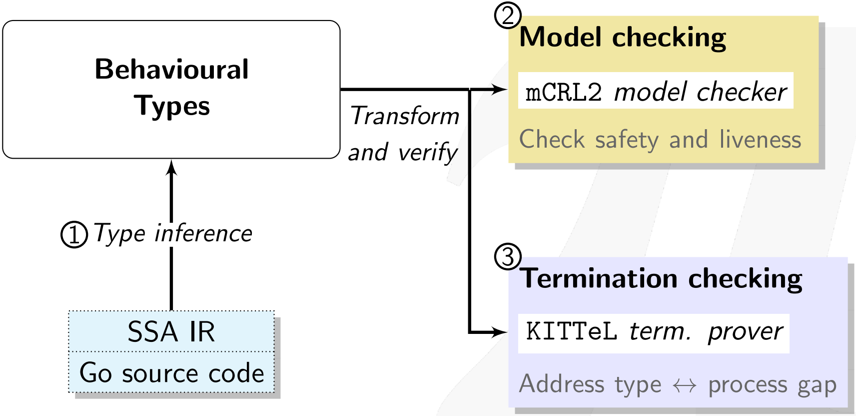
- Model - Checking ( Finite Control )


**STEP 3**

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

# Static verification framework for Go

## Overview



# Concurrency in Go

## Concurrency primitives

```go
func main() {
        ch := make(chan int) // Create channel.
        go send(ch)          // Spawn as goroutine.
        print(<-ch)          // Recv from channel.
}

func send(ch chan int) { // Channel as parameter.
        ch <- 1 // Send to channel.
}
```

- Send/receive blocks goroutines if channel full/empty resp.
- Channel buffer size specified at creation: `make(chan int, 1)`
- Other primitives:
  - Close a channel `close(ch)`
  - Guarded choice `select { case <-ch:; case <-ch2: }`

# Concurrency in Go

## Deadlock detection

```
func main() {
        ch := make(chan int) // Create channel.
        send(ch)             // Spawn as goroutine.
        print(<-ch)          // Recv from channel.
}

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

Missing 'go' keyword

# Concurrency in Go

## Deadlock detection

```
func main() {
        ch := make(chan int) // Create channel.
        send(ch)             // Spawn as goroutine.
        print(<-ch)          // Recv from channel.
}

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

Run program:

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

# Concurrency in Go

## Deadlock detection

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

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

# Concurrency in Go

## Deadlock detection

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

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

Add benign import

Deadlock **NOT** detected

# Abstracting Go with Behavioural Types

## Type syntax

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

- 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)

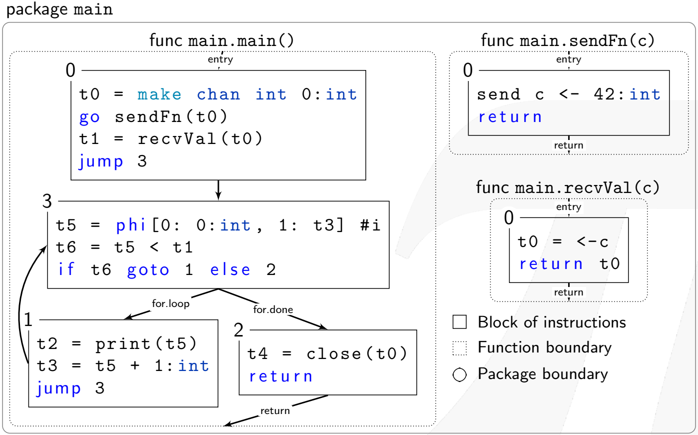
# Infer Behavioural Types from Go program

## Input Go source code

```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

## Program in Static Single Assignment (SSA) form




- Context-sensitive analysis to distinguish channel variables
- Skip over non-communication code

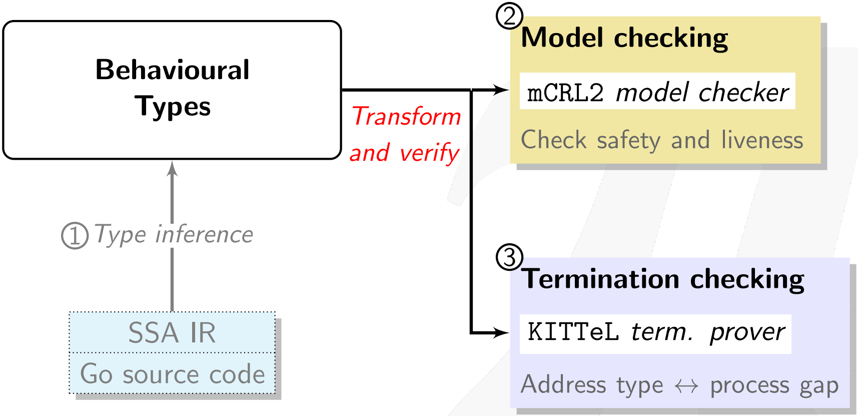
# Infer Behavioural Types from Go program

## Types inferred from program

```
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
```

$$\begin{aligned}
\textbf{main}() &= (\mathtt{new}\ t0)(\textbf{sendFn}\langle t0\rangle \mid \textbf{recvVal}\langle t0\rangle; \textbf{main_3}\langle t0\rangle) \\
\textbf{main_1}(t0) &= \textbf{main_3}\langle t0\rangle \\
\textbf{main_2}(t0) &= \mathtt{close}\ t0; \mathbf{0} \\
\textbf{main_3}(t0) &= \textbf{main_1}\langle t0\rangle \oplus \textbf{main_2}\langle t0\rangle \\
\textbf{sendFn}(c) &= \overline{c}; \mathbf{0} \\
\textbf{recvVal}(c) &= c; \mathbf{0}
\end{aligned}$$

# Model checking behavioural types



# Model checking behavioural types

## Behavioural types as LTS model

1. Generate LTS **model** from type semantics
2. Generate $\mu$-calculus formulae for LTS describing **properties**
3. Check LTS $\models$ formulae with model checker (e.g. mCRL2)

---

Properties of interest:

- Global deadlock freedom
- Channel safety (no send/`close` on closed channel)
- Liveness (partial deadlock freedom)
- Eventual reception

Constraints (on mCRL2 model checker):

- Finite control (no parallel composition in recursion)

Mi Go Liveness / Safety

P ⇓ 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 $P \xrightarrow{*} (\nu \tilde{c})Q$ 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 ($\tau$au) at $a$

# Select

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

Time Out

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

$R_1 = a?$

# Select

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

Time Out

if P is live

$P_1$ is live

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

$R_1$ = 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

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

Time Out

if P is live

$P_1$ is live

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

$R_1 = a?$

$P_2$ is not live

$P_2 | R_2$ is

Barb $\downarrow \tilde{a}$

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

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

# Model checking behavioural types

## Generating $\mu$-calculus formulae (channel safety)

- Given an LTS model, generate formulae for safety properties
- Note: formulae are *model-specific*

### Property: Channel safety

$$\psi_s \stackrel{\text{def}}{=} \left( \bigwedge_{a \in \mathcal{A}} \downarrow_{a^\star} \right) \implies \neg(\downarrow_{\overline{a}} \vee \downarrow_{\texttt{clo}\, a})$$

$\langle\alpha\rangle\phi$ is a modal operator, satisfied if:
There is a T' where T $\xrightarrow{\alpha}$ T' such that formula $\phi$ holds

# Model checking behavioural types

## Generating $\mu$-calculus formulae (channel safety)

### Property: Channel safety

$$\psi_s \stackrel{\text{def}}{=} \Big(\bigwedge_{a \in \mathcal{A}} \downarrow_{a^\star}\Big) \implies \neg(\downarrow_{\overline{a}} \vee \downarrow_{\texttt{clo}\ a})$$

$\langle\alpha\rangle\phi$ is a modal operator, satisfied if:
There is a T' where T $\xrightarrow{\alpha}$ T' such that formula $\phi$ holds

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

# Model checking behavioural types

## Generating $\mu$-calculus formulae (liveness)

### Property: Liveness

$$\psi_{l_a} \stackrel{\text{def}}{=} \Big(\bigwedge_{a \in \mathcal{A}} \downarrow_{a} \vee \downarrow_{\bar{a}}\Big) \implies \text{eventually}\,(\langle \tau_a \rangle \mathtt{true})$$

### Property: Liveness (select)

$$\psi_{l_b} \stackrel{\text{def}}{=} \Big(\bigwedge_{\tilde{a} \in \mathcal{P}(\mathcal{A})} \downarrow_{\tilde{a}}\Big) \implies \text{eventually}\,(\langle \{\tau_a \mid a \in \tilde{a}\} \rangle \mathtt{true})$$

- Liveness: sometimes known as *partial deadlock freedom*
- Program is live if $(\psi_{l_a} \wedge \psi_{l_b})$ holds

# Model checking behavioural types

## Summary

1. Generate LTS **model** from type semantics
2. Generate $\mu$-calculus formulae for LTS describing **properties**
3. Check LTS $\models$ formulae with model checker (e.g. mCRL2)

Properties:

- ✓ Global deadlock freedom
- ✓ Channel safety (no send/`close` on closed channel)
- ✗ Liveness (partial deadlock freedom)
- ✗ Eventual reception
  - Require additional guarantees

# Model checking behavioural types

## Termination checking with KITTeL

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

```go

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

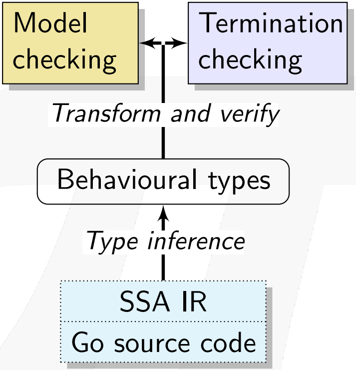
- Type: Live
- Program: NOT live

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
        - Godel-Checker model checking [ICSE'18] (this talk)
        - Gong type verifier [POPL'17]
        - Choreography synthesis [CC'15]
    - Different concurrency issues
        - Other synchronisation mechanisms
        - Race conditions

# Semantics of MiGo types

$$\text{SND}\quad \overline{a}; T \xrightarrow{\overline{a}} T \qquad \text{RCV}\quad a; T \xrightarrow{a} T \qquad \text{TAU}\quad \tau; T \xrightarrow{\tau} T$$

$$\text{END}\quad \texttt{close}\ a; T \xrightarrow{\texttt{clo}\ a} T \qquad \text{BUF}\quad \lfloor a \rfloor^n_k \xrightarrow{\overline{\texttt{clo}}\ a} a^\star \qquad \text{CLD}\quad a^\star \xrightarrow{a^\star} a^\star$$

$$\text{SEL}\quad \frac{i \in \{1,2\}}{T_1 \oplus T_2 \xrightarrow{\tau} T_i} \qquad \text{BRA}\quad \frac{\alpha_j; T_j \xrightarrow{\alpha_j} T_j \qquad j \in I}{\&\{\alpha_i; T_i\}_{i\in I} \xrightarrow{\alpha_j} T_j}$$

$$\text{PAR}\quad \frac{T \xrightarrow{\alpha} T'}{T \mid S \xrightarrow{\alpha} T' \mid S} \qquad \text{SEQ}\quad \frac{T \xrightarrow{\alpha} T'}{T; S \xrightarrow{\alpha} T'; S} \qquad \text{TERM}\quad \mathbf{0}; S \xrightarrow{\tau} S$$

$$\text{COM}\quad \frac{\alpha \in \{\overline{a}, a^\star, a^\bullet\} \qquad T \xrightarrow{\alpha} T' \qquad S \xrightarrow{\beta} S' \qquad \beta \in \{{}^\bullet a, a\}}{T \mid S \xrightarrow{\tau_a} T' \mid S'}$$

$$\text{EQ}\quad \frac{T \equiv_\alpha T' \qquad T \xrightarrow{\alpha} T''}{T' \xrightarrow{\alpha} T''} \qquad \text{DEF}\quad \frac{T\,\{\tilde{a}/\tilde{x}\} \xrightarrow{\alpha} T' \qquad \mathbf{t}(\tilde{x}) = T}{\mathbf{t}\langle \tilde{a} \rangle \xrightarrow{\alpha} T'}$$

$$\text{CLOSE}\quad \frac{T \xrightarrow{\texttt{clo}\ a} T' \qquad S \xrightarrow{\overline{\texttt{clo}}\ a} S'}{T \mid S \xrightarrow{\tau} T' \mid S'} \qquad \text{IN}\quad \frac{k < n}{\lfloor a \rfloor^n_k \xrightarrow{{}^\bullet a} \lfloor a \rfloor^n_{k+1}} \qquad \text{OUT}\quad \frac{k \geq 1}{\lfloor a \rfloor^n_k \xrightarrow{a^\bullet} \lfloor a \rfloor^n_{k-1}}$$

Figure: Semantics of types.

# Barb predicates for MiGo types

$$a; T \downarrow_a \qquad \texttt{close}\ a; T \downarrow_{\texttt{clo}\ a} \qquad \frac{\forall i \in \{1,\ldots,n\} : \alpha_i \downarrow_{o_i}}{\&\{\alpha_i; T\}_{i\in\{1,\ldots,n\}} \downarrow_{\{o_1\ldots o_n\}}}$$

$$\overline{a}; T \downarrow_{\overline{a}} \qquad a^\star \downarrow_{a^\star}$$

$$\frac{T \downarrow_o}{T; T' \downarrow_o} \qquad \frac{T \downarrow_a \quad T' \downarrow_{\overline{a}} \text{ or } T' \downarrow_{a^\star}}{T \mid T' \downarrow_{\tau_a}} \qquad \frac{T\{\tilde{a}/\tilde{x}\} \downarrow_o \quad \mathbf{t}(\tilde{x}) = T}{\mathbf{t}\langle\tilde{a}\rangle \downarrow_o}$$

$$\frac{T \downarrow_a \quad \alpha_i \downarrow_{\overline{a}}}{T \mid \&\{\alpha_i; S_i\}_{i\in I} \downarrow_{\tau_a}} \qquad \frac{T \downarrow_{\overline{a}} \text{ or } T \downarrow_{a^\star} \quad \alpha_i \downarrow_a}{T \mid \&\{\alpha_i; S_i\}_{i\in I} \downarrow_{\tau_a}}$$

$$\frac{k < n}{\lfloor a \rceil^n_k \downarrow_{\bullet a}} \qquad \frac{k \geq 1}{\lfloor a \rceil^n_k \downarrow_{a\bullet}} \qquad \frac{T \downarrow_{\overline{a}} \quad T' \downarrow_{\bullet a}}{T \mid T' \downarrow_{\tau_a}} \qquad \frac{T \downarrow_{a\bullet} \quad \alpha_i \downarrow_a}{T \mid \&\{\alpha_i; S_i\}_{i\in I} \downarrow_{\tau_a}}$$

$$\frac{T \downarrow_o}{T \mid T' \downarrow_o} \qquad \frac{T \downarrow_o \quad a \notin \mathrm{fn}(o)}{(\mathrm{new}^n\ a); T \downarrow_o} \qquad \frac{T \downarrow_o \quad T \equiv T'}{T \downarrow_o}$$

Figure: Barb predicates for types.

# Model checking behavioural types

## Generating $\mu$-calculus formulae (global deadlock freedom)

- Given an LTS model, generate formulae for safety properties
- Note: formulae are *model-specific*

**Property: Global deadlock freedom**

$$\psi_g \stackrel{\text{def}}{=} \Big(\bigwedge_{a \in \mathcal{A}} \downarrow_a \vee \downarrow_{\overline{a}}\Big) \implies \langle \mathbb{A} \rangle \mathtt{true}$$

- $\langle \alpha \rangle \phi$ is a modal operator, satisfied if:
  There is a T' where T $\xrightarrow{\alpha}$ T' such that formula $\phi$ holds

# Model checking behavioural types

## Generating $\mu$-calculus formulae (eventual reception)

### Property: Eventual reception

$$\psi_e \stackrel{\text{def}}{=} \left(\bigwedge_{a \in \mathcal{A}} \downarrow_{a\bullet}\right) \implies \text{eventually}\left(\langle \tau_a \rangle \texttt{true}\right)$$

- Applies only to buffered channels

# Eventually

**Eventually**

$$\text{eventually}(\phi) \stackrel{\text{def}}{=} \mu \mathbf{y}.\, (\phi \vee \langle \mathbb{A} \rangle \mathbf{y})$$

- i.e. $\phi$ holds in some reachable state