OPEN PROBLEMS OF SESSION TYPES
Us ∈ Mobility Research Group

http://mrg.doc.ic.ac.uk/
• **TCS’16**: Monitoring Networks through Multiparty Session Types. Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, Nobuko Yoshida
• **LMCS’16**: Multiparty Session Actors. Rummyana Neykova, Nobuko Yoshida
• **FMSD’15**: Practical interruptible conversations: Distributed dynamic verification with multiparty session types and Python. Romain Demangeon, Kohei Honda, Raymond Hu, Rummyana Neykova, Nobuko Yoshida
• **TGC’13**: The Scribble Protocol Language. Nobuko Yoshida, Raymond Hu, Rummyana Neykova, Nicholas Ng
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.
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;
    }
}
End-to-End Switching Programme by DCC

1. All design work takes place in ABACUS, DCC’s enterprise architecture tool. This can export standard XMI files (an open standard for UML5)

2. XMI is converted into OpenTracing format for consumption by managed service

3. OpenTracing files are combined to build a model in Scribble

4. Model holds types rather than instances to understand behaviour

5. Scribble compiler identifies inconsistency, change & design flaws

6. Issues highlighted graphically in Eclipse

7. Generate exception report and send back to DCC

www.estafet.com
End-to-End Switching Programme by DCC

Caveats:
1. Using earlier implementation of Scribble (CDL), because we already have those tools
2. Using earlier plugin to Eclipse - we'd want to improve this
3. We're not going via OpenTracing - this is part of the bid costs

Scope of the demo:
7. Generate exception report and send back to DCC

3. OpenTracing files are combined to build a model in Scribble
4. Model holds types rather than instances to understand behaviour
5. Scribble compiler identifies inconsistency, change & design flaws
6. Issues highlighted graphically in Eclipse

www.estafet.com

Estafet Managed Service
Interactions with Industries

Strange Loop
SEPTEMBER 15-17 2016 / PEABODY OPERA HOUSE / ST. LOUIS, MO

Adam Bowen @adamm Bowen · Sep 15
I didn’t even know that session types existed an hour ago, but thanks to Nobuko Yoshida’s great talk at #pwiconf, 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

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

The Golang UK Conference
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

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)
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)".
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.
- **[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.
- **[POPL’16]** Dominic Orchard, NY: Effects as sessions, sessions as effects.
Selected Publications 2016/2017

• [CC’16] Nicholas Ng, NY: Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis.
HOW to

- deriv theories to practices
- make theories understandable
- meet theoretical challenges (concurrency, distributions)
- communicate people
**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 behavioral 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, 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, reifiable
  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

Transform and verify

(1) Type inference

SSA IR

Go source code

(2) Model checking

(3) Termination checking

Address type and process gap

Pass to termination prover

Behavioural types
Concurrency in Go

```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 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

```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.

Run program:

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

Nobuko Yoshida
Open Problems of Session Types
Concurrent in Go
Deadlock detection

```go
func main() {
    ch, done := make(chan int), make(chan int)
    go send(ch) // Spawn as goroutine.
    go func() {
        i := 0
        for i++ { // infinite loop
            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 loop
Concernty in Go

Deadlock detection

```go
func main() {
    ch, done := make(chan int), make(chan int)
    go send(ch) // Spawn as goroutine.
    go func() {
        for i := 0; ; i++ { // infinite loop
            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.

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. \texttt{net} for networking) \textbf{disables} it

```go
import _ "net" // Load "net" package
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

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

Deadlock **NOT** detected
Go Programs as Processes

<table>
<thead>
<tr>
<th>Go Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P, Q ) (:=\ \pi; P )</td>
</tr>
<tr>
<td>( \pi ) (:=\ u!\langle e\rangle \mid u?(y) \mid \tau )</td>
</tr>
</tbody>
</table>
## Go Programs as Processes

### Go Program

| $P, Q$ | $\pi; P$ | $\pi$ := $u!(e) | u?(y) | \tau$ |
|-------|---------|-------------------|
|       | $u; P$  |                   |
Go Program

\[
P, Q \ := \ \pi; P \\
\ | \\
\ | \ u; P \\
\ | \ \{\pi_i; P_i\}_{i \in I} \\
\pi \ := \ u!\langle e \rangle \ | \ u?(y) \ | \ \tau
\]
Go Programs as Processes

<table>
<thead>
<tr>
<th>Go Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P, Q ;:=; \pi; P$</td>
</tr>
<tr>
<td>$\quad</td>
</tr>
<tr>
<td>$\quad</td>
</tr>
<tr>
<td>$\quad</td>
</tr>
<tr>
<td>$\pi ;:=; u!\langle e \rangle \mid u? (y) \mid \tau$</td>
</tr>
</tbody>
</table>
Go Programs as Processes

Go Program

\[ P, Q ::= \pi; P \]
\[ u; P \]
\[ \{ \pi_i; P_i \}_{i \in I} \]
\[ e P Q \]
\[ (y: \sigma); P \]

\[ \pi ::= u!\langle e \rangle \mid u?(y) \mid \tau \]
Go Programs as Processes

Go Program

\[
P, Q \quad := \quad \pi; P \\
\quad \quad u; P \\
\quad \quad \{\pi_i; P_i\}_{i \in I} \\
\quad \quad e \ P \ Q \\
\quad \quad (y:\sigma); P \\
\quad \quad P \mid Q \mid 0 \mid (\nu c)P
\]

\[
\pi \quad := \quad u!\langle e \rangle \mid u?\langle y \rangle \mid \tau
\]
Go Program

Go Programs as Processes

<table>
<thead>
<tr>
<th>P, Q :=</th>
<th>π; P</th>
<th>u; P</th>
<th>{π_i; P_i}_{i \in I}</th>
<th>e P Q</th>
<th>(y:σ); P</th>
<th>P</th>
<th>Q</th>
<th>0</th>
<th>(νc)P</th>
<th>X⟨\tilde{e}, \tilde{u}⟩</th>
</tr>
</thead>
<tbody>
<tr>
<td>P, Q :=</td>
<td>\pi := u!(e)</td>
<td>u?(y)</td>
<td>τ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D :=</td>
<td>X(\tilde{x}) = P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P :=</td>
<td>{D_i}_{i \in I} P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Go Programs as Processes

Go Program

\[
P, Q \quad := \quad \pi; P \\
\quad \mid u; P \\
\quad \mid \{\pi_i; P_i\}_{i \in I} \\
\quad \mid e \ P \ Q \\
\quad \mid (y:\sigma); P \\
\quad \mid P \mid Q \mid \mathbf{0} \mid (\nu c)P \\
\quad \mid X(\tilde{e}, \tilde{u})
\]

\[
D \quad := \quad X(\tilde{\chi}) = P
\]

\[
P \quad := \quad \{D_i\}_{i \in I} \ P
\]

\[
\pi \quad := \quad u!\langle e\rangle \mid u?(y) \mid \tau
\]
Abstracting Go with Behavioural Types

Types

\[ \alpha := \overline{u} | u | \tau \]
\[ T, S := \alpha; T | T \oplus S | \&\{\alpha_i; T_i\}_{i \in I} | (T | S) | 0 \]
\[ \quad | (\text{new } a)T | \text{close } u; T | t(\bar{u}) \]
\[ T := \{t(\bar{y}_i) = T_i\}_{i \in I} \text{ in } S \]

- 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

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

Channel Safety

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

\[ P \text{ is channel safe if } P \xrightarrow{*} (\forall \omega) Q \text{ and } Q \downarrow \text{close}(a) \]

\[ \neg (Q \downarrow \text{end}(a)) \land \neg (Q \downarrow \overline{a}) \]

never closing  never send
Migo: Liveness / Safety

**Liveness**

All reachable actions are eventually performed.

\[
P \text{ is live if } P \xrightarrow{\ast} (\forall \bar{Q}) Q
\]

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

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

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

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

\[ R_1 = a? \]

Time Out

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

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

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

Barb \downarrow \bar{a}

\[ \pi_i \downarrow \bar{a}_i \]

Select \{ \pi_i \cdot P_i \} \downarrow \bar{a}

\[ P \downarrow \bar{a} \quad Q \downarrow \bar{a}_i \]

\[ P \uparrow Q \downarrow [\bar{a}_i] \]

Liveness \[ Q \downarrow \bar{a} \Rightarrow Q \downarrow z \text{ at } ai \]
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/\texttt{close} on closed channel)
  - Liveness (partial deadlock)
  - Eventual reception
  - Require additional guarantees
the \( \mu \)-calculus encoding properties with barbs

\( \forall a \in C \ (\not\downarrow a \lor \not\downarrow \bar{a}) \Rightarrow \langle \alpha \rangle T \)

\( \forall a \in C \ \downarrow \text{close} a \Rightarrow \neg (\not\downarrow \bar{a} \lor \not\downarrow \text{close} a) \)

\( \forall a \in C \ (\not\downarrow a \lor \not\downarrow \bar{a}) \Rightarrow \Phi (\langle [a] \rangle T) \land \)

\( \forall \bar{a} \in C^n \ \downarrow \bar{a} \Rightarrow \Phi (\forall a \in \bar{a} \langle [a] \rangle T) \)

[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/closed on closed channel)
  - ✓ Liveness (partial deadlock)
  - ✓ Eventual reception

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

- 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