An empirical study of messaging passing concurrency in Go projects

Nicolas Dilley  Julien Lange

University of Kent

ABCD Meeting — December 2018
Introduction

Go: an open source programming language that makes it easy to build simple, reliable, and efficient software [golang.org].

- Go has become a **key ingredient** of many modern software, e.g., main language of Docker and Kubernetes.
- Go offers **lightweight threads** and **channel-based communication**.
- These communication primitives are similar to synchronisation mechanisms in **process calculi**, e.g., CSP, CCS, and $\pi$-calculus.
func worker(j int, x chan<- int, y <-chan int) {
    for {
        select {
            case x <- j: // send
            case <- y: return // receive
        }
    }
}

func main() {
    a := make(chan int, 5)
    b := make(chan int)
    for i := 0; i < 30; i++ {
        go worker(i, a, b)
    }
    for i := 0; i < 10; i++ {
        k := <-a // receive
        fmt.Println(k)
    }
    close(b)
}
func worker(j int, x chan<- int, y <-chan int) {
    for {
        select {
            case x <-j: // send
            case <-y: return // receive
        }
    }
}

func main() {
    a := make(chan int, 5)
    b := make(chan int)
    
    for i := 0; i < 30; i++ {
        go worker(i, a, b)
    }
    for i := 0; i < 10; i++ {
        k := <-a // receive
        fmt.Println(k)
    }
    close(b)
}
Complex concurrency patterns: concurrent prime sieve

```go
func worker(j int, x chan<- int, y <-chan int) {
    for {
        select {
            case x <- j: // send
            case <- y: return // receive
        }
    }
}

func main() {
    a := make(chan int, 5)
    b := make(chan int)

    for i := 0; i < 30; i++ {
        go worker(i, a, b)
    }
    close(b)
}
```
Complex concurrency patterns: concurrent prime sieve

```go
func worker(j int, x chan<- int, y <-chan int) {
    for {
        select {
            case x <- j:     // send
            case <-y: return // receive
        }
    }
}

func main() {
    a := make(chan int, 5)
    b := make(chan int)

    for i := 0; i < 30; i++ {
        go worker(i, a, b)
    }

    for i := 0; i < 10; i++ {
        k := <-a      // receive
        fmt.Println(k)
    }
}
```
func worker(j int, x chan<- int, y <-chan int) {
    for {
        select {
            case x <-j: // send
            case <-y: return // receive
        }
    }
}

func main() {
    a := make(chan int, 5)
    b := make(chan int)

    for i := 0; i < 30; i++ {
        go worker(i, a, b)
    }
    for i := 0; i < 10; i++ {
        k := <-a // receive
        fmt.Println(k)
    }
    close(b)
}
Growing support for verification of Go programs.

Static verification:

- **Dingo-hunter**: multiparty compatibility [Ng & Yoshida; CC’16]
- **Gong**: (bounded) model checking [L, Ng, Toninho, Yoshida; POPL’17]
- **Godel**: mCRL2 model checker [L, Ng, Toninho, Yoshida; ICSE’18]
Context: verification of Go programs

Growing support for verification of Go programs.

Static verification:

- **Dingo-hunter**: multiparty compatibility [Ng & Yoshida; CC’16]
- **Gong**: (bounded) model checking [L, Ng, Toninho, Yoshida; POPL’17]
- **Godel**: mCRL2 model checker [L, Ng, Toninho, Yoshida; ICSE’18]
- **Gopherlyzer**: forkable regular expression [Stadtmüller, Sulzmann, Thieman; APLAS’16]
- **Nano-Go**: abstract interpretation [Midtgaard, Nielson, Nielson; SAS’18]
Growing support for verification of Go programs.

Static verification:

- **Dingo-hunter**: multiparty compatibility [Ng & Yoshida; CC’16]
- **Gong**: (bounded) model checking [L, Ng, Toninho, Yoshida; POPL’17]
- **Godel**: mCRL2 model checker [L, Ng, Toninho, Yoshida; ICSE’18]
- **Gopherlyzer**: forkable regular expression [Stadtmüller, Sulzmann, Thieman; APLAS’16]
- **Nano-Go**: abstract interpretation [Midtgaard, Nielson, Nielson; SAS’18]

Runtime verification:

- **Gopherlyzer-GoScout**: [Sulzmann & Stadtmüller; PPDP’17] and [Sulzmann & Stadtmüller; HVC’17]
Challenges for the verification of message passing programs

**Scalability** (wrt. program size)

- Number of message passing primitives (send, receive, etc)
- Number of threads
- Size of channel bounds
Challenges for the verification of message passing programs

**Scalability** (wrt. program size)
- Number of message passing primitives (send, receive, etc)
- Number of threads
- Size of channel bounds

**Expressivity** (of the communication/synchronisation patterns)
- Spawning new threads within loops
- Creating new channels within loops
- Channel passing
Research questions

- **RQ1:** *How often are messaging passing operations used in Go projects?*
Research questions

- **RQ1:** How often are messaging passing operations used in Go projects?
  - How many projects use message passing?
Research questions

- **RQ1**: *How often are messaging passing operations used in Go projects?*
  - How many projects use message passing?
  - How intensively do they use message passing?
Research questions

- **RQ1**: How often are messaging passing operations used in Go projects?
  - How many projects use message passing?
  - How intensively do they use message passing?
- **RQ2**: How is concurrency spread across Go projects?
Research questions

- **RQ1:** How often are messaging passing operations used in Go projects?
  - How many projects use message passing?
  - How intensively do they use message passing?
- **RQ2:** How is concurrency spread across Go projects?
  - Can a static analysis focus on specific parts of a codebase?
Research questions

- **RQ1:** How often are messaging passing operations used in Go projects?
  - How many projects use message passing?
  - How intensively do they use message passing?
- **RQ2:** How is concurrency spread across Go projects?
  - Can a static analysis focus on specific parts of a codebase?
- **RQ3:** How common is the usage of asynchronous message passing in Go projects?
Research questions

▶ **RQ1:** *How often are messaging passing operations used in Go projects?*
  ▶ How many projects use message passing?
  ▶ How intensively do they use message passing?

▶ **RQ2:** *How is concurrency spread across Go projects?*
  ▶ Can a static analysis focus on specific parts of a codebase?

▶ **RQ3:** *How common is the usage of asynchronous message passing in Go projects?*
  ▶ Is asynchrony a problem wrt. scalability?
Research questions

▶ **RQ1:** *How often are messaging passing operations used in Go projects?*
  ▶ How many projects use message passing?
  ▶ How intensively do they use message passing?

▶ **RQ2:** *How is concurrency spread across Go projects?*
  ▶ Can a static analysis focus on specific parts of a codebase?

▶ **RQ3:** *How common is the usage of asynchronous message passing in Go projects?*
  ▶ Is asynchrony a problem wrt. scalability?

▶ **RQ4:** *What concurrent topologies are used in Go projects?*
Research questions

- **RQ1:** *How often are messaging passing operations used in Go projects?*
  - How many projects use message passing?
  - How intensively do they use message passing?

- **RQ2:** *How is concurrency spread across Go projects?*
  - Can a static analysis focus on specific parts of a codebase?

- **RQ3:** *How common is the usage of asynchronous message passing in Go projects?*
  - Is asynchrony a problem wrt. scalability?

- **RQ4:** *What concurrent topologies are used in Go projects?*
  - What sort of constructs should we focus on next?
Selected the **top 900 Go projects** (wrt. number of stars)
Manually selected 865 projects (35 million PLOC).
Automatically analysed the AST of each `.go` in each project.
Telemetry stored in **machine readable** CSV files and **human browsable** HTML files.
RQ1: How often are messaging passing operations used in Go projects?
How common is message passing in 865 projects?

<table>
<thead>
<tr>
<th>Feature</th>
<th>Projects</th>
<th>Proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>chan</td>
<td>661</td>
<td>76%</td>
</tr>
<tr>
<td>send</td>
<td>617</td>
<td>71%</td>
</tr>
<tr>
<td>receive</td>
<td>674</td>
<td>78%</td>
</tr>
<tr>
<td>select</td>
<td>576</td>
<td>66%</td>
</tr>
<tr>
<td>close</td>
<td>402</td>
<td>46%</td>
</tr>
<tr>
<td>range</td>
<td>228</td>
<td>26%</td>
</tr>
</tbody>
</table>

- 204 projects out of 865 (∼ 24%) do not create any communication channels.
- the receive primitive is the most frequently used message passing operation.
How common is message passing in 865 projects?

<table>
<thead>
<tr>
<th>Feature</th>
<th>Projects</th>
<th>Proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>chan</td>
<td>661</td>
<td>76%</td>
</tr>
<tr>
<td>send</td>
<td>617</td>
<td>71%</td>
</tr>
<tr>
<td>receive</td>
<td>674</td>
<td>78%</td>
</tr>
<tr>
<td>select</td>
<td>576</td>
<td>66%</td>
</tr>
<tr>
<td>close</td>
<td>402</td>
<td>46%</td>
</tr>
<tr>
<td>range</td>
<td>228</td>
<td>26%</td>
</tr>
</tbody>
</table>

- 204 projects out of 865 (~ 24%) do not create any communication channels.
- the receive primitive is the most frequently used message passing operation.

NB: receive is also used for delay and timeouts.
Intensity of message passing: absolute measurements

Occurrences in 661 projects

Occurrences in 32 projects

The 32 projects are those whose size falls within 10% of the median size (between 1.7 and 2.1 kPLOC).
Intensity of message passing: relative measurements

▶ 6.34 channels for every 1 kPLOC (median of 4.69) in concurrency-related files.
▶ Some clear outliers, e.g., anaconda with one channel creation every 18 PLOC.
▶ On average: **1.26 sends and 2.08 receives per channel.**
RQ2: *How is concurrency spread across Go projects?*
Concurrency spread

Concurrency spread in 661 projects  Concurrency spread in 32 projects

- **Size:** gives the ratio of concurrent size to the total number of physical lines of code.

- **Package:** ratio of number of packages featuring concurrency to the total number of packages.

- **File:** gives the ratio of number of files containing some concurrency features to the total number of files.
RQ3: How common is the usage of asynchronous message passing in Go projects?
Communication channels in 661 projects

<table>
<thead>
<tr>
<th>Type</th>
<th>occurrences</th>
<th>proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>All channels</td>
<td>22226</td>
<td>100%</td>
</tr>
</tbody>
</table>
Communication channels in 661 projects

<table>
<thead>
<tr>
<th>Type</th>
<th>occurrences</th>
<th>proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>All channels</td>
<td>22226</td>
<td>100%</td>
</tr>
<tr>
<td>Channels with known bounds</td>
<td>20868</td>
<td>94%</td>
</tr>
<tr>
<td>Synchronous channels</td>
<td>13639</td>
<td>61%</td>
</tr>
<tr>
<td>Asynchronous channels (known)</td>
<td>7229</td>
<td>33%</td>
</tr>
<tr>
<td>Channels with unknown bounds</td>
<td>1358</td>
<td>6%</td>
</tr>
</tbody>
</table>

- Asynchrony is much less common than synchrony (default).
- 3237/7229 (45%) asynchronous channels with statically known bounds were in test files.
Known sizes of asynchronous channels

<table>
<thead>
<tr>
<th>size</th>
<th>mean</th>
<th>std</th>
<th>min</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1193.62</td>
<td>29838.20</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>1,000,000</td>
</tr>
</tbody>
</table>

- Channel bounds are $\leq 5$ in 75% of the cases.
- **Large bounds** tend to be used to simulate unbounded asynchrony.
RQ4: What concurrent topologies are used in Go projects?
Complex concurrency patterns: concurrent prime sieve

```go
func generate(ch chan<- int) {
    for i := 2; ; i++ {ch <-i}
}
```
Complex concurrency patterns: concurrent prime sieve

```go
func generate(ch chan<- int) {
    for i := 2; ; i++ {ch <-i}
}

func filter(in chan int, out chan int, p int) {
    for {i := <-in
        if i%p != 0 {out <-i}
    }
}

func main() {
    ch := make(chan int)
go generate(ch)
    bound := readFromUser()
    for i := 0; i < bound; i++ {
        prime := <-ch
        fmt.Println(prime)
        ch1 := make(chan int)
go filter(ch, ch1, prime)
    ch = ch1
}
```
Complex concurrency patterns: concurrent prime sieve

```go
func generate(ch chan<- int) {
    for i := 2; ; i++ { ch <- i }
}

func filter(in chan int, out chan int, p int) {
    for { i := <-in
        if i%p != 0 { out <- i }
    }
}

func main() {
    ch := make(chan int)
    go generate(ch)
    bound := readFromUser()
```
func generate(ch chan<- int) {
    for i := 2; ; i++ {ch <-i}
}

func filter(in chan int, out chan int, p int) {
    for {i := <-in
        if i%p != 0 {out <-i}
    }
}

func main() {
    ch := make(chan int)
    go generate(ch)
    bound := readFromUser()
    for i := 0; i < bound; i++ {
        prime := <-ch
        fmt.Println(prime)
        ch1 := make(chan int)
        go filter(ch, ch1, prime)
        ch = ch1
    }
}
Frequency of concurrency patterns in 865 projects

<table>
<thead>
<tr>
<th>Feature</th>
<th>projects</th>
<th>proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td>711</td>
<td>82%</td>
</tr>
<tr>
<td>go in (any) for</td>
<td>500</td>
<td>58%</td>
</tr>
<tr>
<td>go in bounded for</td>
<td>172</td>
<td>20%</td>
</tr>
<tr>
<td>go in unknown for</td>
<td>474</td>
<td>55%</td>
</tr>
</tbody>
</table>

NB: 45% of channel as formal parameters had a specified direction.
## Frequency of concurrency patterns in 865 projects

<table>
<thead>
<tr>
<th>Feature</th>
<th>Projects</th>
<th>Proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td>711</td>
<td>82%</td>
</tr>
<tr>
<td>go <strong>in (any)</strong> <strong>for</strong></td>
<td>500</td>
<td>58%</td>
</tr>
<tr>
<td>go in bounded <strong>for</strong></td>
<td>172</td>
<td>20%</td>
</tr>
<tr>
<td>go in unknown <strong>for</strong></td>
<td>474</td>
<td>55%</td>
</tr>
<tr>
<td>chan in (any) <strong>for</strong></td>
<td>111</td>
<td>13%</td>
</tr>
<tr>
<td>chan in bounded <strong>for</strong></td>
<td>19</td>
<td>2%</td>
</tr>
<tr>
<td>chan in unknown <strong>for</strong></td>
<td>103</td>
<td>12%</td>
</tr>
<tr>
<td>channel aliasing in <strong>for</strong></td>
<td>14</td>
<td>2%</td>
</tr>
</tbody>
</table>

NB: 45% of channel as formal parameters had a specified direction.
## Frequency of concurrency patterns in 865 projects

<table>
<thead>
<tr>
<th>Feature</th>
<th>projects</th>
<th>proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td>711</td>
<td>82%</td>
</tr>
<tr>
<td>go in (any) for</td>
<td>500</td>
<td>58%</td>
</tr>
<tr>
<td>go in bounded for</td>
<td>172</td>
<td>20%</td>
</tr>
<tr>
<td>go in unknown for</td>
<td>474</td>
<td>55%</td>
</tr>
<tr>
<td>chan in (any) for</td>
<td>111</td>
<td>13%</td>
</tr>
<tr>
<td>chan in bounded for</td>
<td>19</td>
<td>2%</td>
</tr>
<tr>
<td>chan in unknown for</td>
<td>103</td>
<td>12%</td>
</tr>
<tr>
<td>channel aliasing in for</td>
<td>14</td>
<td>2%</td>
</tr>
<tr>
<td>channel in slice</td>
<td>31</td>
<td>4%</td>
</tr>
<tr>
<td>channel in map</td>
<td>8</td>
<td>1%</td>
</tr>
<tr>
<td>channel of channels</td>
<td>49</td>
<td>6%</td>
</tr>
</tbody>
</table>

NB: 45% of channel as formal parameters had a specified direction.
Frequency of concurrency patterns in 865 projects

<table>
<thead>
<tr>
<th>Feature</th>
<th>Projects</th>
<th>Proportion</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td>711</td>
<td>82%</td>
</tr>
<tr>
<td>go in (any) for</td>
<td>500</td>
<td>58%</td>
</tr>
<tr>
<td>go in bounded for</td>
<td>172</td>
<td>20%</td>
</tr>
<tr>
<td>go in unknown for</td>
<td>474</td>
<td>55%</td>
</tr>
<tr>
<td>chan in (any) for</td>
<td>111</td>
<td>13%</td>
</tr>
<tr>
<td>chan in bounded for</td>
<td>19</td>
<td>2%</td>
</tr>
<tr>
<td>chan in unknown for</td>
<td>103</td>
<td>12%</td>
</tr>
<tr>
<td>channel aliasing in for</td>
<td>14</td>
<td>2%</td>
</tr>
<tr>
<td>channel in slice</td>
<td>31</td>
<td>4%</td>
</tr>
<tr>
<td>channel in map</td>
<td>8</td>
<td>1%</td>
</tr>
<tr>
<td>channel of channels</td>
<td>49</td>
<td>6%</td>
</tr>
</tbody>
</table>

NB: 45% of channel as formal parameters had a specified direction.
### Known bounds of `for` loops containing `go`

<table>
<thead>
<tr>
<th>bound</th>
<th>mean</th>
<th>std</th>
<th>min</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>280.53</td>
<td>1957.50</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>100</td>
<td>50000</td>
</tr>
</tbody>
</table>

- 55% of projects use for loops with **unknown bounds**.
- 788/918 (86%) occurrences of a creation of a goroutine within a bounded `for` were located in a **test file**.
- Unfolding loops is probably not a good idea!
Conclusions

- 76% of the projects use communication channels.
- The number of primitives per channel is low, suggesting that channels are used for simple synchronisation protocols.
Conclusions

- 76% of the projects use communication channels.
- The number of primitives per channel is low, suggesting that channels are used for simple synchronisation protocols.
- On average, just under half of the packages of the Go projects we analysed contain concurrency features,
- around 20% of files contain concurrency-related features.
Conclusions

- 76% of the projects use communication channels.
- The number of primitives per channel is low, suggesting that channels are used for simple synchronisation protocols.
- On average, just under half of the packages of the Go projects we analysed contain concurrency features,
- around 20% of files contain concurrency-related features.
- Synchronous channels are the most commonly used channels.
Conclusions

- 76% of the projects use communication channels.
- The number of primitives per channel is low, suggesting that channels are used for simple synchronisation protocols.
- On average, just under half of the packages of the Go projects we analysed contain concurrency features,
- around 20% of files contain concurrency-related features.
- Synchronous channels are the most commonly used channels.
- 58% of the projects include thread creations in for loops.
- Channel creation in for loops is uncommon.
Thanks.

Any questions?