UiO : **Department of Informatics**
University of Oslo

Detection of Bugs and Code Smells through Static Analysis of Go Source Code

Christian Bergum Bergersen

Master's Thesis Autumn 2016

Detection of Bugs and Code Smells through Static Analysis of Go Source Code

Christian Bergum Bergersen

August 9, 2016

Abstract

Go is a new language especially known for its speed, simplicity, and concurrency approach. The language has gained some promising momentum as a newcomer, several big projects like *Docker* is implemented in the language. As Go increases popularity among developers, the software industry is still hesitating to fully include the usage of the language in their projects.

One of the reasons is the lack of tool support around the language, i.e. the lack of tools for automated static analysis of Go source code. The industry strives for implementation and delivery of defect free software to their customers. In the struggle for delivering *beautiful* and defect free code are they completely dependent on static analysis of source code to reveal defects and suspicious code at an early stage in the development phase.

This thesis works out a set of basic definitions of *bugs* and *code smells* in Go, and implements the first version of the static analysis tool detecting violations of these definitions by scanning the source code. The tool is also shipped as a SonarQube plugin for continuously analyzing and measuring code quality.

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

I would like to thank both my supervisors Martin Steffen and Volker Stolz for bearing out with me, as the thesis started out as an idea rather different than what it ended. Thanks for letting me make a more practical approach to the field of formal methods! I would also thank Computas AS, my further employer, for giving me insight into the software industry through my three-year long internship, during my studies.

Last, but not least, thanks to my parents Liv Ragnhild Schrøder and Bjørn Kjell Bergersen for your endless support and love throughout my studies.

Chapter 1

Introduction

1.1 Motivation

Today the society is more and more software driven, i.e. in both forms of safety and business critical systems, as the demand for rapid extensions, improvements, and new features increases, so does the size and complexity of the projects delivering these systems. One of many challenges in these projects is to ensure some degree of code quality.

The overall goals of these projects are to deliver software according to specification within the cost and time limit given. To achieve this one is blessed with good software engineering principles to avoid pitfalls throughout the project, in software projects, a significant source of pitfalls are considered the code presence of defects and architectural symptoms that may indicate deeper problems.

The field of software testing has emerged to solve or damming up the challenge to reveal defects and other suspicious constructs in software. Static analysis of source code is one software testing technique often continuously applied in the implementation phase as code is written.

In many industrial projects, the usage of static analysis is in all stakeholder's interests as defects and erroneous code are detected at an early stage, reducing the risk for delayed project progress and code breaches.

Regardless of how technical promising a language is for solving challenges in industrial projects, its strength cannot weight up with the lack of tool support for static code analysis in the language. This claim was confirmed during a presentation of the Go programming language[6, 9, 17] for a group considering new technologies in the Norwegian software company Computas AS. They sought after general tool support for automated static analysis for Go, i.e. tools like PMD[8] or FindBugs[14] which exist for Java, they made it clear that tool support for static analysis in a language is essential for quality assurance in their projects.

Bugs	Valid syntactic code that will compile, but will cause runtime panic or other behavior that is technically incorrect.
Code Smells	Valid syntactic code that will compile, but may be symptoms of deeper problems indicating design weakness, but not classifies as technical errors.
Cyclomatic Complexity	Measures function and method code complexity through the code.

Table 1.1: The tool inspects the code after bugs, code smells and high cyclomatic complexity in functions and method.

1.2 Goals and Contributions

The goal of this thesis is to meet the demand for tool support raised in the meeting with Computas AS. Such a static analyzer tool would be helpful for the entire Go community and may help the industry a step closer to include Go in their projects.

To achieve the goal of covering a static analysis tool in Go, the main contribution in this thesis is to design and implement the first version of such a tool, greatly inspired by PMD and FindBugs as they have gained quite a momentum in the Java community. A SonarQube plugin for the tool is also provided to cover the industry demand and support for continuous inspection through a quality management platform. Table 1.1 shows an overview of what the tool detects in its analysis.

1.2.1 Extensibility

The opportunity to add new static checking algorithms to detect new types of code smells and bugs is crucial as new harmful code constructs, and code habits among developers are discovered.

1.2.2 Correctness

Extensive use of unit tests is implemented in the tool to reach the goal of correct identification of code smells and bugs in the tool, achieving reduced flagging of legal code constructs and avoiding ignorance of illegal code constructs defined in the algorithms in the tool.

1.2.3 Precision

Besides ensuring correct identification of common bugs and code smells using unit tests, extensive precision measurements are done by applying the tool to uses cases like analyzing the source code of prominent Go projects and the source code of the tool itself. The result from these analyses is used to calibrate the different algorithms in the analyzer.

1.2.4 Portability

The tools must be platform independent, it means that it should be possible to compile and execute the tool on all operating systems and platforms which is supported by Go itself, naturally since the tool is used to inspect source code in the development phase of Go code.

It is also naturally that the tool supports all major operating systems and architectures as the tool is integrated into other continues inspection tools for project management, either by directly invoking the tool or by consuming analysis results output in JSON [18] format.

Specifically, this means that the tool at least supports the following operating system and architectures:

- `linux/amd64`.
- `darwin/amd64` (Mac OS X).
- `windows/amd64`.

1.3 Thesis Overview

The next chapter will give a brief introduction to Go as a programming language, focusing on distinctive characteristics in the language differing from other mainstream languages which is critical to understand to evaluate code constructs as either bugs or code smells.

Further, the chapter will give a historical and theoretical description of the process of measuring cyclomatic complexity and the process of detecting code smells and bugs.

After introducing the theoretical background, Chapter 3 dives into the implementation of the linting tool, showing why and how code constructs are considered as bugs or code smells, and therefore detected by the tool.

After introducing the theoretical background, a review of the detection algorithms in the tool is provided in chapter3, together with code examples describing why and how the tool discovers bugs and code smells.

Chapter 4 addresses the practical use and the results of running the tool on Go source code for the tool itself and the *Go compiler*.

Chapter 4 ends the thesis with an evaluation and conclusion of the work and result carried out.

1.4 Project Website

All code written as part of the thesis, and the thesis itself are collected and available on http://www.mn.uio.no/ifi/english/research/groups/pma/completedmasters/2016/bergersen

In addition, the tool's repository is located on https://github.com/chrisbbe/GoAnalysis for public access and further contributions.

Chapter 2

Background

2.1 The Go Programming Language

Dennis Ritchie created the C programming language[13] as a high-level language. C provides an abstract machine model that closely reflected the contemporary computer architecture at the time. Since the release of C in 1978, the abstraction has become less like the real hardware of today. C's abstraction model represents a single processor and a single block of memory. The development of computer architecture has advanced a lot, extending the old abstraction model with the addition of multicore processors, several levels of caches and more advanced technology. These days, even mobile phones have multicore processors!

Multiple processors are the most noticeable change in the computer architecture for programmers, as one has to implement consciously programs to utilize all the cores on a processor. To implement programs in parallel in languages like C requires significant effort, as the language is originally developed with the old single core abstraction model in mind.

In 2007, a team at Google consisting of Robert Griesemer, Rob Pike, and Ken Thompson started working on ideas for a new modern programming language. One of the issues they wanted to address in the language is the challenges with concurrency.

2.1.1 General

Go is an object-oriented, concurrent, imperative and strongly typed language where syntax and declaration derive from the C language. In opposite to other languages like Java, Go's syntax is small and straightforward, although its small syntax Go is still very expressive. In opposite to traditional object-oriented design, Go does not support inheritance. Instead, interface types, duck typing, and first class functions are provided. Listing 2.1 shows the standard Hello World example in Go.

2.1.2 Syntax and Semantic

People with knowledge of C will recognize the syntax of Go as similar, both Go and C shares the concept of pointers and structs. The semantic differ

```go
package main

import "fmt"

func main() {
  fmt.Println("Hello World Go!")
}
```

Listing 2.1: Traditional Hello World in Go.

as structs in Go is a part of the object-oriented design in Go by allowing to associate methods with a struct type, pointer arithmetics is in opposite with C not allowed in Go.

Go provides all common built-in types like strings, numeric and Boolean types. Go also provides some built-in datatypes as a hash-map, arrays, slices and channels for communication.

Control Structure

The control structures of Go is different from other mainstream languages like Java and C in the way that there is only one type of loops, the flexible `for` loop.

Standard conditional control structures as `if-else` and `switch` statements exist and behaves as in other mainstream languages. Listing 2.2 shows the two forms of loops in go.

```go
  sum := 0
  for i := 0; i < 10; i++ {
   sum += i
  }

  alphabeth := []string{"A", "B", "C"}
  for index, line := range alphabeth {
    fmt.Printf("%d) %s\n", index, line)
  }
```

Listing 2.2: The two forms of looping in Go.

Functions and Methods

Functions are first class citizen in Go, meaning that it is possible to pass functions as arguments to other functions, use functions as return values of other functions and execute functions as asynchronous threads, also called goroutines. Functions can also return multiple values using named return variables, which are handy in many situations, e.g., in a situation where a function returns coordinates (x, y) in the Cartesian coordinate system.

In addition to pass-by-value as parameter passing technique, Go supports pass-by-reference using pointers. Pointers are a well-known feature to C programmers, often confusing and error prone at first sight. The pointer is a variable that holds another variable's memory address. In other words, the pointer variable holds a memory address to a location in memory containing the actual value. As pointers are passed as arguments, they are also returned by functions and methods.

Like C, Go represents pointers with the * operator followed by the type of the stored value which the pointer references in memory. Operator & is used to dereference the memory address held in a pointer. Dereferencing the pointer gives us the memory address to the location where the actual value. In the first line of the program in listing 2.3, the argument passed to the function is a pointer variable of type `int`, which holds the memory address to a location in memory holding the `int` value. At line 5 the memory address is passed to the function.

A great source of unexpected behavior in languages supporting pointers is unexpected behavior as developers are not aware of a variable types i.e. if the variable holds a pointer or the value itself.

```
1 func inc(v *int) {
2     *v++;
3 }
4 x := 1
5 inc(&x) //Passing the memory adress to x.
6 fmt.Println(x) //Prints 2.
```

Listing 2.3: Pass-by-reference by using a pointer.

Data Structures

As mentioned Go's three built-in data structures are arrays, maps, and. Structs play a major role in Go's way to do object-orientation and is described closer in the section about object-orientation.

A more interesting form of an array is a slice; a slice is a subset of an array represented by two pointers referencing the start and end -index of the underlying array. Listing 2.4 shows an example of "slicing" where variable `u` and `t` are slicing the underlying array `s`. Slicing works by pointing the slice start and end index pointer on the subset of the array being "sliced". Notice that altering an element in `u`, `t` or `s` will be reflected down to the underlying array.

Allocation in Go has been a great source of confusion and discussion in the Go community. There exist two allocations primitives, `new()` and `make()`, the usage is simple: `make()` is only used to allocate channels, maps, and slices. The difference between these two allocation primitives is that `make()` allocates the memory chunk and initializes the data structure for which type its given as argument. In opposite with `new()` which only allocates the memory chunk for the type given as argument.

The usage of structs are closely related to the object-oriented design of Go and is referenced further in the Object-orientation section.

```
s := []string{"A", "B", "C", "D", "E", "F", "G"}
t := s[:3]  //t = ["A", "B", "C"]
u := s[4:6] //u = ["E", "F", "G"]
```

Listing 2.4: Slicing a subset of array `s` into variable `t` and `u`.

```go
package main

import "fmt"

type Coordinate struct {
        N, E float32
}

func (cord *Coordinate) String() string {
        return fmt.Sprintf("%fN, %fE", cord.N, cord.E)
}

func main() {
        nordkapp := &Coordinate{N:71.16, E: 25.78}
        fmt.Printf("Nordkapp: %s\n", nordkapp)
}
```

Listing 2.5: Implementation of a custom type with a corresponding method.

2.1.3 Object-orientation

To determine if Go is an object-oriented language, one has to look on the origination of the concept introduced in the first object-oriented language Simula, where objects, classes, inheritance, subclasses and virtual methods where introduced, revolutionizing the paradigm of programming.

Go does not have classes the way Simula or Java has, where the class both encapsulate the data in the form of variable and behavior in the form of methods that can act upon the data. Go's approach object-orientation is enabled through the use of structs to hold data, and a particular form of functions representing behavior, these functions is bound to the struct type and is called methods instead of functions. The idea is similar to classes but does not encapsulate the data and methods into a single entity.

¨Listing 2.5 shows the principle of object-orientation in Go, a custom struct type is defined, and a corresponding method for that type is implementing by specifying the type as the receiver in the function signature.

The reason why Go has chosen not to go for the traditional way of object orientation with inheritance and instead choose composition is clearly to avoid explicit relationship between types and interfaces, leaving out the task of managing the type hierarchy. Go takes the approach of Duck Typing instead.

Duck Typing

Instead of explicit declare and specify how two or more types relates, Go uses duck typing which allows a type to act as any interface type if the type implements the methods defined by the interface it wants to act as. This way a type can automatically satisfy many interfaces at once without the complexity of multiple inheritances, giving the opportunity to add new interfaces along the way without changing the original type. In other

```go
package main

import "fmt"

type Duck interface {
        Quack()
        Walk()
}

type Bird struct {
        name string
}

func main() {
        bird := Bird{"Donald"}
        DoIt(bird)
}

func DoIt(d Duck) {
        d.Walk()
        d.Quack()
}

func (b Bird) Walk() {
        fmt.Printf(b.name + " walks\n")
}

func (b Bird) Quack() {
        fmt.Printf(b.name + " quacks\n")
}
```

Listing 2.6: Classic example of Duck Typing, where the `Bird` type implements methods in the `Duck` interface, allowing the `Bird` to behave like the `Duck`.

words, one is determining the suitability of an object by looking at the presence of methods, rather than looking at the type of the object.

Program 2.5 contains a custom-type `Bird` and the interface `Duck` defining two methods. The `Bird` type satisfies the interface by implementing both methods `Walk()` and `Quack()` allowing the `Bird` type variable `name` of to be passed as argument to function DoIt(), i.e. the `Bird` behaves like the `Duck`.

Often duck typing is shortly phrased: *"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck"*.

2.1.4 Concurrency

Concurrency is the real strength of Go. The language provides comfortable support for concurrency using goroutines and channels. Goroutines is a lightweight form of thread shipped with built-in channels for secure communication between goroutines. Go's high-level concurrency features goroutines and channels make it possible to focus on the business logic instead of writing error-free concurrent programs. Still, there are possibilities to do concurrency the old fashion way of using synchronization primitives like mutexes to ensure mutual exclusion on shared data. However, it should be avoided as long as possible according to Go's concurrency mantra: *Don't communicate by sharing memory, share memory by communicating*.

Goroutines

Syntactically a goroutine is a function call that executes asynchronously, by adding the keyword `go` before a function call causes the function to execute as an independent new goroutine. Listing 2.7 shows an example of the traditional consumer-produces problem where two goroutines are communicating through channels; this example is rather trivial as only strings are sent between the goroutines, a channel could be declared to communicate any type. Notify the `done` channel used by the main goroutine as a barrier to ensure that both goroutines finish before the main goroutine.

Channels

The advantage of Go is that it follows C.A.R Hoare's [10] formalism to facilitates communication between goroutines. In his paper, Hoar defines communication channels between processes. Go adopts channels as the preferred way to share data between goroutines. Go channels can be both unbuffered synchronous and buffered asynchronous channels, both types of channels are bidirectional by default, but one can constraint a channel only to send or only to receive.

Unbuffered synchronization channels introduce the possibilities of deadlock situations, e.g. in a situation where two goroutines are trying to send on the same channel, waiting for each other to receive the message. Infinitely blocked until someone reads the message it tries to send. Notice that a single infinitely blocked goroutine because of a send action on a channel is not denoted as deadlocked!

Since unbuffered channels block until its action completes, unbuffered channels can be exploit to do conditionally synchronization; the danger is that two goroutines block each other causing a deadlock.

Listing 2.8 shows a more sophisticated usage of goroutines and channels where the `select` built-in select statements is used to listen on multiple goroutines at once, in this example `main()` will timeout other connections if data is not received within the time limit defined.

2.2 Static Code Analysis

Static program analysis [7] is one of the two main groups of testing techniques found in software testing as shown in figure 2.1. In contrast to dynamic software testing, static program analysis is performed without actually executing the program. The analysis is either in most cases conducted on the source code or the compiled machine code. The

```go
package main

import "fmt"

func producer(msg chan string, done chan int) {
  fmt.Println("Producing message...")
  msg <- "ack" //Sending message to channel.
  done <- 1 //Signaling termination.
}

func consumer(msg chan string, done chan int) {
  msgvar := <- msg //From channel into msg.
  fmt.Printf("Consuming message:  %s\n", msgvar)
  done <- 1 //Signaling termination.
}

func main() {
  done := make(chan int) //Unbuffered channel
  chan1 := make(chan string) //Unbuffered channel.
  go consumer(chan1, done) //Start async goroutine.
  go producer(chan1, done) //Start async goroutine.
  <-done //Waiting for receive.
  <-done //Waiting for receive.
  fmt.Println("All Goroutines are done!")
}
```

Listing 2.7: Goroutines communicating and synchronizing through channels.

challenge is to compute reliable approximate information about the dynamic behavior of the program.

The term static analysis is typically understood by the means of an automatic tool supported analysis, but the term also includes the scope of static manual techniques for analysis. The process of inspection and review code by humans is a manual static analysis process rapidly applied in addition to automated static analysis in industrial software projects.

2.2.1 Linting: Detecting Bugs and Code Smells

`Lint` [11] is originally a Unix utility which examines C source program, the tool looks for potential bugs and obscurities, it also detects some wasteful, or error prone, constructions which nevertheless are, strictly speaking, legal.

A brief list of some of the suspicious constructs `Lint` will detect:

- Unused variables and functions. (Not bugs, but a source of inefficiency).
- Usage of variables before they are set.
- Assignment of longs to ints. (Accuracy is lost).

Today lint and linting has evolved into to a collective term, generally describing the tool and process of performing code analysis in different

```go
package main

import (
        "time"
        "fmt"
)

func timeout(t chan bool) {
        time.Sleep(5000000)
        t <- true
}

func readString(s chan string) {
        time.Sleep(5000000 * 2)
        s <- "Hello"
}

func main() {
        t := make(chan bool)
        s := make(chan string)
        go readString(s)
        go timeout(t)
        select {
        case msg := <-s:
                fmt.Printf("Received: %s\n", msg)
        case <-t:
                fmt.Printf("Timed out!\n")
        }
}
```

Listing 2.8: Select statement listening on two channels, giving the ability to timeout connections.

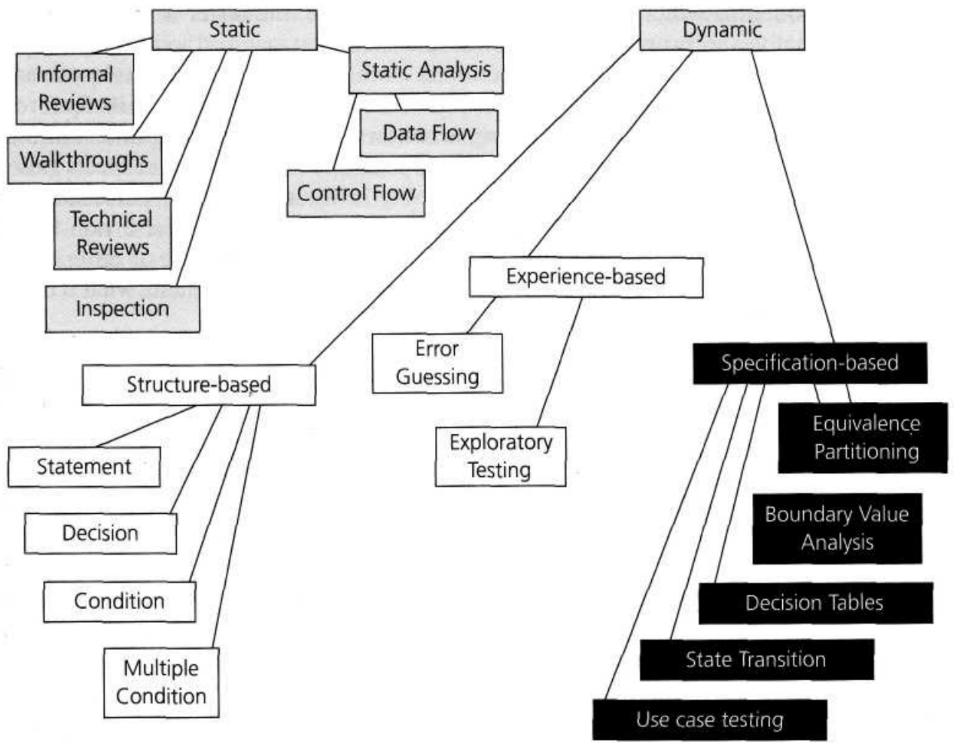


Figure 2.1: Static and dynamic are the main groups of testing techniques.

languages. In opposite to the original Lint tool, the tools of today contain loads of rules which the tool is checking that source code do not violate, these rules are mainly divided into the two groups of *bugs* and *code smells*.

It is also worth mentioning that Linting is a heuristic method which is not guaranteed to be perfect or optimal nor used as a firm indicator of program correctness. Linting often needs to be used in interaction with humans determine the results, as the linter might detect false-positives or false-negatives, or in situations where the developer is smarter than the tool, e.g. in situations where the tools complain about code constructs the developer have made intentionally. In the latter example, there is normal in Linter tools to have a mechanism to shut up the linter to report a violation of specified rules, described closer in section 3.3.

Code Smells

Bade smells are code constructs or structural characteristics in code that *might* be hard to understand, extend and maintain. The occurrence of smelly code can be handled through the process of refactoring the code. Martin Fowler defines refactoring as the process of improving the internal structure of a program without altering its external behavior [12].

Chapter 3 in Martin Fowler's book defines bad smells in code through some code constructs examples, all these examples are more generic towards object oriented languages generally than to a specific language, e.g. the subject of detection duplicated code and long parameters lists.

In the first version of the tool implemented in this thesis, the focus will not be the more general definitions of code smells as described by Fowler, but more against code smells and obvious bugs found in the Go language. These definitions of smelly code are found through the process of gaining both theoretical and practical experience with the semantic and syntax of the language.

2.2.2 Cyclomatic Complexity

Cyclomatic complexity is a software measurement developed by Thomas J. McCabe in his article [15] where he proposes a mathematical technique that allows us to identify software modules in programs that will be difficult to test or maintain.

McCabe describes in his article that an original practice to ensure reasonable modularization in programs was to limit program modules in physical size, but as shown by McCabe, the technique is not adequate. A pretty small program around 50 lines with `IF-THEN` conditions will have around 33.5 million distinct control paths, with other words *"Practical impossible to test!"*.

The approach to computing cyclomatic complexity in object-oriented languages is to measure and control the number of paths through functions and methods. Achieved by using the control flow graph of the function or method as an intermediate representation of the flow of control, the nodes of the graph correspond to basic-blocks in the function or method which we are measuring. The connecting edges between nodes in the graph corresponds to which basic-block i.e. node that might be executed immediately after the node which has the outgoing edge.

The cyclomatic number $v(G)$ of a graph G is computed using equation 2.1, where e is the number of edges, n is the number of vertices, p is the number of strongly connected components in the control flow graph.

$$v(G) = e - n + p \tag{2.1}$$

In section 3.2 we are implementing McCabe's approach to measuring the cyclomatic complexity of Go source code in the tool, first by identifying basic-blocks in the code and then convert these basic-blocks and their relationship into a control flow graph used to calculate the cyclomatic complexity.

Application

McCabe describes the intention of applying cyclomatic complexity (CC) as moving the focus away from limiting programs in physical size and instead encourage programmers to calculate the CC for their function and methods. With the goal of limiting functions and methods in complexity size by splitting up functions and methods into smaller pieces, increasing the program's understandability and maintainability.

Defined by McCabe the upper bound value of 10 is reasonable but not a magical value, i.e. situations with a big trivial switch statement exceeding

CC Value	Risk Evaluation
1 - 10	Simple program, without much risk.
11 - 20	More complex, moderate risk.
21 - 50	Complex, high risk programs.
> 50	Practical untestable, very high risk.

Table 2.1: Cyclomatic complexity risk threshold values.

the value of 10. The Software Engineering Institute has defined [16] the risk matrix based on threshold levels of CC as seen in table 2.1.

Basic-Blocks

A basic-block is a block of code containing a straight line of statements with no branch statements expect at the start and end of the block. A branch statement target begins a basic-block, while a branch statement ends a basic-block.

The nodes of a control flow graph are the basic-blocks, and the edges are formed from the conditional and unconditional jumps in the code. The control flow graph, together with each of its basic-blocks can be constructed by a single iteration of the source code. The following algorithm identifies and generates basic-blocks:

1. Determine the set of leaders: Use the following rules.
   (a) The first statement in the program is a *leader*.
   (b) Any statement that is the target of a conditional or unconditional statement is a *leader*.
   (c) Any statement that immediately follows a conditional or unconditional statement is a *leader*.
2. Construct the basic-blocks using the *leaders*. For each leader, its basic-block consists of the leader and all statements up to but not including the next leader or the end of the program.
3. The blocks control may transfer after reaching the end of a block are that blocks *successors* blocks, opposite the blocks from which control may have come when entering a block are called that block's *predecessors*.

Listing 2.9 shows the algorithm applied, the code is divided into basic-blocks, where the successors are as follow:

- Basic-Block #0 $\rightarrow$ Basic-Block #1.
- Basic-Block #2 $\rightarrow$ Basic-Block #3.
- Basic-Block #3 $\rightarrow$ Basic-Block #{4,5}.
- Basic-Block #4 $\rightarrow$ Basic-Block #3.

```go
/* ------------------------------ */
package main

func main() {                   // BASIC-BLOCK #0
/* ------------------------------ */
    fmt.Println(gcd(33, 77))
}                               // BASIC-BLOCK #1
/* ------------------------------ */
func gcd(x, y int) int {        // BASIC-BLOCK #2
/* ------------------------------ */
    for y != 0 {                // BASIC-BLOCK #3
/* ------------------------------ */
        x, y = y, x % y
    }                           // BASIC-BLOCK #4
/* ------------------------------ */
    return x
}                               // BASIC-BLOCK #5
/* ------------------------------ */
```

Listing 2.9: Go code solving *Greatest Common Divisor*, divided into basic-blocks. Function *main()* consist of block 0-1, while *gcd()* consist of block 2-5.

Note that basic-block #1 don't have basic-block #2 as successor, that is because we are only concerned with the cyclomatic complexity for each function/method. And therefore only need the control flow graph on function/method level, according to the algorithm it would be right to add block #2 as successor to block #1.

Also, note that *successor* blocks are decided by the condition statement semantics, the `for` loop in basic-block #3 might evaluate to `FALSE` and jump to block #5, similarly will the `for` body (block #4) when finish jump up to block #3 for condition evaluation.

Control Flow Graph

Frances E. Allen in here article [1] describes basic control flow relationships expressed in a directed graph, later known as control-flow graphs. Today a control-flow graph is often used in static analysis to show the flow of control across hole systems, packages, files or single methods and functions. Since we are doing a measurement of the cyclomatic complexity on function and method level, we can limit the control-flow graph computation to function and method level.

Basic-block generation as described in section 2.2.2 in many ways forms the control-flow graph itself, each basic-block holds a list of successor blocks which the control might flow to. The control-flow construction involves converting the basic-block data-structure over to a directed graph, a directed graph package has been developed for the purpose, this graph package also implements *Tarjan's* [3] linear depth first algorithm to extract the set of strongly connected components through, which are necessary to compute the cyclomatic complexity.

$$\begin{tikzcd}
\text{Start} \arrow[r] & A \arrow[r] & B \arrow[r, bend right] \arrow[dr] & C \arrow[l, bend right] & \\
 & & & D \arrow[r] & \text{Exit}
\end{tikzcd}$$

Figure 2.2: The flow graph for the *Greatest Common Divisor* program in 2.9.

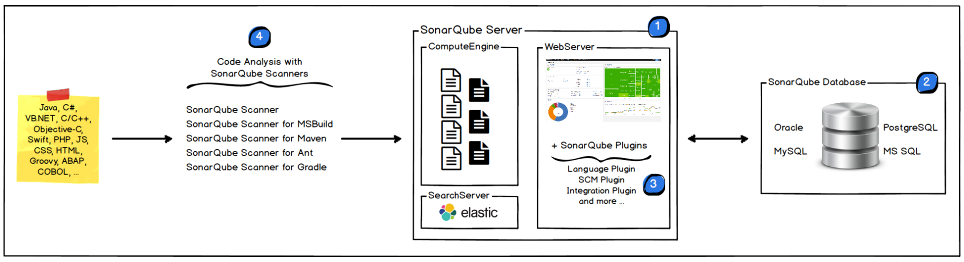

Java, C#, VB.NET, C/C++, Objective-C, Swift, PHP, JS, CSS, HTML, Groovy, ABAP, COBOL, ...
4
Code Analysis with SonarQube Scanners
SonarQube Scanner
SonarQube Scanner for MSBuild
SonarQube Scanner for Maven
SonarQube Scanner for Ant
SonarQube Scanner for Gradle
SonarQube Server
1
ComputeEngine
SearchServer
elastic
WebServer
+ SonarQube Plugins
Language Plugin
SCM Plugin
Integration Plugin
and more ...
3
SonarQube Database
2
Oracle
MySQL
PostgreSQL
MS SQL

Figure 2.3: SonarQube architecture.

Figure 2.2 shows the control-flow graph drawn from the Go source code listed in 2.9, the graph is easily generated by iterating over the basic-blocks and then iterate over each blocks successor block, drawing an edge between the block itself and all successor blocks. As seen in the figure, basic-block **B** and **C** visualizes the semantic in a `for` loop, where the control flows in a loop until the condition in **B** becomes `FALSE`.

2.3 SonarQube

SonarQube[4] is a Java-based quality management platform used in projects to continuously analyze and measure source code quality progression over time. Plugins are powering SonarQube's ability to perform analysis of source code in different languages.

Figure 2.3 shows the architecture of SonarQube where the SonarQube scanner is the client downloading available plugins from the SonarQube server, applying the analysis plugin for the source code language being scanned. When the SonarQube scanner is finished scanning the result is uploaded to the SonarQube server. Analysis results in visualized through the web interface on the SonarQube, giving the ability to compare the progression of the source code quality over time in the project.

Chapter 3

Implementing the Static Go Analyzer

This chapter describes the general high-level implementation of the static analyzer tool named GoAnalyzer.

The tool is built on the knowledge of Go's syntax and semantics introduced in Section 2.1. This knowledge is used to identify code constructs that are possible to create with Go syntax, but semantically is considered as suspicious or error prone. To automatically detect semantically odd programs, rules are defined on how syntax compositions should behave to reveal violations when scanning source code.

Section 3.4 contains a list of defined rules implemented in the tool, each rule contains a brief description of the reason why the code construct is either considered as a bug or code smell, and how enforcement of the rule is carried throughout the analysis. Examples of the tool in action are also provided.

The last section discusses the challenge of implementing the GoAnalyzer tool into SonarQube, discussing different proposals and the final solution implemented.

Source code for the GoAnalyzer tool and the SonarQube plugin resides in the repository described in section 1.4.

3.1 Architecture

The main contribution in this thesis is the implementation of an automatic static analyzer tool for Go source code; the tool takes as input a set of files to be checked and outputs warnings about suspicious code constructs as *code smells* and *bugs* identified in the code.

The tool is *rule* driven. This means that there are clearly defined rules about how given code constructs in the code should be handled. The tool checks whether these rules are broken. Breaches are flagged as violations.

Development of the set of rules strains from experience with both general programming languages and specifically Go as language, i.e. the rules comply code constructs which are commonly agreed upon as *code smells* and *bugs* throughout discussion groups and literature.

The tool is meant for guidance, and do not prove the correctness of code. Still, the tool will be helpful to identify code constructs that programmers may be unaware of, on the other side the tool might warn about code constructs that the programmer does intentionally. For intentionally rule breaches, the tool supports annotations, which suppresses flagging of rule violation.

The tool is implemented entirely in Go, only depending on the standard library, the tool can therefore be compiled down to an executable file for every platform supported by Go, without any third party dependencies.

The tool provides a command-line interface with the ability to specify folder paths containing source code through the `-dir` argument, the tool will then recursively search the folder hierarchy after `.go` files and perform analysis on these files, the result is outputted to console. There is also implemented a `-json` option switch that enables structured JSON output of the analysis result, this option is utilized by the SonarQube plugin written in Java by calling the tool through the shell and consume its JSON output.

3.2 Measure Cyclomatic Complexity

The process of measuring the cyclomatic complexity of plain source code consist of three steps. The firs step involve iterating over and divide the source code into basic-blocks. The second step consists of converting the set of basic-blocks into an intermediate representation of the control flow behavior in the program, for that a directed graph is applied. In the third step, information is extracted from the control flow graph to feed it into the cyclomatic complexity equation 2.1 to get the cyclomatic complexity measure.

In the tool, only cyclomatic complexity at function and method level are measured. That is because only statements that increase cyclomatic complexity can be declared inside functions and methods.

3.2.1 Identifying Basic-Blocks

Dividing source code into basic-blocks is implemented in the tool by recursively walk the abstract syntax tree (AST) gained from the parsing function located in the parser package in Go's standard library. The advantage of walking the AST in opposite to iterate through the plain source code line by line is that the AST gives the possibility to decide if a node ends or starts a basic-blocks based on whether the node type represents a source or target of a branching statement. Determination of blocks successors is done by utilizing the natural hierarchical properties provided by the tree structure in which a source or target node is located, i.e. a parent's successor block is all the blocks generated by its child nodes.

Package `bblock` implemented in the tool contains a data type to represent basic-blocks, various methods to perform operations on the basic-block and functions to identify and retrieve basic-blocks from source

```
0) FUNCTION_ENTRY (EndLine: 8)
    -> (1) FOR_STATEMENT (EndLine: 9)
1) FOR_STATEMENT (EndLine: 9)
        -> (2) FOR_BODY (EndLine: 11)
        -> (3) RETURN_STMT (EndLine: 12)
2) FOR_BODY (EndLine: 11)
        -> (1) FOR_STATEMENT (EndLine: 9)
3) RETURN_STMT (EndLine: 12)
4) FUNCTION_ENTRY (EndLine: 15)
        -> (5) RETURN_STMT (EndLine: 18)
5) RETURN_STMT (EndLine: 18)
```

Listing 3.1: Verbose output from the basic-block identification algorithm.

code. Listing 3.1 shows the output from printing the set of basic-blocks identified from the source code in Listing 2.9.

3.2.2 Control Flow Representation

A package `graph` is implemented in the tool as a neat generic directed graph package providing construction methods to build graph representations with directed edges. The graph also contains a depth-first search algorithm and a strongly connected components algorithm that is needed to measure cyclomatic complexity in control flow graphs.

The graph package is generic by allowing any type that satisfies the interface to be held in a node, in this way other algorithms can use the package. The flow of control in methods and functions is represented using the package.

Strongly Connected Components

The measure of cyclomatic complexity requires the number of strongly connected components in the directed graph. The choose of the algorithm to provide the ability to compute the set of strongly connected components is Tarjan's algorithm [3]. The algorithm is effective in the fact that the running time complexity is linear as nodes(V) is only visited a single time, and its edges(E) are at most followed once:

$$O(|V| + |E|)$$

Converting basic-blocks to control flow graph

The basic-block type implements the directed graph interface allowing us to represent the control flow graph using the graph package.

The job of identifying the relationship between basic-blocks is performed within the basic-block algorithm that identifies blocks successors. A basic-blocks successor can directly be interpreted as a directed edge between the two nodes corresponding to the basic-block and the successor. Listing 3.2 contains the code that iterates over all basic-blocks and again

```go
controlFlowGraph := New()

for _, basicBlock := range basicBlocks {
  for _, successorBlock := range basicBlock.GetSuccessorBlocks() {
    controlFlowGraph.InsertEdge(
        &graph.Node{Value: basicBlock},
        &graph.Node{Value: successorBlock}
    )
  }
}
```

Listing 3.2: Iterating through basic-blocks and their successors, drawing and edge between them in the process of generating the control flow graph.

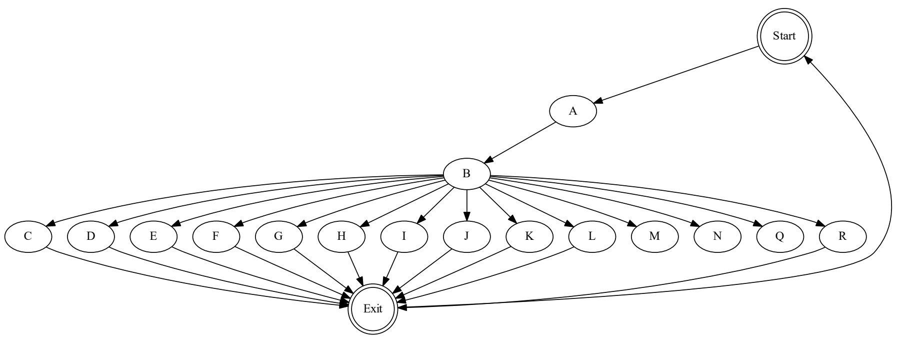


Figure 3.1: Control flow graph for function `monthNumberToString()` listed in 3.3.

iterates over all the successors to that basic-block, adding an edge between the block itself and the successor.

3.2.3 Measurements

After dividing the code into basic-blocks to build the intermediate representation of the program through the control flow graph, the third and final step consist of extracting information from the graph and feed it into the cyclomatic complexity equation mention in 2.1.

Representation of the graph is made through the custom graph package developed in this thesis, the package is provided with custom tailored methods to extract the number of vertices and edges, and to compute the set of strongly connected components in the graph.

The code snippet listed in 3.3 has a corresponding control flow graph as showed in figure 3.1, the number of nodes or vertices (V) equals 18, number edges (E) between the vertices equals 31 and the single set of strongly connected components equals:

$$\{R, Q, N, M, L, K, J, I, H, G, F, E, D, C\}$$

By inserting the values in the equation v(G) = e - n + p, function

```go
func monthNumberToString(month int) string {
  switch month {
  case 1:
    return "January"
  case 2:
    return "February"
  case 3:
    return "March"
  case 4:
    return "April"
  case 5:
    return "May"
  case 6:
    return "June"
  case 7:
    return "Juni"
  case 8:
    return "August"
  case 9:
    return "September"
  case 10:
    return "October"
  case 11:
    return "November"
  case 12:
    return "Desember"
  default:
    return "Invalid month"
  }
}
```

Listing 3.3: Converting month number to corresponding name through a `switch` statement

```go
package main

import "log"

// @SuppressRule("GOTO_USED")
func main() {
        counter := 0

        LOOP:
        if 100 > counter {
                counter++
                goto LOOP
        }
        log.Printf("Counter: %d\n", counter)
}
```

Listing 3.4: `GOTO_USED` rule suppressed.

`monthNumberToString()` yields the complexity of:

$$v(G) = 31 - 18 + 1 = 14$$

Exceeding the upper approved limit of 10. A CC measure between 11 and 20 is rated as medium in the effort and cost of testing the code. However, as specified in the background chapter on cyclomatic complexity, the upper limit is not a magical number, and it seems reasonable to allow larger switch statements in the same trivial niche of the example in Listing 3.3.

3.3 Suppressing rules

As specified in 2.2.1, linting is a heuristic method which is not perfect. There are occasions when the programmer is smarter than the analyzer. There might be valid reasons to bypass code rules defined in the tool, e.g. allowing unsynchronised printing from the `fmt` package or to allow higher cyclomatic complexity than the defined limit. Thus, some way of communicating with the tool, typically to ignore rules, is desirable.

The way this was solved was to embed the following annotation

```
@SuppressRule("RULE_NAME")
```

in function comments, if the annotations are well formed, the code in the function will not be checked against the rule specified, see program 3.4 for example usage of annotations to suppress rules. This way of communicating with the analyzer is both friendly toward readability as one easily can see which rules that are suppressed in each function by reading the code, in opposite to specify ignored rules in a dedicated XML or JSON configuration file.

3.4 Bug and Code Smell Checking

This section will go through the rules defined in the analyzer tool, describing how and why code constructs are recognized as rule violations. Each rule provides code examples of rule breaches and how the tool reacts on applying the tools on these breaches.

3.4.1 High Cyclomatic Complexity

Section 3.2 dealt with the detailed description of function and method computations of cyclomatic complexity (CC), while the background section 2.2.2 covers the motivation of measuring CC in programs.

The CC measure check in the tool flags function and method measures exceeding the value of 10. Violations higher than 10 indicates more complex code which may still be in some cases reasonable. However, testing is harder because of an exponentially increasing number of execution paths in the function or method.

The check enforces code smell rule `CYCLOMATIC_COMPLEXITY`, intentionally implementation of functions and methods with higher cyclomatic complexity than 10 can be silenced by suppressing the rule.

3.4.2 Races when referencing loop iterator variables in Goroutines

Goroutines in Go is like threads in other languages, ref section 2.1.4, used to process data in parallel. Because goroutines can be executed as closures, there are possibilities to reference variables in the outer scope of the closure function.

The program in Listing 3.5 shows a for-loop firing of closures in parallel by referencing the single variable `var` that takes on the value of each slice. The result will be non-deterministic as there is no guarantee for when a goroutine will start. The correct way of implementing the closure loop listed in Listing 3.5, is to pass the `var` variable as a copy argument to the closure.

The tool checks and identifies potential misuse of loop iterator variables only if the closure body references loop iterator variables, references to other variables in the outer scope is not flagged, even dough they might be a sign of unwanted races. Listing 3.5 shows the reference to the loop iterator from the goroutine closure.

3.4.3 Ignoring Errors

To be able to discover unexpected behavior or an abnormal state that a program may encounter, error handling and correction are crucial.

Errors in Go is represented by the built-in `error` interface, any type in Go can play the *role* of being a `error` by implementing the `Error() string` method.

```go
package main

import "log"

func main() {
        //Not-thread safe.
        for num := 0; num < 50; num++ {
                go func() {
                        log.Printf("Goroutine #%d\n", num)
                }()
        }
}
// $ ./GoAnalyzer -dir RaceIterator
// ------------------------------------------------
// Violations in RaceIterator :
//    0) RACE_CONDITION (Line 18) - Loop iterator
//       variables must be passed as argument to
//       Goroutine, not referenced.
// ------------------------------------------------
// Found total 1 violations!
// Took 3.0017ms
```

Listing 3.5: Race occurs on variable `num` as all 50 goroutines will reference the same variable.

Functions and methods that expect to encounter abnormal behavior return the `error` value as the last return result value. When a function returns a non-nil `error` value, it is an indication of that an abnormal behavior occurred in the function, the value itself contains a detailed description of the error. Ignoring `error` values returned by functions and methods is dangerous as one always take for granted that the program will behave as expected, It also greatly increases the difficulty of debugging as it is no clear sign of where the error occurred.

There is mainly two ways to detect ignorance of errors in Go code:

- By assigning the returning `error` value into a blank identifier _
- By not assigning the resulting `error` value from function and method calls into a variable.

It is natural to determine errors as handled if these cases are avoided, as the goal of this rule is to force assignment of returning errors into a variable, further usage of the variable will be ensured at compile time as the compiler requires all variables to be used.

Ignoring errors categorize as *code smell* as it is not a defect in the code, but a violation of fundamental design principles.

Listing 3.6 shows a code snippet where the operation of opening a file is checked for errors, if not any errors were encountered the programs tries to close the file, at this point the program overlook the possibilities of errors that might encounter when closing a file. At the bottom of the Listing, the tool detects and flags the ignorance of the error returned from `os.Close()`.

```go
func main() {
    if file, err := os.Open("me.go"); err != nil {
        defer file.Close() // Close() returns error.
    }
}
// $ ./GoAnalyzer -dir DeferReturnValue.go
// ------------------------------------------------
// Violations in DeferReturnValue.go :
//  0) ERROR_IGNORED (Line 3) - Never ignore errors,
//     ignoring them can lead to program crashes!
// ------------------------------------------------
// Found total 1 violations!
// Took 22.4845ms
```

Listing 3.6: `defer` calls function returning `error` value

3.4.4 `String()` method recursively calling itself

All formatted printing and string builder functions in Go's standard library uses the Stringer interface to allow custom data types in Go to define their native format for their value.

```go
    type Stringer interface {
      String() string
    }
```

The effect of implementing the interface for a type is that the String method will be called when a value of the type is passed as an operand to any functions accepting any format specifier representing a string, or by passing the operand to an unformatted print or string builder function.

The infinite recursion bug is introduced when one implements the `String()` interface for a type by using a print or string builder that will again call its String() method. Execution of that types `String()` method will lead to infinite recursive calls on itself causing stack overflow.

This algorithm is not shipped in the version of the tool implemented in this thesis, due to precision problems with the large proportion of false positives detection when applying the algorithm. Still, the presence of the bug in software is critical. An inclusion of this algorithm in the tool is addressed in the section about further work 5.1.

3.4.5 Printing from the `fmt` package

Rule `FMT_PRINTING` warns about references to print functions in the standard `fmt` package, print functions in this package are usually intended for debugging purposes and can remain in production code, and is therefore classified as a *code smell*. The print functions in the package are not synchronized and might clog the standard output (`stdout`.

By using the logger in the `log` package shipped in the standard library, one gains synchronized log printing among threads/goroutines and the flexibility of turn log printing on/of with a single switch, and redirect the

```
1  package main
2
3  import "log"
4
5  type Address struct {
6      Street, City string
7  }
8
9  func (address Address) String() string {
10     return fmt.Sprintf("%s", address)
11 }
12
13 func main() {
14     myAddress := Address{
15         Street: "Trimveien 6",
16         City: "Oslo"
17     }
18     log.Printf("My address: %s\n", myAddress)
19 }
20 // $ ./GoAnalyzer -dir RecursiveString
21 // ------------------------------------------------
22 // Violations in RecursiveString :
23 //     0) STRING_CALLS_ITSELF (Line 10) - Format
24 //        specifier for argument calls its own
25 //        String() method recursively, causing
26 //        runtime panic!
27 // ------------------------------------------------
28 // Found total 1 violations!
29 // Took 23.106ms
```

Listing 3.7: Format specifier for with `address` at line 10 causes recursive calls against itself.

```
1  package main
2
3  import (
4      "fmt"
5      "log"
6  )
7
8  // @SuppressRule("ERROR_IGNORED")
9  func main() {
10         fmt.Printf("Printing with fmt.Printf()")
11         log.Printf("Printing with logger\n")
12 }
13 // $ ./GoAnalyzer -dir FmtPrinting
14 // -----------------------------------------------
15 // Violations in FmtPrinting :
16 //    0) FMT_PRINTING (Line 10) - Printing from
17 //       the fmt package are not synchronized
18 //       and usually intended for debugging
19 //       purposes. Consider to use the log
20 //       package!
21 // -----------------------------------------------
22 // Found total 1 violations!
23 // Took 25.4083ms
```

Listing 3.8: Code smells when using `fmt` printing, use `log` instead!

logging stream to any desired destination as buffers or files. The following functions in the package should be avoided:

- `fmt.Print()`
- `fmt.Println()`
- `fmt.Printf()`

The program in Listing 3.8 shows the usage of `fmt.Printf()` which can be replaced with the log function on line 11, attached at the bottom is the output of applying the tool on the code.

3.4.6 Maps allocated with `new()`

Maps in Go is a built-in implementation of a traditional hash-table providing the common operations on a map like lookup, add and delete records.

Map types are references types, like a pointer pointing on the chunk of memory that constitute the map. A map variable is by default a `nil` map that needs to be assigned a chunk of initialized memory, the built-in allocation function `make()` allocates and initialized the hash table data structure and returns a map value pointing to the initialized chunk of memory holding the map.

A common error in Go is to use the other allocation primitive `new()` to allocate maps, new() serves a purpose different from `make()` in the form that it allocates zeroed memory areas, and does not prepare the memory

```
package main

import "log"

func main() {
        myMap := new(map[string]float64)
        log.Println(*myMap)
}
// $ ./GoAnalyzer -dir NewMap
// ------------------------------------------------
// Violations in NewMap :
//    0) MAP_ALLOCATED_WITH_NEW (Line 8) - Maps
//       must be initialized with make(), new()
//       allocated a nil map causing runtime panic
//       on write operations!
// ------------------------------------------------
// Found total 1 violations!
// Took 22.3688ms
```

Listing 3.9: Write operations against a map allocated with `new()` will cause runtime panic.

area for use like `make()`. Allocation of maps with `new()` will only allow read operations, write operations will cause runtime panic.

As presence of write operations against maps allocated with `new()` in the code will cause runtime panic, all maps allocated with `new()` is considered as a violation of the rule `MAP_ALLOCATED_WITH_NEW`, clearly categorized as bug, see Listing 3.9.

3.4.7 Statically evaluated conditions

Conditional statements like *if*, *else-if* and *for-loops* are applied to change the flow of control in programs dynamically, depending on the programs state. For conditional statements to serve the purpose of representing dynamic determination of programs flow under runtime, it is natural to have the impression that the Boolean condition in these statements cannot be evaluated statically.

Static conditional statements containing Boolean expressions is considered as meaningless as they do not serve their purpose, the conditional statement can be removed as one can determine the flow of the program before runtime. In addition to increase the readability by removing decision points that increases the cyclomatic complexity, a tiny fraction of performance increases is gained through saving of some few CPU instructions.

The tool will enforce the rule `CONDITION_EVALUATED_STATICALLY` by checking after Boolean conditions that always are true or false in the simplest sense of measuring nil comparison (always false) and by detecting the TRUE or FALSE operand as the Boolean expression. Statically evaluated Boolean conditions are considered as code smell. Listing 3.10 shows an example of the tool detecting static conditions.

```go
package main

import (
    "fmt"
    "log"
)

// @SuppressRule("ERROR_IGNORED")
func main() {
        fmt.Printf("Printing with fmt.Printf()")
        log.Printf("Printing with logger\n")
}
// $ ./GoAnalyzer -dir FmtPrinting
// ------------------------------------------------
// Violations in FmtPrinting :
//    0) FMT_PRINTING (Line 10) - Printing from
//       the fmt package are not synchronized
//       and usually intended for debugging
//       purposes. Consider to use the log
//       package!
// ------------------------------------------------
// Found total 1 violations!
// Took 25.4083ms
```

Listing 3.10: If condition can be evaluated at compile time.

3.4.8 Return statement kills code

Dead code is code that can be executed because there do not exist any execution path to the code, the code is unreachable. Dead code is commonly determined as meaningless and disruptive elements in the source code.

The tool enforces the rule DEAD_CODE by detecting code smells in functions and methods where a return statement is declared above other statements in the same scope, as the code in the scope is executed sequential and a statement always will return and terminate the function or method, the code below the return statement will never be executed, and is therefore considered as dead. Listing 3.11 shows how the return statement kills the logging statement at line 9.

3.4.9 Usage of GOTO statements

Dijkstra jumps into the endless debate about the effect of GOTO statements in his article [5] from 1968. He clearly takes distance from the use of GOTO statements by stating *"... I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce"*. Dijkstra justifies his statement by describing the go to statement as to primitive that invites to make a mess of one's programs.

Dijkstra's view of GOTO statements is commonly shared among computer scientist today to some degree, the tool therefore checks after branching statements of type (break, continue, goto, fallthrough)) with a subsequent label. Occurrences are flagged as violations of the code smell

```go
package main

import "log"

func main() {
	log.Println("Hello World")
	pi := 3.14
	return
	log.Printf("Finally, Pi = %d\n", pi)
}
// $ ./GoAnalyzer -dir ReturnKillsCode
// ------------------------------------------------
// Violations in ReturnKillsCode :
//     0) RETURN_KILLS_CODE (Line 8) - Code is dead
//        because of return! There is
//        no possible execution path to the code
//        below in this scope!
// ------------------------------------------------
// Found total 1 violations!
// Took 36.2019ms
```

Listing 3.11: There is no execution path to the last logging statement because of the `return` statement.

```go
package main

import "log"

func main() {
	counter := 0
	LOOP:
	if 100 > counter {
		counter++
		goto LOOP
	}
	log.Printf("Counter: %d\n", counter)
}
```

rule `GOTO_USED`. All four statements is a form of `GOTO` statement although they differ a bit in behavior, still the main behavior is transform of control to another line of code.

The program listed in 3.4.9 shows a trivial Go program constructing a loop by a GOTO statement and the output of running the tool on the code.

3.4.10 Empty `for` and `if-else` bodies

The rule of enforcing no presence of empty if, else-if and for-loop bodies is a code smell in the same category of unreachable code. The purpose of implementing decision points is to change the flow of control in programs, giving the possibility to change the behavior of the program on behave of Boolean conditions. Therefore empty presence of conditional statements of bodies are totally wasted as they do not have any function in the code, see

```go
package main

import "math/rand"

func main() {
        if 50 == rand.Intn(100) {} else {}

        for i := 0; i < 10; i++ {}
}
// $ ./GoAnalyzer -dir EmptyIfElseFor
// ------------------------------------------------
// Violations in EmptyIfElseFor :
//  0) EMPTY_IF_BODY (Line 8) - If body is empty,
//     wasteful to not do anything with the if
//     condition.
//  1) EMPTY_ELSE_BODY (Line 10) - ELse body is
//     empty, wasteful to not do anything with
//     the else condition.
//  2) EMPTY_FOR_BODY (Line 14) - For body is
//     empty, wasteful to not do anything with the
//     for condition.
// ------------------------------------------------
// Found total 3 violations!
// Took 28.3488ms
```

Listing 3.12: Empty `if-else` and `for` bodies.

Listing 3.12.

3.5 SonarQube Plugin

SonarQube envisages that all analysis work should be done inside the plugin, which is written in Java. This requires a Java implementation to cover the entire language by writing the grammar, lexer, parser, abstract syntax tree (AST) and AST visitor methods to measure metrics and enforcing coding rules.

To bypass the work of covering the Go language in Java a shortcut has been selected by adapting the SonarQube plugin to call the GoAnalyzer executable file with the `-json` switch on, the tool will then return the analysis result as JSON which is unmarshalled into Java objects representing the analysis results. Still, with this adequate solution, the SonarQube plugin is packaged into a single JAR file with no external dependencies, and therefore shipped and run as if the plugin entirely was developed in Java.

The limitations of not natively implement the analyzer in Java, and instead call the "external" executable GoAnalyzer tool, is the limitation of platform support. As Java program almost runs on all platforms through their Java Virtual Machine, Go is compiled down to platform dependent byte-code. Which means that the SonarQube plugin is restricted to run on platforms that correspond with the GoAnalyzer executable files bundled with the JAR file. The version of the SonarQube plugin developed in this thesis can be executed on the operating systems and architectures listed in the portability section (1.2.4).

Chapter 4

Case Studies

Two case studies have been run. The purpose of applying the tool on the source code in real software projects is to see how the tool deals with source code repositories of different size, and the inspection accuracy.

The source code of *Go compiler* and the *GoAnalyzer* are the programs we will apply the tool on in the search of bugs and code smells. All the analysis is performed using the SonarQube plugin which executes the `GoAnalyzer` tool and sends the result to the SonarQube server for result aggregation and presentation.

Also, a performance technical analysis of the tool is done for each of the two use cases, to see how well the GoAnalyzer tool performs on memory and time consumption.

4.1 Obtaining performance measurements of the tool

Multiple runs of analysis are required in order to get statically representable data. During these runs, data about time and memory usage are gathered. The timing of the analysis is performed by the GoAnalyzer tool itself internally, and printed in the final result, this way we avoid of doing an external timing of the analyzer who will be inaccurate as we also would measure the time taken to print the analysis result. Memory is measured in the max amount of memory used through the hole analysis; there is no point in measuring memory usage over time as the GoAnalyzer is fast.

To eliminate operating system interference with the analysis performance, i.e. reducing CPU clock speed and voltage to reduce power consumption. The task of executing the analysis multiple times and collect the result and performance data from each run is automated through the bash script in Listing 4.1.

All the analysis that lay the foundation for the results presented in this chapter is performed on a workstation provided by the University of Oslo. Table 4.1 lists the detailed hardware and software specifications for the computer:

HP Compaq Elite 8100	
Operating System	Red Hat Enterprise Linux Workstation 7.2 (Maipo)
CPU	Intel Core i7 CPU 870 2.93GHz
RAM	2 x 4 GB DDR3 1333MHz
Go	go1.6.3 linux/amd64

Table 4.1: Software and hardware specification of the computer executing the analysis.

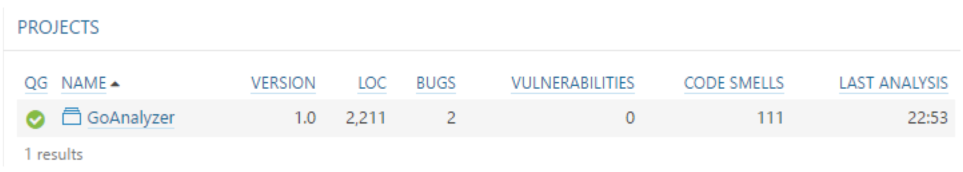
```bash
#!/usr/bin/env bash
if [ "$#" -ne 1 ]; then
  echo "Error: Missing directory as argument"
  exit 1
fi

for i in {1..11};
do
  /usr/bin/time -f 'Memory used: %M kbytes' --append -o
      GoAnalyzerRun$i.txt ./GoAnalyzer -dir $1 > GoAnalyzerRun$i.
      txt
  echo Run nr $i. Result dumped in GoAnalyzerRun$i.txt
done
```

Listing 4.1: Bash script to execute the GoAnalyzer multiple times and dump the result to files, memory usage is also attached.

4.2 GoAnalyzer

Using the tool to inspect its own source code is like take its own medicine. By rapidly applying the tool on the code through the development phase, the number of code bugs is absent, and the number of code smells is minimal. Figure 4.1 shows the overview result of the analysis. As seen in the analysis result printed in Appendix A, there are only files under *testcode* directories that are containing bugs. These files are part of unit tests implemented in the tool and the bugs in the files are intentionally added. Also, the majority of detected code smells resides in the files used in unit testing of the tool. Only the non-intentionally rule violations discovered in the source code of the GoAnalyzer are the code smells complaining about errors ignored in package `graph` and `cfgraph` when not checking the potential error that can happen when writing to the buffer. The last non-intentionally complaint is the high measure of cyclomatic complexity in package `bblock`, investigated deeper in next section.


PROJECTS

QG	NAME ▴	VERSION	LOC	BUGS	VULNERABILITIES	CODE SMELLS	LAST ANALYSIS
✓	GoAnalyzer	1.0	2,211	2	0	111	22:53

1 results

Figure 4.1: SonarQube overview of analysis.

4.2.1 A High Measure Of Cyclomatic Complexity

When the tool inspects the code in Appendix B, method `Visit()` measures to cyclomatic complexity value 47, which is way beyond the limit 10 and an upper reasonable value 15 described by McCabe. The rule encourages to split up the method into smaller function and methods to reduce the complexity to under the value of 10. In a personal view, the method should be annotated

```
@SuppressRule("CYCLOMATIC_COMPLEXITY")
```

to acknowledge that the method resides intentionally. The intention in the method is to implement the `Visitor` interface of package `ast` in the standard library which allows one to visit each node in the abstract syntax tree of a file. The method will through its traversal of the tree identify branching nodes and split the statements in the file into basic-blocks. With the help of the state machine represented in the method, relationships between basic-blocks are identified. The usage of a big type switch as in the method is typically in Go for type assertion.

4.3 Go compiler

The Go compiler [2] is, of course, an important piece of software in the world of Go, as it is the official compiler for the Go programming language. Historically the compiler was first written in C, since version 1.5 the compiler and its toolchain was reimplemented in Go. The benefits of getting rid of C by bootstrapping the compiler is that the toolchain will be unified and provide a heterogeneous codebase for all platforms, making it easier to introduce support for new platforms.

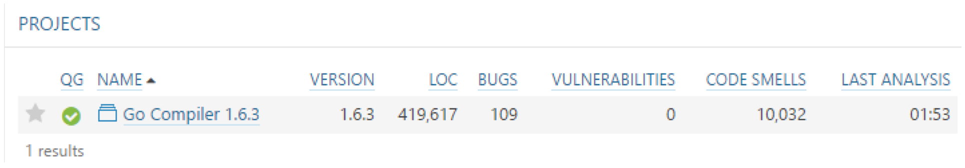
This use case analysis and compares source code of the Go compiler version 1.6.0, 1.6.1, 1.6.2 and 1.6.3, constituting all releases of version 1.6. The compiler consists of around 93% Go code, 6% Assembler code, and 0,5% C code.

4.3.1 Preparing The Source Code For Analysis

As the analyzer performs inspection using the abstract syntax tree, it is a requirement that all files constituting a package throughout the source code can be parsed to perform an inspection on the package. Since the Go compiler is developed with the usage of unit tests, there are occurrences of `.go` files in the repository being a part of unit tests that is not syntactically valid, i.e. not possible to parse. Causing the GoAnalyzer to panic with a parsing error message.

By inspecting the source files flagged by the tool to as syntactically invalid, it was identified that these records were either part of unit tests intentionally representing faulty syntax. To perform the analysis of the Go compiler, these files who did not parse was deleted from the repository.

- go\src\cmd\go\testdata\src\badpkg\x.go



PROJECTS

QG	NAME ▲	VERSION	LOC	BUGS	VULNERABILITIES	CODE SMELLS	LAST ANALYSIS
✓	Go Compiler 1.6.3	1.6.3	419,617	109	0	10,032	01:53

1 results

Figure 4.2: SonarQube overview of the Go compiler analysis.

- go\src\cmd\go\testdata\src\badtest\badsyntax\x_test.go
- go\src\cmd\go\testdata\src\notest\hello.go
- go\src\cmd\go\testdata\src\syntaxerror\x_test.go

4.3.2 Results

Table 4.2 show the summary of the analysis of Go compiler version 1.6.3. The number of rule violations is too large to inspect the correctness of each violation manually. Still, the high amount of races on loop iterator variables is interesting, random investigation of some of these races violations shows both that the tool catches real potential races against loop variables and that the tool reports false positives i.e. flags goroutines which receive the loop variable value through an argument instead of directly references. This situation is confirmed after implementing unit tests replication the behaviour of these false positive detections found in the Go compiler, the issue of false positives is not resolved in the version of the tool implemented in this thesis.

Besides the discovery of the tool being unprecise for race detection, the results of analysis the Go compilers is not surprising. The high amount of cyclomatic complexity measure violations of 10 is not surprising as the usage of switch statements are high.

The amount of violations of printing from the `fmt` package and errors ignored interacts. As all printing functions in the `fmt` package may fail returning a `error`, a tendency through the source code of the compiler being analysed and all the public Go source code I have read shows a trend of ignoring the potential error that may happen when calling a print function in the `fmt` package. Roughly one could say that the number of printing from `fmt` package violations cause the corresponding amount of `error` ignored violations.

Still, the amount of `error` ignored is significant, the reason can be explained with the tendency or habit among developers not to check the returning `error` value from functions that they experience do not fail, for instance, the habit of not checking errors that may occur when writing to a buffer.

Rule	Number of violations
Map allocated with `new()`	4
High cyclomatic complexity	2033
Race on loop variable	105
Error ignored	5189
Printing from `fmt` package	1334
Static condition	35
`return` kills code	60
`GOTO` statements used	1246
Empty `if` body	71
Empty `for` loop body	72

Table 4.2: Rules violated in the Go compiler v1.6.3.

Chapter 5

Conclusion and Further Work

In this thesis, the first version of a tool for a rule driven static analysis of Go source code has been described and implemented. The tool detects various types of bugs and code smells when inspecting the code, helping to identify potential problems in the code at an early stage in the development phase, and to measure the source code quality in projects e.g. with the aid of the SonarQube plugin provided.

Through the case studies, the tool is proved to be effective in the demand of running time and memory consumption, even on large code bases with thousands of files. On the other hand, in the case study of the Go compiler, the precision and correctness goal is not satisfied as numerous false positives rule violations were flagged. Concerning the algorithm detecting potential race conditions on loop variables. These false positive situations where not covered in the unit test trying to ensuring the goal of correctness in the tool. Further work on the tool should address this defect.

If one disregards the temporary issue with unprecise detection rule in the version of tool implemented in this thesis, the value in the tool lays in the active usage of the tool on one's code from a project's lifetime. In this way, one can follow up results from analysis and address rule violations either by solving the problem describes by the rule, or by intentionally suppressing rules.

Extensibility is the third goal in the thesis; the tool has been developed with extensibility in mind, facilitate opportunities for extending the rule base with new algorithms either by using the abstract syntax tree or by adding new packages, or utilizing other packages already implemented in the tool, without major modification to existing code.

5.1 Future Work

Some of the unexplored areas in this thesis are other fields of static analysis e.g. escape analysis to determine concurrency issues as deadlocks and races. The fields may be relevant to discover to implement more sophisticated algorithms for static analysis.

5.1.1 Resolve discovered precision issues

Unfortunately precision issues with two of the algorithms described in the implementation phase of the tool exist in the version of the tool delivered in this thesis. The algorithm detecting `String()` method recursively calls itself closer described in Section 3.4.4 is completely excluded from the tool. The racer algorithm detecting races against the loop variable when executing goroutines inside a for-loop detects successfully correct occurrences of potential races, but also a high rate of false positives. Both algorithms are considered important and should be correctly implemented.

5.1.2 Detailed specification of rule suppression and activation

In the version of the tool implemented in this thesis, specification of rules to be suppressed during an analysis is rather primitive. It is only possible to suppress a rule at function and method level, in some situations, it is not desirable to disable a whole function or method against code inspection against a rule. As a future change of the function or method may introduce undesirable violations not discovered due to the global rule suppression for the function or method.

To solve this situation, one could consider introducing fine-grained rule suppression at statement or expression level. When the code base grows, it is also more convenient to specify which rules that are activated under analysis, instead of disabling unwanted violations as they appear. Achieved by specifying the rules to be active during analysis in a structured file at package level for which the rules applies.

5.1.3 Specification of measure thresholds

As new rules enforcing measure thresholds is added to the ruleset, it may be useful to have a mechanism for fine-grained specification of measure thresholds. Realized in the same fashion as rule specification and activation.

Appendices

Appendix A

Analysis result of `GoAnalyzer`

```
PACKAGE: main (analyzer/linter/testcode/threadlooping)
 Violations in main.go :
  0) RACE_CONDITION (Line 18) - Loop iterator variables must be
      passed as argument to Goroutine, not referenced..
---------------------------------------------------------------------------
PACKAGE: main (analyzer/linter/ccomplexity/bblock/testcode)
 Violations in _nestedswitches.go :
  0) CYCLOMATIC_COMPLEXITY (Line 8) - Cyclomatic complexity in
      main() is 17, upper limit is 10..
 Violations in _returnswitcher.go :
  0) CYCLOMATIC_COMPLEXITY (Line 14) - Cyclomatic complexity in
      monthNumberToString() is 13, upper limit is 10..
 Violations in _select.go :
  0) FMT_PRINTING (Line 22) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
  1) FMT_PRINTING (Line 28) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
  2) FMT_PRINTING (Line 30) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
 Violations in _simpleswitch.go :
  0) FMT_PRINTING (Line 15) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
  1) FMT_PRINTING (Line 17) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
  2) FMT_PRINTING (Line 19) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
  3) FMT_PRINTING (Line 21) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
 Violations in _switch.go :
  0) FMT_PRINTING (Line 15) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
  1) FMT_PRINTING (Line 17) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
```

```
       . Consider to use the log package!.
  2) FMT_PRINTING (Line 18) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  3) FMT_PRINTING (Line 20) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  4) FMT_PRINTING (Line 22) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  5) FMT_PRINTING (Line 24) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  6) FMT_PRINTING (Line 27) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
Violations in _gcd.go :
  0) FMT_PRINTING (Line 10) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  1) FMT_PRINTING (Line 11) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
Violations in _nestedswitches.go :
  0) FMT_PRINTING (Line 16) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  1) FMT_PRINTING (Line 18) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  2) FMT_PRINTING (Line 22) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  3) FMT_PRINTING (Line 24) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  4) FMT_PRINTING (Line 26) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  5) FMT_PRINTING (Line 29) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  6) FMT_PRINTING (Line 37) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  7) FMT_PRINTING (Line 39) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  8) FMT_PRINTING (Line 41) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  9) FMT_PRINTING (Line 43) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  10) FMT_PRINTING (Line 47) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  11) FMT_PRINTING (Line 49) - Printing from the fmt package are
       not synchronized and usually intended for debugging purposes
```

```
     . Consider to use the log package!.
Violations in _looper.go :
 0) FMT_PRINTING (Line 15) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
Violations in _returnswitcher.go :
 0) FMT_PRINTING (Line 11) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
Violations in _simple.go :
 0) FMT_PRINTING (Line 10) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
Violations in _simplelooperswitch.go :
 0) FMT_PRINTING (Line 15) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
 1) FMT_PRINTING (Line 17) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
Violations in _typeswitch.go :
 0) FMT_PRINTING (Line 27) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
Violations in _fallthrough.go :
 0) FMT_PRINTING (Line 13) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
 1) FMT_PRINTING (Line 15) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
 2) FMT_PRINTING (Line 18) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
 3) FMT_PRINTING (Line 21) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
 4) FMT_PRINTING (Line 23) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
 5) FMT_PRINTING (Line 25) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
 6) FMT_PRINTING (Line 28) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
 7) FMT_PRINTING (Line 31) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
Violations in _nestedifelse.go :
 0) FMT_PRINTING (Line 17) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
 1) FMT_PRINTING (Line 20) - Printing from the fmt package are
     not synchronized and usually intended for debugging purposes
     . Consider to use the log package!.
 2) CONDITION_EVALUATED_STATICALLY (Line 14) - Condition will
     always be true.
```

```
  3) CONDITION_EVALUATED_STATICALLY (Line 18) - Condition will
      always be false.
----------------------------------------------------------------------------
PACKAGE: main (analyzer/linter/ccomplexity/cfgraph/testcode)
 Violations in _simple.go :
  0) FMT_PRINTING (Line 10) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
 Violations in _switcher.go :
  0) FMT_PRINTING (Line 10) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
 Violations in _gcd.go :
  0) FMT_PRINTING (Line 19) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
  1) FMT_PRINTING (Line 20) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
 Violations in _looper.go :
  0) FMT_PRINTING (Line 15) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
----------------------------------------------------------------------------
PACKAGE: main (analyzer/linter/testcode/fmtprinting)
 Violations in main.go :
  0) FMT_PRINTING (Line 11) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
  1) FMT_PRINTING (Line 12) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
  2) FMT_PRINTING (Line 13) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
----------------------------------------------------------------------------
PACKAGE: main (analyzer/)
 Violations in GoAnalyzer.go :
  0) ERROR_IGNORED (Line 111) - Never ignore erros, ignoring them
      can lead to program crashes.
----------------------------------------------------------------------------
PACKAGE: linter (analyzer/linter)
 Violations in linter.go :
  0) ERROR_IGNORED (Line 174) - Never ignore erros, ignoring them
      can lead to program crashes.
  1) ERROR_IGNORED (Line 175) - Never ignore erros, ignoring them
      can lead to program crashes.
----------------------------------------------------------------------------
PACKAGE: linter_test (analyzer/linter)
 Violations in linter_test.go :
  0) ERROR_IGNORED (Line 62) - Never ignore erros, ignoring them
      can lead to program crashes.
----------------------------------------------------------------------------
```

```
PACKAGE: stack_test (analyzer/linter/ccomplexity/graph/stack)
 Violations in stacker_test.go :
  0) ERROR_IGNORED (Line 18) - Never ignore erros, ignoring them
      can lead to program crashes.
---------------------------------------------------------------------------

PACKAGE: main (analyzer/linter/testcode/emptyifbody)
 Violations in main.go :
  0) EMPTY_IF_BODY (Line 11) - If body is empty, wasteful to not
      do anything with the if condition..
---------------------------------------------------------------------------

PACKAGE: graph (analyzer/linter/ccomplexity/graph)
 Violations in graph.go :
  0) ERROR_IGNORED (Line 248) - Never ignore erros, ignoring them
      can lead to program crashes.
  1) ERROR_IGNORED (Line 249) - Never ignore erros, ignoring them
      can lead to program crashes.
  2) ERROR_IGNORED (Line 250) - Never ignore erros, ignoring them
      can lead to program crashes.
  3) ERROR_IGNORED (Line 251) - Never ignore erros, ignoring them
      can lead to program crashes.
  4) ERROR_IGNORED (Line 252) - Never ignore erros, ignoring them
      can lead to program crashes.
  5) ERROR_IGNORED (Line 253) - Never ignore erros, ignoring them
      can lead to program crashes.
---------------------------------------------------------------------------

PACKAGE: main (analyzer/linter/testcode/emptyforbody)
 Violations in main.go :
  0) EMPTY_FOR_BODY (Line 8) - For body is empty, wasteful to not
      do anything with the for condition..
---------------------------------------------------------------------------

PACKAGE: main (analyzer/linter/testcode/goto)
 Violations in main.go :
  0) GOTO_USED (Line 12) - Please dont use GOTO statements, they
      lead to spagehetti code!.
---------------------------------------------------------------------------

PACKAGE: main (analyzer/linter/testcode/staticconditions)
 Violations in main.go :
  0) CONDITION_EVALUATED_STATICALLY (Line 14) - Condition will
      always be true.
  1) CONDITION_EVALUATED_STATICALLY (Line 18) - Condition will
      always be false.
  2) CONDITION_EVALUATED_STATICALLY (Line 22) - Both left and
      right operand is basic literal that can be eveualted at
      compile time.
  3) CONDITION_EVALUATED_STATICALLY (Line 26) - Both left and
      right operand is basic literal that can be eveualted at
      compile time.
  4) CONDITION_EVALUATED_STATICALLY (Line 38) - Comparison of
      function Bytes is always false.
  5) CONDITION_EVALUATED_STATICALLY (Line 41) - Condition will
      always be true.
---------------------------------------------------------------------------

PACKAGE: cfgraph (analyzer/linter/ccomplexity/cfgraph)
```

```
 Violations in controlflowgraph.go :
  0) ERROR_IGNORED (Line 31) - Never ignore erros, ignoring them
      can lead to program crashes.
  1) ERROR_IGNORED (Line 36) - Never ignore erros, ignoring them
      can lead to program crashes.
  2) ERROR_IGNORED (Line 37) - Never ignore erros, ignoring them
      can lead to program crashes.
  3) ERROR_IGNORED (Line 38) - Never ignore erros, ignoring them
      can lead to program crashes.
  4) ERROR_IGNORED (Line 39) - Never ignore erros, ignoring them
      can lead to program crashes.
  5) ERROR_IGNORED (Line 40) - Never ignore erros, ignoring them
      can lead to program crashes.
  6) ERROR_IGNORED (Line 41) - Never ignore erros, ignoring them
      can lead to program crashes.
  7) ERROR_IGNORED (Line 44) - Never ignore erros, ignoring them
      can lead to program crashes.
  8) ERROR_IGNORED (Line 45) - Never ignore erros, ignoring them
      can lead to program crashes.
  9) ERROR_IGNORED (Line 46) - Never ignore erros, ignoring them
      can lead to program crashes.
  10) ERROR_IGNORED (Line 47) - Never ignore erros, ignoring them
      can lead to program crashes.
  11) ERROR_IGNORED (Line 50) - Never ignore erros, ignoring them
      can lead to program crashes.
  12) ERROR_IGNORED (Line 53) - Never ignore erros, ignoring them
      can lead to program crashes.
---------------------------------------------------------------------------

PACKAGE: main (analyzer/linter/testcode/bufferwriting)
 Violations in main.go :
  0) ERROR_IGNORED (Line 14) - Never ignore erros, ignoring them
      can lead to program crashes.
  1) ERROR_IGNORED (Line 19) - Never ignore erros, ignoring them
      can lead to program crashes.
---------------------------------------------------------------------------

PACKAGE: main (analyzer/linter/testcode/cyclomaticcomplexity)
 Violations in main.go :
  0) CYCLOMATIC_COMPLEXITY (Line 13) - Cyclomatic complexity in
      monthNumberToString() is 14, upper limit is 10..
 Violations in main.go :
  0) FMT_PRINTING (Line 10) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
      . Consider to use the log package!.
  1) ERROR_IGNORED (Line 10) - Never ignore erros, ignoring them
      can lead to program crashes.
---------------------------------------------------------------------------

PACKAGE: main (analyzer/linter/testcode/errorignored)
 Violations in main.go :
  0) ERROR_IGNORED (Line 16) - Never ignore erros, ignoring them
      can lead to program crashes.
  1) ERROR_IGNORED (Line 17) - Never ignore erros, ignoring them
      can lead to program crashes.
  2) ERROR_IGNORED (Line 22) - Never ignore erros, ignoring them
      can lead to program crashes.
  3) ERROR_IGNORED (Line 27) - Never ignore erros, ignoring them
      can lead to program crashes.
  4) ERROR_IGNORED (Line 35) - Never ignore erros, ignoring them
      can lead to program crashes.
  5) ERROR_IGNORED (Line 51) - Never ignore erros, ignoring them
      can lead to program crashes.
---------------------------------------------------------------------------

PACKAGE: main (analyzer/linter/testcode/labeledbranch)
 Violations in main.go :
  0) GOTO_USED (Line 15) - Please dont use GOTO statements, they
      lead to spagehetti code!.
---------------------------------------------------------------------------

PACKAGE: bblock (analyzer/linter/ccomplexity/bblock)
 Violations in basicblock.go :
  0) CYCLOMATIC_COMPLEXITY (Line 232) - Cyclomatic complexity in
      Visit() is 47, upper limit is 10..
---------------------------------------------------------------------------

PACKAGE: main (analyzer/linter/testcode/emptyelsebody)
 Violations in main.go :
  0) EMPTY_ELSE_BODY (Line 16) - ELse body is empty, wasteful to
      not do anything with the else condition..
---------------------------------------------------------------------------

PACKAGE: main (analyzer/linter/testcode/newmap)
 Violations in main.go :
  0) MAP_ALLOCATED_WITH_NEW (Line 11) - Maps must be initialized
      with make(), new() allocates a nil map causing runtime panic
       on write operations!.
---------------------------------------------------------------------------

PACKAGE: main (analyzer/linter/ccomplexity/testcode)
 Violations in _switcher.go :
  0) CYCLOMATIC_COMPLEXITY (Line 13) - Cyclomatic complexity in
      monthNumberToString() is 14, upper limit is 10..
 Violations in _gcd.go :
  0) FMT_PRINTING (Line 19) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  1) FMT_PRINTING (Line 20) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
 Violations in _helloworld.go :
  0) FMT_PRINTING (Line 10) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
 Violations in _swap.go :
  0) FMT_PRINTING (Line 12) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
  1) FMT_PRINTING (Line 14) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
 Violations in _switcher.go :
  0) FMT_PRINTING (Line 10) - Printing from the fmt package are
      not synchronized and usually intended for debugging purposes
       . Consider to use the log package!.
---------------------------------------------------------------------------
```

```
PACKAGE: main (analyzer/linter/testcode/earlyreturn)
 Violations in main.go :
  0) RETURN_KILLS_CODE (Line 13) - Code is dead because of return!
       There is no possible execution path to the code below in
      this scope!.
--------------------------------------------------------------------------

## ANALYSIS SUMMARY ##
Total 113 violations found!
Total number of Go files: 56
Total lines of code (LOC): 2356
Total lines of comments: 334
Total time used: 334.462461ms
For rule details: https://github.com/chrisbbe/GoAnalysis/wiki
```

Appendix B

Package `bblock` source code

```go
// Copyright (c) 2015-2016 The GoAnalysis Authors. All rights
    reserved.
// Use of this source code is governed by a BSD-style license that
     can
// be found in the LICENSE file.
package bblock

import (
 "fmt"
 "go/ast"
 "go/parser"
 "go/token"
 "log"
 "sort"
)

type BasicBlockType int

//Basic Block types.
const (
 FUNCTION_ENTRY BasicBlockType = iota
 IF_CONDITION
 ELSE_CONDITION
 SWITCH_STATEMENT
 CASE_CLAUSE
 SELECT_STATEMENT
 COMM_CLAUSE
 RETURN_STMT
 FOR_STATEMENT
 RANGE_STATEMENT
 GO_STATEMENT
 CALL_EXPRESSION
 ELSE_BODY
 FOR_BODY
 EMPTY
 START
 EXIT
 UNKNOWN
)

var basicBlockTypeStrings = [...]string{
 FUNCTION_ENTRY:  "FUNCTION_ENTRY",
 IF_CONDITION:   "IF_CONDITION",
 ELSE_CONDITION:  "ELSE_CONDITION",
 SWITCH_STATEMENT: "SWITCH_STATEMENT",
 CASE_CLAUSE:   "CASE_CLAUSE",
 SELECT_STATEMENT: "SELECT_STATEMENT",
 COMM_CLAUSE:   "COMM_CLAUSE",
 RETURN_STMT:   "RETURN_STMT",
 FOR_STATEMENT:  "FOR_STATEMENT",
 RANGE_STATEMENT: "RANGE_STATEMENT",
 GO_STATEMENT:   "GO_STATEMENT",
 CALL_EXPRESSION: "CALL_EXPRESSION",
 ELSE_BODY:    "ELSE_BODY",
 FOR_BODY:     "FOR_BODY",
 EMPTY:      "EMPTY",
 START:      "Start",
 EXIT:       "Exit",
 UNKNOWN:     "UNKNOWN",
}

func (bbType BasicBlockType) String() string {
 return basicBlockTypeStrings[bbType]
}

func (basicBlock *BasicBlock) UID() string {
 //Both START and EXIT blocks are meta-blocks, giving them
     negative UID which will not be confused with 'real' blocks.
 if basicBlock.Type == START || basicBlock.Type == EXIT {
  return fmt.Sprintf("%d", 0-basicBlock.Type)
 }
 return fmt.Sprintf("%d", basicBlock.EndLine)
}

func (basicBlock *BasicBlock) String() string {
 if basicBlock.Type == START {
  return basicBlock.Type.String()
 } else if basicBlock.Type == EXIT {
  return basicBlock.Type.String()
 }
 return fmt.Sprintf("BLOCK NR.%d (%s) (EndLine: %d)", basicBlock.
     Number, basicBlock.Type.String(), basicBlock.EndLine)
}

func (basicBlock *BasicBlock) AddSuccessorBlock(successorBlocks
    ...*BasicBlock) {
 for _, successorBlock := range successorBlocks {
  basicBlock.successor[successorBlock.EndLine] = successorBlock //
      Update or add.
 }
}

func NewBasicBlock(blockNumber int, blockType BasicBlockType,
    endLine int) *BasicBlock {
 return &BasicBlock{Number: blockNumber, Type: blockType, EndLine:
      endLine, successor: map[int]*BasicBlock{}}
}

func (basicBlock *BasicBlock) GetSuccessorBlocks() []*BasicBlock {
 keys := make([]int, len(basicBlock.successor))
 basicBlocks := []*BasicBlock{}

 i := 0
 for k := range basicBlock.successor {
  keys[i] = k
  i++
 }
 sort.Ints(keys) //Sort keys from map.

 //Add the basic-block into the array.
 for _, key := range keys {
  basicBlocks = append(basicBlocks, basicBlock.successor[key])
 }
 return basicBlocks
}

type BasicBlock struct {
 Number      int
 Type       BasicBlockType
 EndLine     int
 successor    map[int]*BasicBlock
 FunctionName   string
 FunctionDeclLine int
}

type visitor struct {
 basicBlocks  map[int]*BasicBlock
 sourceFileSet *token.FileSet

 lastBlock   *BasicBlock
 prevLastBlock *BasicBlock

 returnBlock *BasicBlock
 forBlock   *BasicBlock
 forBodyBlock *BasicBlock
 switchBlock *BasicBlock
}

// UpdateBasicBlock updates all the variables from the
    newBasicBlock into the basicBlock object.
func (basicBlock *BasicBlock) UpdateBasicBlock(newBasicBlock *
    BasicBlock) {
 if newBasicBlock != nil {
  basicBlock.Number = newBasicBlock.Number
  basicBlock.Type = newBasicBlock.Type
  basicBlock.EndLine = newBasicBlock.EndLine
  basicBlock.successor = newBasicBlock.successor
  basicBlock.FunctionName = newBasicBlock.FunctionName
  basicBlock.FunctionDeclLine = newBasicBlock.FunctionDeclLine
 }
}

func (v *visitor) AddBasicBlock(blockType BasicBlockType, position
     token.Pos) *BasicBlock {
 line := v.sourceFileSet.File(position).Line(position)
 basicBlock := NewBasicBlock(-1, blockType, line) //-1 indicates
     number will be set later.

 // Bookkeeping.
 v.prevLastBlock = v.lastBlock
```

```go
	v.lastBlock = basicBlock

	//Update the existing block., or add new block.
	if bb, ok := v.basicBlocks[line]; ok {
		bb.UpdateBasicBlock(basicBlock)
		v.lastBlock = bb
		return bb
	} else {
		v.basicBlocks[line] = basicBlock
	}
	return basicBlock
}

// GetBasicBlocks converts map holding the basic-blocks to the
    ordered set
// of basic-blocks, in right order!
func (v *visitor) GetBasicBlocks() []*BasicBlock {
	keys := make([]int, len(v.basicBlocks))
	basicBlocks := make([]*BasicBlock, len(v.basicBlocks))

	i := 0
	for k := range v.basicBlocks {
		keys[i] = k
		i++
	}
	sort.Ints(keys) //Sort keys from map.

	//Add the basic-block into the array.
	for index, key := range keys {
		basicBlocks[index] = v.basicBlocks[key]
		basicBlocks[index].Number = index //Set basic-block number.
	}
	return basicBlocks
}

func GetBasicBlocksFromSourceCode(filePath string, srcFile []byte)
     ([]*BasicBlock, error) {
	fileSet := token.NewFileSet()
	file, err := parser.ParseFile(fileSet, filePath, srcFile, parser.
	    ParseComments|parser.AllErrors)
	if err != nil {
		return nil, fmt.Errorf("Parse error: %s", err)
	}

	visitor := &visitor{sourceFileSet: fileSet, basicBlocks: make(map
	    [int]*BasicBlock)}
	ast.Walk(visitor, file)

	basicBlocks := visitor.GetBasicBlocks()

	numberOfBasicBlocks := len(basicBlocks)
	for index, bBlock := range basicBlocks {
		if bBlock.Type != FOR_BODY && bBlock.Type != ELSE_CONDITION &&
		    bBlock.Type != ELSE_BODY && bBlock.Type != COMM_CLAUSE &&
		    bBlock.Type != CASE_CLAUSE && bBlock.Type != RETURN_STMT {
			if numberOfBasicBlocks > index+1 {
				bBlock.AddSuccessorBlock(basicBlocks[index+1])
			}
		}
 }

 return basicBlocks, nil
}

func PrintBasicBlocks(basicBlocks []*BasicBlock) {
 for _, bb := range basicBlocks {
  log.Printf("%d) %s (EndLine: %d)\n", bb.Number, bb.Type.String()
      , bb.EndLine)

  for _, sBB := range bb.GetSuccessorBlocks() {
   log.Printf("\t-> (%d) %s (EndLine: %d)\n", sBB.Number, sBB.Type
       .String(), sBB.EndLine)
  }
 }
}

func GetBasicBlockTypeFromStmt(stmtList []ast.Stmt) (
    BasicBlockType, ast.Stmt) {
 for _, stmt := range stmtList {
  switch stmt.(type) {
  case *ast.ReturnStmt:
   return RETURN_STMT, stmt
  case *ast.CaseClause:
   return CASE_CLAUSE, stmt
  case *ast.SwitchStmt:
   return SWITCH_STATEMENT, stmt
  }
 }
 return UNKNOWN, nil
}

func (v *visitor) Visit(node ast.Node) ast.Visitor {
 if node != nil {
  switch t := node.(type) {

  case *ast.FuncDecl:
   funcDeclBlock := v.AddBasicBlock(FUNCTION_ENTRY, t.Pos())
   // Set information about Function in the block.
   funcDeclBlock.FunctionName = t.Name.Name
   funcDeclBlock.FunctionDeclLine = v.sourceFileSet.File(t.Pos()).
       Line(t.Pos())

   if t.Body != nil {
    for _, s := range t.Body.List {
     if _, ok := s.(*ast.ReturnStmt); ok {
      v.returnBlock = v.AddBasicBlock(RETURN_STMT, s.End())
     }
    }

    if v.returnBlock == nil {
     v.returnBlock = v.AddBasicBlock(RETURN_STMT, t.End())
    }

    //Visit all statements in body.
    for _, s := range t.Body.List {
     v.Visit(s)
    }
   }
```

```

    v.returnBlock = nil
    return nil

   case *ast.ReturnStmt:
    prevBlock := v.lastBlock
    v.returnBlock = v.AddBasicBlock(RETURN_STMT, t.Pos())

    if prevBlock != nil && prevBlock.Type != RETURN_STMT && len(
        prevBlock.successor) == 0 {
     prevBlock.AddSuccessorBlock(v.returnBlock)
    }

    if v.switchBlock != nil {
     v.switchBlock.AddSuccessorBlock(v.returnBlock)
    }

   case *ast.GoStmt:
    v.AddBasicBlock(GO_STATEMENT, t.Pos())

   case *ast.IfStmt:
    ifBlock := v.AddBasicBlock(IF_CONDITION, t.Pos())

    for _, stmt := range t.Body.List {
     v.Visit(stmt)
    }

    var elseConditionBlock, elseBodyBlock *BasicBlock
    if t.Else != nil {

     if v.lastBlock != nil && v.lastBlock.Type != RETURN_STMT {
      elseConditionBlock = v.AddBasicBlock(ELSE_CONDITION, t.Else.
          Pos())
     } else {
      // We don't want to set return block as successor to
          elseCondition.
      v.returnBlock = nil
     }

     elseBodyBlock = v.AddBasicBlock(ELSE_BODY, t.Else.End())
     ifBlock.AddSuccessorBlock(elseBodyBlock)

     if v.returnBlock != nil {
      if elseConditionBlock != nil {
       elseConditionBlock.AddSuccessorBlock(v.returnBlock)
      }
      elseBodyBlock.AddSuccessorBlock(v.returnBlock)
     }
    }

   case *ast.ForStmt:
    v.forBlock = v.AddBasicBlock(FOR_STATEMENT, t.Pos())

    if v.returnBlock != nil {
     v.forBlock.AddSuccessorBlock(v.returnBlock)
    }

    tmpReturnBlock := v.returnBlock
    tmpForBlock := v.forBlock
```

```
  v.returnBlock = v.forBlock

  for _, s := range t.Body.List {
   v.Visit(s)
  }

  v.returnBlock = tmpReturnBlock
  v.forBlock = tmpForBlock

  if v.lastBlock.Type == FOR_STATEMENT {
   v.AddBasicBlock(FOR_BODY, t.End())
  }

  if v.lastBlock.Type != RETURN_STMT {
   v.lastBlock.AddSuccessorBlock(v.forBlock)
  }

  v.forBlock = nil
  return nil

 case *ast.SwitchStmt:
  v.switchBlock = v.AddBasicBlock(SWITCH_STATEMENT, t.Pos())
  if v.forBlock != nil {
   v.forBlock.AddSuccessorBlock(v.switchBlock)
   v.switchBlock.AddSuccessorBlock(v.forBlock)
  }

  if v.returnBlock != nil {
   v.switchBlock.AddSuccessorBlock(v.returnBlock)
  }

  for _, s := range t.Body.List {
   v.Visit(s)
  }
  return nil

 case *ast.TypeSwitchStmt:
  v.switchBlock = v.AddBasicBlock(SWITCH_STATEMENT, t.Pos())
  if v.forBlock != nil {
   v.forBlock.AddSuccessorBlock(v.switchBlock)
   v.switchBlock.AddSuccessorBlock(v.forBlock)
  }

  for _, s := range t.Body.List {
   v.Visit(s)
  }
  return nil

 case *ast.SelectStmt:
  v.switchBlock = v.AddBasicBlock(SELECT_STATEMENT, t.Pos())
  if v.forBlock != nil {
   v.forBlock.AddSuccessorBlock(v.switchBlock)
   v.switchBlock.AddSuccessorBlock(v.forBlock)
  }

  for _, s := range t.Body.List {
   v.Visit(s)
  }
  return nil
```

```

case *ast.CaseClause:
 var caseClause *BasicBlock
 if basicBlockType, s := GetBasicBlockTypeFromStmt(t.Body); basicBlockType != UNKNOWN {
  caseClause = v.AddBasicBlock(basicBlockType, s.Pos())
 } else {
  caseClause = v.AddBasicBlock(CASE_CLAUSE, t.End())
 }

 if v.forBlock != nil {
  caseClause.AddSuccessorBlock(v.forBlock)
 }

 if v.switchBlock != nil {
  v.switchBlock.AddSuccessorBlock(caseClause)
 }

 if v.returnBlock != nil {
  caseClause.AddSuccessorBlock(v.returnBlock)
 }

 tmpSwitchBlock := v.switchBlock
 tmpReturnBLock := v.returnBlock
 for _, s := range t.Body {
  v.Visit(s)
 }
 v.switchBlock = tmpSwitchBlock
 v.returnBlock = tmpReturnBLock

 if v.returnBlock != nil && caseClause.Type != RETURN_STMT && caseClause.Type != SWITCH_STATEMENT {
   : This must be refactored more beautiful
  containsForStatement := false
  for _, b := range caseClause.GetSuccessorBlocks() {
   if b.Type == FOR_STATEMENT {
    containsForStatement = true
   }
  }
  if !containsForStatement {
   caseClause.AddSuccessorBlock(v.returnBlock)
  }
 }

case *ast.CommClause:
 var caseClause *BasicBlock
 if basicBlockType, s := GetBasicBlockTypeFromStmt(t.Body); basicBlockType != UNKNOWN {
  caseClause = v.AddBasicBlock(basicBlockType, s.Pos())
 } else {
  caseClause = v.AddBasicBlock(COMM_CLAUSE, t.End())
 }

 if v.forBlock != nil {
  caseClause.AddSuccessorBlock(v.forBlock)
 }

 if v.switchBlock != nil {
  v.switchBlock.AddSuccessorBlock(caseClause)
```

```
   }

   if v.returnBlock != nil {
    caseClause.AddSuccessorBlock(v.returnBlock)
   }

   tmpSwitchBlock := v.switchBlock
   tmpReturnBLock := v.returnBlock
   for _, s := range t.Body {
    v.Visit(s)
   }
   v.switchBlock = tmpSwitchBlock
   v.returnBlock = tmpReturnBLock

   if v.returnBlock != nil && caseClause.Type != RETURN_STMT && caseClause.Type != SWITCH_STATEMENT {
    containsForStatement := false
    for _, b := range caseClause.GetSuccessorBlocks() {
     if b.Type == FOR_STATEMENT {
      containsForStatement = true
     }
    }
    if !containsForStatement {
     caseClause.AddSuccessorBlock(v.returnBlock)
    }
   }
  }
 }
 return v
}
```

Bibliography

[1] Frances E. Allen. “Control Flow Analysis.” In: *ACM SIGPLAN Notices 5.7* (1970).

[2] The Go Authors. *golang/go: The Go Programming Language*. URL: https://github.com/golang/go (visited on 07/27/2016).

[3] Michael F. Plass Bengt Aspvall and Robert Endre Tarjan. “A linear-time algorithm for testing the truth of certain quantified boolean formulas.” In: *Information Processing Letters Volume 8, number 3* (1979).

[4] G. Ann Campbell and Patroklos P. Papapetrou. FOREWORD BY Olivier Gaudin. *SonarQube in Action*. Manning Publications, 2013. ISBN: 978-1617290954.

[5] Edsger W. Dijkstra. “EWD 215: A Case against the GO TO Statement.” In: *Communication of the ACM, Volume 11, No. 3* (1968).

[6] Alan A. A. Donovan and Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley, 2015. ISBN: 978-0134190440.

[7] Hanne R. Nielson Flemming Nielson and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999. ISBN: 3540654100.

[8] Google. *PMD*. URL: https://pmd.github.io/ (visited on 08/07/2016).

[9] Google. *The Go Programming Language*. URL: https://golang.org/ (visited on 07/27/2016).

[10] C.A.R. Hoare. “Communicating Sequential Processes.” In: *Communication of the ACM* (1978).

[11] S. C. Johnson. *UNIX Programmer's Manual. Lint, A C Program Checker*. Seventh Edition, Volume 2A. 1979. URL: http://cm.bell-labs.com/7thEdMan/v7vol2a.pdf (visited on 07/13/2015).

[12] Martin Fowler with Kent Beck et al. *Refactoring : improving the design of existing code*. Addison-Wesley, 1999.

[13] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language, Second Edition*. Prentice-Hall, 1988.

[14] University of Maryland. *FindBugs - Find Bugs in Java Programs*. URL: http://findbugs.sourceforge.net/ (visited on 08/08/2016).

[15] Thomas J. McCabe. “A Complexity Measure.” In: *IEEE Transaction on Software Engineering, Vol. SE-2, No. 4* (1976).

[16] Carnegie Mellon University Software Engineering Institute. *C4 Software Technology Reference Guide - A Prototype*. Tech. rep. CMU/SEI-97-HB-001. 1997, Jan. 1997.

[17] Mark Summerfield. *Programming in Go: Creating Application for the 21st Century*. Addison-Wesley, 2012. ISBN: 978-0321774637.

[18] www.json.org. *Introducing JSON*. URL: http://www.json.org/ (visited on 07/13/2016).