DEPARTMENT OF
COMPUTER SCIENCE

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

JORGE GABRIEL GRAVE COELHO
BSc in Computer Science

AUTOMATIC DETECTION AND RESOLUTION OF DEADLOCKS IN GO PROGRAMS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2022

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

AUTOMATIC DETECTION AND RESOLUTION OF DEADLOCKS IN GO PROGRAMS

JORGE GABRIEL GRAVE COELHO

BSc in Computer Science

Adviser: António Maria Lobo César Alarcão Ravara
Associate Professor, FCT | NOVA University Lisbon

Co-adviser: João Manuel dos Santos Lourenço
Associate Professor, FCT | NOVA University Lisbon

Examination Committee

Chair: Henrique João Lopes Domingos
Associate Professor, FCT | NOVA University Lisbon

Rapporteur: Marco Giunti
Researcher, University of Beira Interior

Adviser: António Maria Lobo César Alarcão Ravara
Associate Professor, FCT | NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2022

Automatic detection and resolution of deadlocks in Go programs

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.17) [17].

Abstract

The Go programming language is acquiring momentum in the development of concurrent software. Even though Go supports the shared-memory model, the message-passing alternative is the favoured idiomatic approach. Naturally, this practice is not exempt of the usual difficulties: programs may deadlock and the language run-time has only very basic support for deadlock detection. Previous research on deadlock detection mainly focused on shared-memory concurrency models. For mainstream languages, tools and approaches specific to the message-passing paradigm are scarce and incipient. There is however a large body of work on models of concurrency that only recently started to be applied to languages like Go. Since the Go run-time lets many deadlocks pass unnoticed, and the existing solutions provided by third party tools detect many deadlocks but only try to fix a limited set of specific patterns, imposing severe conditions to do so, there is a clear need for more general deadlock resolution strategies, going beyond prevention and avoidance. To gain insight on real-world deadlock bugs, we first built and categorized a collection of bugs sourced from high-profile open-source Go projects. Next, we extended and implemented an algorithm that takes an abstraction of the communication behaviour of the program and, when all the communication operations on channels necessary for progress are present, but a deadlock is possible, presents the problem and offers a possible resolution for the error. The extensions allows our approach to analyse a much wider range of real world programs. We conclude with an evaluation, comparing with two other state-of-the-art solutions.

Keywords: Deadlock detection, Deadlock resolution, Static analysis, Go programming language, Message-passing

Resumo

A linguagem de programação Go tem ganhado tração no desenvolvimento de software concorrente. Apesar de o Go suportar o modelo de partilha de memória, o modelo alternativo de partilha de mensagens é a abordagem idiomática. Naturalmente, esta prática não está isenta das dificuldades usuais: os programas podem bloquear e o *run-time* da linguagem só possui um suporte muito básico para a deteção destes bloqueios. Investigação anterior na deteção de bloqueios focou principalmente no modelo de partilha de memória. Para linguagens convencionais, ferramentas e abordagens dedicadas ao paradigma de passagem de mensagens são escassas e incipientes. No entanto, existe um grande conjunto de trabalhos sobre modelos de concorrência que só recentemente começou a ser aplicado em linguagens como o Go. Visto que o *run-time* do Go deixa muitos bloqueios passar despercebidos e as soluções existentes detetam muitos bloqueios, mas só tentam resolver um conjunto muito pequeno de padrões. De modo a ganhar conhecimento sobre erros de bloqueio reais, nós começámos por construir e categorizar uma coleção de erros obtidos a partir de projetos Go open-source de alto perfil. De seguida, nós estendemos e implementámos um algoritmo que recebe uma abstração do comportamento da comunicação de um programa e quando todas as operações de comunicação nos canais necessários para o progresso estão presentes, mas um bloqueio é possível, apresenta o erro e oferece uma possível resolução do erro. A nossa extensão permite analisar um conjunto muito maior de programas reais. Concluímos com uma avaliação, comparando com duas outras soluções do estado da arte.

Palavras-chave: Deteção de bloqueios, Resolução de bloqueios, Análise estática, Linguagem de programação Go, Passagem de mensagens

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

In this chapter we will provide context for the need for concurrent programming and expose some inherent challenges with the use of it. Next we present a detailed description of the problem that address, followed by an overview of the objectives of the thesis and our contributions.

1.1 Context

In the last couple of decades the development of new processors has suffered a major shift. Initially the improvements to the performance of processor architectures focused on adding new instructions, speeding up existing ones, and increasing the overall clock frequency. More recently, the progress has been aimed at increasing the parallelism of the processor [7]. This can be seen with the advent of processors that feature Simultaneous Multithreading[1] and multi-core designs. The latter becoming the most prevalent and successful with modern processors containing up to 128 cores [9].

On the software side, this shift means that, in order to take full advantage of the resources of a modern computer, it is necessary to make use of concurrent programming. By writing concurrent programs that are able to utilize the hardware parallelism of modern processors, it is possible to increase considerably their performance.

It is also of note that certain problem domains can be better suited to an implementation that uses concurrent programming, such as programs that are primarily event-driven.

Unfortunately, concurrent programming brings many downsides. The first of which is that writing concurrent programs that are bug-free can very challenging. Programmers naturally think sequentially and as such, can have a difficult time reasoning about all the potential interactions and behaviours a concurrent program can have, specially in large codebases.

To make matters worse, concurrent programming often causes non-deterministic behaviour. The non-deterministic scheduling of the threads can lead to different results

[1] Simultaneous Multithreading is a technique that allows multiple instructions to execute in parallel while sharing some execution resources.

for two executions of the same program. In the worst cases, this behaviour leads to Heisenbugs, bugs that are very uncommon and difficult to reproduce. Due to the difficulty in reproducing and analysing these bugs, they can be introduced and remain hidden and silent for a long time before being caught and fixed [29].

In a survey conducted on 684 employees at Microsoft [10], Godefroid and Nagappan measured that not only a majority of staff dealt with concurrency bugs at least monthly, but also that over 70% of the bugs take more than one day to debug and fix, concluding that debugging and fixing concurrency bugs take up a significant portion of the developer's time.

Concurrency bugs can be categorized into various types, such as data races, atomicity violations, order violations, and deadlocks. In this work we will tackle deadlock bugs. A deadlock can be defined as a permanent inter-blocking of a set of processes that either compete for system resources or communicate with each other. In the context of a computer program, a deadlock leads to one or more threads becoming blocked and in most cases preventing further progress of the program.

Although deadlock bugs are generally the easiest to notice when they occur, often times leading to a complete halt of the program's progress, they can still be difficult to reproduce, analyse, and fix. In a study of concurrency bugs in large scale software projects written in C/C++, almost 30% of the concurrency bugs were deadlock bugs [18]. More concretely, for a deadlock to occur all of the following conditions must hold (Coffman conditions [3]):

- **Mutual exclusion.** The shared resource must be accessed by a single process at a time.
- **Hold and Wait.** A process must already be holding a resource and is waiting for access to additional resources.
- **No pre-emption.** A process must only give up the resource voluntarily.
- **Circular wait.** A circular wait must form where each process is waiting for access to a resource that is held by the next process in the chain.

Several strategies exist for dealing with deadlocks [12]:

- **Ignoring deadlocks.** The simplest and perhaps most employed strategy is to disregard the problem. The deadlock might just not occur frequently enough to be bothersome, or the software can just be restarted without significant data loss. It has the advantage of not imposing extra overhead that other solutions might impose and the disadvantage that it isn't a "proper" solution.
- **Deadlock prevention.** Since all the Coffman conditions must be met for a deadlock to occur, a deadlock can be prevented by ensuring that at least one of those conditions cannot be met. For example, the *Hold and Wait* condition can be removed if the

processes acquire all the resources needed at once before proceeding. This strategy, however, implies the advance knowledge of the resources the process will require, something that is frequently not possible or very impractical to implement.

- **Deadlock avoidance.** A deadlock can be avoided by ensuring that the system never enters a state where a deadlock is possible. With the knowledge of the currently acquired resources and the maximum number of resources each process can allocate, it is possible to determine if a new acquisition will evolve the system into a state where a deadlock is possible. However, this strategy not only imposes additional overhead with the need to track resource allocations, but also requires the *a priori* knowledge of the maximum number of resources each process can allocate. With the latter being specially impractical or impossible to implement.

- **Deadlock detection.** With this strategy deadlocks can and will occur. But after detecting a deadlock, a procedure can be employed to resolve the deadlock and resume execution. One possible procedure is to release the offending resources and roll back the changes made to a point where no conflicting resources were held. This will clear the deadlock and the rolled back process can restart. Since this strategy does not prevent the process from reaching a deadlock, the same deadlock can occur indefinitely, possibly leading to starvation.

Overall these strategies either impose overhead in the execution of the program, require advance knowledge of what resources a process will need, or require the ability to rollback a process to a previous state. Each of these requirements are often times not practical, frequently demanding a complete refactoring of the program's behaviour, or a considerable increase in the program's complexity. This has lead to the emergence of new programming languages and tools with features and programming models that help the programmer develop programs with fewer deadlocks. These tools can either statically or dynamically analyse the program's code or execution to evaluate if a program can deadlock. With the context of where the deadlock occurs in the program's code, developers can analyse the codebase and determine the changes needed to prevent the deadlock from happening.

Significant research has been conducted to develop tools and algorithms that detect deadlocks on languages that feature a shared-memory concurrency model, such as C/C++ and Java. However, in the recent years, new programming languages that have message-passing concurrency models have risen significantly in popularity. Go and Rust are two of the most prominent, reaching 12th and 26th place, respectively, in the TIOBE programming language popularity index [28].

The message-passing concurrency model brings new problems that are not fully addressed by the research targeting languages that feature concurrency models based on shared-memory.

In the 2019 Go developer survey [14], developers said that Go's concurrency, was not only important, but were also satisfied with it's implementation (Figure 1.1). This, once again, echoes the need for concurrency in today's computing environment. It also shows that overall, developers are satisfied with the primitives Go provides. On the other hand, debugging concurrency while only slightly lower in the importance metric, it is the aspect that scored lowest in the satisfaction metric. Indicating that most developers are not content with the current environment of concurrency debugging tools, even though it is an aspect of Go's development that the majority feel it is important.

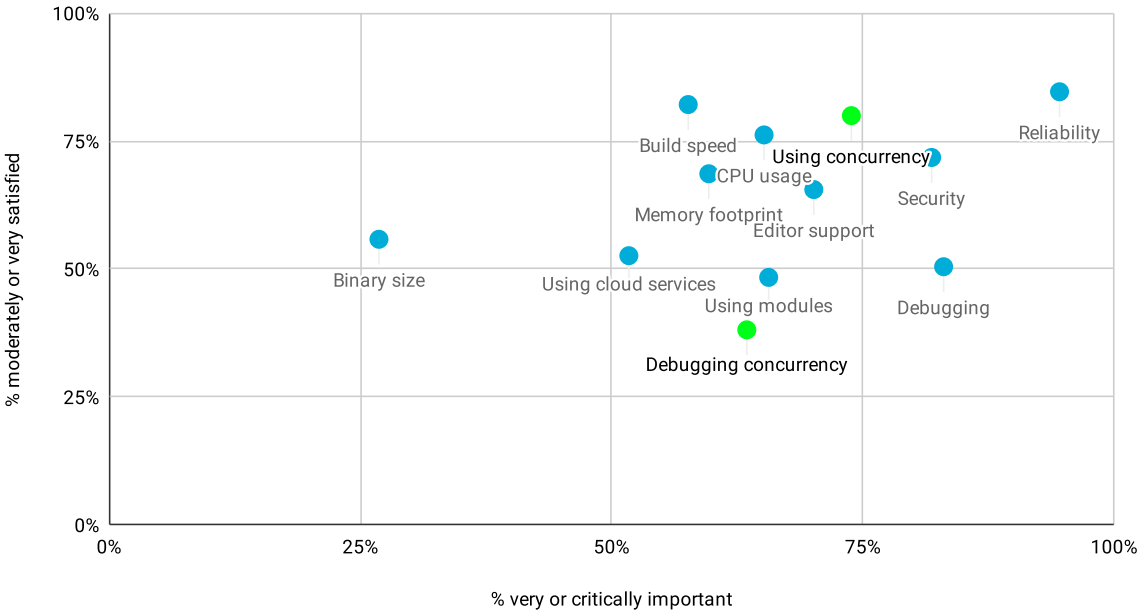

Figure 1.1: Importance vs. Satisfaction of various aspects of Go development [14].

1.2 Problem

> *How to automatically detect and avoid common deadlock patterns in Go code, proposing fixes to the faulty part of the code?*

As we have seen, deadlock bugs not only are difficult to debug and fix, but also can cause severe failures. In the recent years there has been a considerable effort and research published on tools and methods to detect deadlock bugs in Go, ranging from simple built-in dynamic analysis to static analysis on inferred behaviour types or even the compiled Go binary. Typically, once a deadlock is detected, the tool provides some debugging information to the user such as where and how the deadlock occurs. With this information the developer is then tasked with resolving the deadlock. As it is done manually, this process can be error-prone and slow. This second step of the process can be improved. If the deadlock resolution step is automated, the process can be even more streamlined, not only saving countless hours of development time but also providing more reliable results.

Program analysis however, it's not without its limitations. Rice's theorem states that all non-trivial properties of the behaviour of programs (written in Turing-complete programming languages) are undecidable. However, it is still possible to over-approximate and obtain useful answers for realistic programs [21]. In practice, this means that false positives are possible, resulting from the approximated decidable problem not matching in behaviour with the original undecidable problem. Hereby, program analysis tools must be developed to provide results as accurate as possible, reporting little to no false positives. Otherwise, the value of the tool to the user will be greatly diminished.

1.3 Objectives

The work has two main objectives:

Bug collection. The first objective is to build a collection of common Go code patterns that induce deadlocks when executed. Together with a taxonomy on the type of bugs, the dataset helps determine the efficacy of deadlock detection solutions. Moreover, with this collection of deadlock code patterns, we performed an evaluation of existing solutions.

Deadlock detection and resolution. The second objective is to build and implement a solution that statically analyses a Go codebase to determine potential deadlock bugs and give hints of possible solutions to the programmer. The resulting tool could then be integrated into a programmer's development environment and, in real-time, provide feedback. It would also allow the programmer to find and fix the bugs even before the buggy code is deployed and executed.

1.4 Contributions

The main contributions of our work are:

Overview of the state of the art (§3) We provide an overview and critique of the state of the art in relation to the detection of deadlocks in Go programs, including one approach that provides a resolution to some types of deadlocks.

Bug collection (§4) We have built a collection of deadlocks bugs, gathered from real-world Go projects. The collection of contains snippets of code that retain just the bug inducing part of the code, and a taxonomy where we categorized the deadlocks and associated fixes. With this data we gathered some insights into the structure of the deadlocks and fixes.

Deadlock detection and resolution (§5) Following the work by Francalanza, Giunti, and Ravara [8] and Almeida [1], we developed a solution capable of statically analysing Go code in a fully automated manner, with the objective of detecting deadlock bugs

caused by the misuse of the message passing concurrency model, and providing resolutions to the deadlocks found. In addition to a considerable refactoring and optimization of the existing tool, we added support for two prominent features of the Go language: the `select` statement, and recursive processes. These changes greatly improve the applicability of the tool when analysing real world code. For each deadlock found, our tool then uses one of two strategies to resolve the deadlock.

Evaluation (§6) The final contribution is an evaluation of our approach, as compared with two others, using the bug collection as a benchmark suite of deadlock bugs.

The GoDDaR Tool. A primary contribution of our work is the **Go D**eadlock **D**etection **a**nd **R**esolution tool, where we implemented the approach described in this document. GoDDaR provides an end-to-end solution, capable of analysing source code in Go and outputting deadlocks found in the code along with possible resolutions for each deadlock. The tool operates in a fully automated manner, requiring no input from the programmer and no changes to the original source code.

Our solution is implemented in OCaml and can be accessed at https://github.com/JorgeGCoelho/GoDDaR, along with usage instructions and examples.

2

Background

This chapter will start by highlight two modern programming languages that were developed with safe concurrent programming in mind, with each taking a different approach, next we will describe some common approaches for program analysis, and we will finish by providing a comprehensive description of the CCS process calculus, as it will play an important role in our approach.

2.1 Modern Concurrent Programming Languages

2.1.1 Rust Language

Rust is a new programming language [26] that focuses on performance and safety, particularly with regards to safe concurrency. It is a compiled language that checks most of the safety guaranties at compile time. In order to be performant, Rust doesn't feature a runtime library or automatic garbage collection, instead it relies on techniques such as Resource Acquisition Is Initialization (RAII) to provide its features during compilation.

While many languages trade-off high-level safety guarantees with low-level control of the resources, Rust sets out to achieve both. With mechanisms for low-level memory management, Rust can be good choice for projects that would otherwise be implemented in C/C++.

In order to provide memory safety, Rust features an ownership and borrowing system where access to aliases of references are restricted. If a reference is not aliased, then it only has one owner and the reference is mutable, meaning the contents of the reference can be modified. On the other hand, if the reference is aliased, then the contents of the reference are immutable.

Together with other strategies like forced initialization of variables, the absence of null pointers, and bounds-checking, Rust provides strong memory safety guarantees, capable of preventing bugs such as null pointer dereferences, use-after-free, double-free, dangling pointers, and buffer overreads/overwrites.

The ownership system does not allow multiple threads to have mutable references to the same data at the same time. With only immutable access to shared data, communication

between threads would not be possible, and as such concurrent programming would not be practical.

In order to allow threads to share and modify memory safely the Rust standard library features types that implement interior mutability. Interior mutability allows the mutation of data even when there are immutable references to that data. One type that implements interior mutability is the `std::sync::Mutex<T>`. The `Mutex<T>`, as the name suggests, implements a mutual exclusion primitive that protects shared data. It allows access/mutation of the enclosed data by a single thread at a time.

Another concurrency primitive that Rust provides are channels. Channels allow message-based communication, much like the channels featured in Go. When a reference is sent via a channel, the sender loses the ownership of the reference, while the receiver acquires it. By transferring the ownership, the ownership system will prevent the sender from accessing the sent reference again, ensuring memory safety.

Unfortunately, while all these strategies are effective at preventing data-races [4], they do not prevent deadlocks. It is still possible to create situations where a circular dependency is formed between threads that prevents them from progressing.

2.1.2 Go Language

Go is a compiled, concurrent, garbage collected, and statically typed programming language [25] initially designed and developed at Google in 2007. It was designed to improve programming productivity in the modern era of multicore machines, networked systems, and massive codebases.

As part of the Go's design goals to improve productivity with regards to concurrent programming, Go features a set built-in concurrency primitives: lightweight processes (Goroutines), channels, and the *select* statement.

2.1.2.1 Goroutines

Goroutines are Go's implementation of lightweight-processes. The goroutines are scheduled by the Go runtime to be executed on a kernel level thread, allowing for more efficient context switches. Together with a independent call stack that grows dynamically, goroutines offer a much cheaper thread management strategy and as such Go programs can efficiently have thousands of goroutines running concurrently.

Goroutines are very easy to create. Go introduces the **go** statement that, when placed before a function call, executes the function in a newly created goroutine. The **go** statement returns immediately, allowing both goroutines to continue executing concurrently.

The Listing 1 shows an simple Hello World program that makes use of goroutines.

In the Listing we can also observe two more characteristics of the Go language. The first is that Go features anonymous functions. These functions can be assigned to a variable or invoked directly. Together with the **go** statement, this provides a very simple and effortless way of introducing concurrency. The second is that when the main goroutine returns the

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    fmt.Println("Hello ")
    go func() {
        fmt.Println("World!")
    }()
    time.Sleep(1 * time.Second)
}
```

Listing 1: Example of the creation of a goroutine.

Go runtime terminates the program without waiting for other goroutines to complete. In order to work around that behaviour we suspend the execution of the main goroutine before returning in order to allow the second goroutine to execute its print statement. While Go features better methods that provide synchronization such as waitgroups and channels, in this example a simple sleep works well enough.

2.1.2.2 Channels

With channels, Go implements a CSP-style concurrency model. Channels work similarly to UNIX pipes, allowing goroutines to send and receive data between each other.

To create channels, Go's has the built-in functions `make(channel_type)` and `make(channel_type, capacity)`. Without the capacity specified, Go assumes a capacity of 0. The capacity of the channel will define its behaviour. The behaviour of channels can be classified into two types: asynchronous (buffered) or synchronous (unbuffered).

With a non-0 buffer capacity, channels will act as a FIFO queue, and operations of the channel will be asynchronous if buffer is not empty (receives) or not full (sends). When the channel's buffer is full, send operations will block until an entry in the queue is free. Receive operations will block if the channel's buffer is empty, and will only unblock when it's able to retrieve a value from the channel's queue.

With a buffer capacity of 0, a channel is unbuffered and the operations on the channel act synchronously. Each send/receive operation on the channel will block until another goroutine executes the opposite operation. This behaviour provides a simple yet effective way of ensuring synchronization between goroutines.

In the Listing 2 we provide an example of a Hello World program that makes use of channels.

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Print("Hello ")
7      c := make(chan string)
8      c <- "World!" // Block
9      fmt.Println(<- c)
10 }
```

Listing 2: Example of the use of channels, with a blocking bug.

The program starts off by printing `"Hello "`. In line 7 a channel that sends/receives values of type `string` is created and assigned to variable c. On the next line we send the string `"World!"` through the channel. Finally, the string is received from the channel and printed.

Unfortunately the example will never print the full `"Hello World!"`. In the current implementation of the program, the channel created is synchronous and as such the send operation will block indefinitely because there will never be another goroutine to handshake and receive the value.

There are two simple ways of fixing the program.

The first is to change the capacity of the channel. With a non-0 capacity, it would allow the send operation to place the value in the channel's buffer and not block the execution. At line 9, the receive would then fetch the string from the channel's buffer and print the string.

```
1  package main
2
3  import "fmt"
4
5  func main() {
6      fmt.Print("Hello ")
7      c := make(chan string)
8      go func() {
9          fmt.Println(<- c)
10         c <- ""
11     }()
12     c <- "World!"
13     <- c
14 }
```

Listing 3: Example of the use of channels, fixed by introducing an extra goroutine.

The second way is to move the receive operation to a separate goroutine, see Listing 3, line 9. With the send and receive operations in two goroutines, they are able to handshake and transfer the value from the main goroutine to the child goroutine. The use of a synchronous channel also enforces synchronization, ensuring that both goroutines reach a certain point before progressing. In this case, it ensures that the `"World!"` string is printed after `"Hello "`.

In the fixed example, we also added an extra pair of send/receive operations (lines 10 and 13) to make sure that the child goroutine prints the string before the main goroutine returns.

Another important characteristic of the Go channels is that channels are first-class objects. Allowing them to be stored in variables, passed as arguments to functions, stored in record types, and even be sent across channels. This allows for great flexibility in the communication structure of Go programs.

Go's runtime implementation of channels block when an operation is performed over a **`nil`** channel. Channels can also be closed. Send operations over closed channels cause the Go program to panic and receive operations immediately return the zero value of the channel's type.

2.1.2.3 Select Statement

The final concurrency construct that Go provides is the **`select`** statement. The select statement features syntax is similar to the switch statement found in many programming languages, such as C and Java.

For each case statement, the runtime checks if the send/receive operation is able to be performed. If so the operation is executed and the code for that case is also executed. If more than one case statement can proceed, a single case is chosen at random. If no case statement can proceed, the select statement blocks until one of the communication operations can proceed.

In Listing 4 we provide an example of the use of the select statement. In this example the select statement is used to implement a timeout for an operation. We make use of the `time.After()` function from the Go's standard library. This function returns a channel that, after the given duration has elapsed, sends the current time through the channel. The select statement will block until one case statements can proceed, the one that will be available first will be executed. With this behaviour, either the `timeConsumingOperation()` sends its data to the `output` channel within 10 seconds or after the 10 seconds elapses, the `time.After()` function sends a value through the channel.

The select statement also allows the addition of a **`default`** case. If a select statement has a default case, the default case will be chosen if no other cases can proceed. This effectively makes the select statement non-blocking.

Together, these three constructs allow for a very powerful yet simple method of writing concurrent programs.

```go
package main

import (
    "fmt"
    "time"
)

func timeConsumingOperation(output chan string) {
    ...
}

func main() {
    output := make(chan string, 1)
    go timeConsumingOperation(output)
    timeout := time.After(10*time.Second)
    select {
        case data := <- output:
            fmt.Println(data)
        case <- timeout:
            fmt.Println("Timed out")
    }
}
```

Listing 4: Example of the use of the select statement.

The language still allows concurrency models based on sharing memory. All goroutines share a single memory address space, and in the standard library Go provides basic synchronization primitives such as mutexes and conditional variables.

Compared to Rust, Go provides a more lenient approach to safe concurrent programming. Instead of imposing a complex set of restrictions in the code, that can be checked at compiled time, Go relies more on conventions based on message-passing, that are less likely to introduce concurrency bugs. In order to reinforce this, Go promotes the following motto:

> Don't communicate by sharing memory;
> share memory by communicating.

2.2 Approaches for Program Analysis

There are three general approaches to analyse the behaviour of a program. The program analysis can be performed during execution of the program (dynamic analysis), without executing the program (static analysis), or an hybrid of both. Generally, dynamic analysis incurs an overhead during execution and can only evaluate the code paths that are executed, requiring a exhaustive testing harness to analyse the whole program. Without the requirement of executing the program, static analysis can be more versatile and

practical. Not only can it analyse the program as a whole, but also frequently requires less or no input from the user, allowing it to be fully automated and incorporated to the development life cycle.

The dynamic analysis approach involves applying some kind of code instrumentation, collecting the events or traces necessary to evaluate the state of the program. Either during execution or in a subsequent offline analysis, the traces are collected to build a model of the current state of the program. With a resource-allocation graph, it can be determined if a cycle has formed between the threads, indicating the occurrence of a deadlock.

For static analysis various techniques have been created. One of which is abstract interpretation, where every statement of the program is modelled in an abstract machine. Through this process, the analysis can obtain an approximation to the possible behaviour of the program. The goal it to obtain a computable semantic interpretation. One simple example is the analysis of the sign of a series of integer manipulations, keeping only their state (+, -, ±). In the case of a multiplication the sign of the result can be derived with no loss of precision, while with the sum the sign of the result depends on the values of the operands, and as so the abstraction may lose precision.

Another method for analysing programs is through the use of a constraint solver. The strategy is divided into two steps. The constraint generation back-end, and a constraint solver front-end. The back-end generates a collection of constraints modelling the possible behaviour of the program and the properties to be evaluated. With the generated constraints, the front-end is used to find a solution. If a solution is found the property is satisfied.

Model checking is a technique where a finite-state model is checked if it meets a given specification. First, a model is obtained that abstracts a system or program, describing the possible behaviour of the system in a precise and unambiguous manner. This step can be performed either manually or automatically, in which case the model can be obtained from an existing program specification. The models are then accompanied with algorithms that systematically explore all the states in the system. In this way it can be shown if a model satisfies a given property. These properties can specify liveness or safety requirements.

Finally, it is possible to use a mix of both static and dynamic analysis. Each type of analysis has their trade-offs. Static analysis provides a more sound approach, although more conservative due to abstraction, and dynamic analysis generally is more precise due to no approximation but otherwise unsound. With a hybrid approach it can be possible to take advantage of each approach's strengths, resulting in a more correct and complete analysis.

2.3 Calculus of Communicating Systems

2.3.1 Reactive Systems

The classic way of abstracting computing systems is generally that of a black box that takes inputs and returns outputs. In this view, these systems can be described by how they transform an initial state to a final state. This model, however, does not allow to adequately express some important aspects of other types of computing systems, namely reactive systems. Reactive systems differ in two major ways, non-termination and the notion of interaction.

For many of the algorithms and systems we build, termination is an expected and desirable behaviour, often necessary to obtain the results of the computation. However, for some computing systems the execution is inherently non-terminating. As an example, it is reasonable to expect a control program that monitors a nuclear reactor to never terminate.

With the potential for uninterrupted execution, reactive systems instead rely on interactions to communicate with their environment and in doing so, receive inputs and return outputs. These interactions play a key role in the behaviour of the system, triggering potential changes in the system's state and, in turn, influencing the environment by sending some signals back.

2.3.1.1 Modelling Reactive Systems

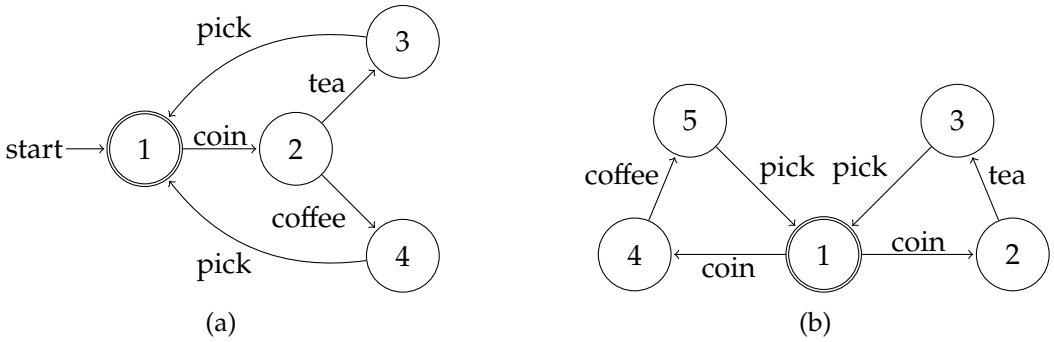

Figure 2.1: Two possible automata representing a vending machine.

A vending machine can be seen as a reactive system [11]. The machine is (usually) always available to serve the next drink and offers a means of interaction via a frontend consisting of physical buttons, slots for the insertion of coins, and a tray from which the user can pick up their drink.

In Figure 2.1, we present two possible automata that describe the behaviour of the vending machine. This vending machine accepts the following user interaction: upon the insertion of a coin, the user can choose either tea or coffee, after which the user can pick up the chosen drink.

$$\text{coin.(coffee.pick + tea.pick)} \tag{2.1}$$

$$\text{coin.(coffee + tea).pick} \tag{2.2}$$

The program (automaton) is correct if it implements (accepts) exactly the intended behaviour, i.e. the same language. If the expressions 2.1 and 2.2 are the regular expressions of the program simulating the vending machine and of the intended behaviour, respectively, then we can assert that the program is correct, since both regular expressions accept or denote the same language.

However, this method of comparing the behaviour reactive systems via automata-based models falls apart if we look at the automata in Figure 2.1b. Converting the automaton, yields the following regular expression:

$$\text{coin.coffee.pick + coin.tea.pick} \tag{2.3}$$

We can again see that the regular expression 2.3 denotes the same language as the previous two expressions 2.1 and 2.2. This would imply that the regular expression 2.3 implements the same intended behaviour. However, this is not the case. When a user inserts the coin, the automaton non-deterministically chooses to proceed to the state 2 or 4. This removes the choice from the user, preventing them from picking between coffee and tea.

With this example, we can see another language is necessary to model reactive systems. In particular, this language should:

- Allow the representation of non-terminating behaviour.
- Distinguish between input and output action.
- Support parallelism and communication.
- Have a finer notion of equivalence.

Calculus of Communicating Systems[19] (CCS) is a process calculus created by Robin Milner that models the concurrent behaviour of processes. These processes are able to communicate with their environment and between each other when executed in parallel.

Processes can be seen as a black box containing the name that identifies it and the channels over which it can interact with. The drawing in Figure 2.2 pictures the interface of a process that represents a Coffee and Tea Machine (CTM). The drawing shows that the CTM process can communicate via three different channels, named 'tea', 'coin', 'coffee', and 'take'. The communication is modelled via operations that conceptually that send or receive a value or message through the channel. From the Figure 2.2 we can say that the CTM process uses the channel 'coin' for input and the channels '$\overline{\text{tea}}$', '$\overline{\text{coffee}}$', and '$\overline{\text{take}}$' for output. These input and output operations are referred to as actions. In general a represents an input action and $\overline{a}$ an output action.

While this static information about the process can be useful, what is really necessary is to model the behaviour of the process being specified. To this end, CCS introduces a compact, but still very expressive process algebra to precisely define the behaviour of a reactive system.

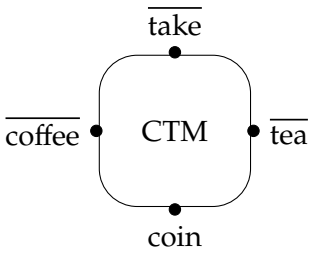

Figure 2.2: Interface of the CTM process.

2.3.2 Syntax

Assume a countable set $\mathcal{N}$ of action *names*; then CCS actions are defined as follows:

$\alpha ::=$	Actions
a	Input action
$\mid \overline{a}$	Output action
$\mid \tau$	Silent (internal) action

a and $\overline{a}$ are observable actions, while τ is an unobservable action.

We will now go over the operators of the CCS language, while giving an informal description of each of one.

Inaction The most basic process of all is the one that performs no action whatsoever. This process is expressed as **0**.

Action prefixing The first construction, and the most basic is the action prefixing. Intuitively, for the process $\overline{\text{coin}}.\overline{\text{coffee}}.\text{pick}.\mathbf{0}$, after the action $coin$ is performed the process proceeds behaving like process $\overline{\text{coffee}}.\text{pick}.\mathbf{0}$. Repeating this procedure, we can see that the process will perform the actions in sequence until it reaches **0**.

$$\begin{aligned} & \overline{\text{coin}}.\overline{\text{coffee}}.\text{pick}.\mathbf{0} \\ \xrightarrow{\overline{\text{coin}}} \quad & \overline{\text{coffee}}.\text{pick}.\mathbf{0} \\ \xrightarrow{\overline{\text{coffee}}} \quad & \text{pick}.\mathbf{0} \\ \xrightarrow{\text{pick}} \quad & \mathbf{0} \end{aligned}$$

Process definition As previously mentioned, processes can be given names. This means that we can introduce names for (complex) processes and use these names in the definition of other process descriptions. An example can be:

$$\text{User} \stackrel{\text{def}}{=} \overline{\text{coin}}.\overline{\text{coffee}}.\text{pick}.\text{User}$$

The introduction of names for processes allows us to give define recursive definitions of process behaviours and to model the non-terminating behaviour of reactive systems. The User process is one such example. By repeatedly replacing User with its definition, we obtain

$$\begin{aligned}\text{User} &\overset{\text{def}}{=} \overline{\text{coin}}.\overline{\text{coffee}}.\text{pick}.\text{User}\\ &= \overline{\text{coin}}.\overline{\text{coffee}}.\text{pick}.\overline{\text{coin}}.\overline{\text{coffee}}.\text{pick}.\text{User}\\ &= \overline{\text{coin}}.\overline{\text{coffee}}.\text{pick}.\overline{\text{coin}}.\overline{\text{coffee}}.\text{pick}.\overline{\text{coin}}.\overline{\text{coffee}}.\text{pick}.\text{User}\\ &= \underbrace{\overline{\text{coin}}.\overline{\text{coffee}}.\text{pick}\ldots\overline{\text{coin}}.\overline{\text{coffee}}.\text{pick}}_{n\text{ times}}.\text{User}\end{aligned}$$

for each positive integer n.

Choice With the previous constructs, it is not possible to represent the behaviour of the vending machine shown earlier. To allow the description of processes whose behaviour may follow different patterns of interaction, CCS offers the choice operator, written with '+'. The vending machine can be represented as the following CCS process:

$$\text{CTM} \overset{\text{def}}{=} \text{coin}.(\text{coffee}.\overline{\text{pick}}.\text{CTM} + \text{tea}.\overline{\text{pick}}.\text{CTM})$$

Intuitively, the CTM process after receiving a coin as input, may behave as either $\text{coffee}.\overline{\text{pick}}.\text{CTM}$ or $\text{tea}.\overline{\text{pick}}.\text{CTM}$ depending on the user's choice.

Parallel composition The parallel composition operation, written with '|', allows the description of systems consisting of multiple processes running in parallel.

For example, the CCS expression CTM|User describes a system composed of two processes, that run in parallel. They may communicate via the communication channels that they share and use in complementary fashion. In the interface representation of a CCS process, this expression can be illustrated as shown in Figure 2.3.

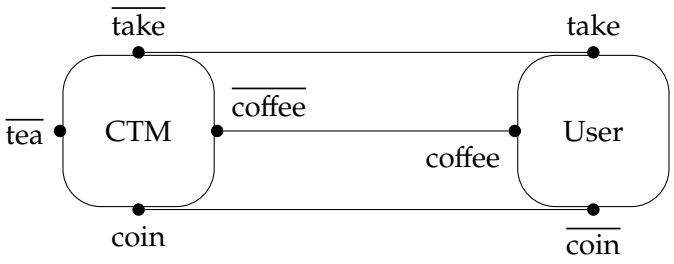

Figure 2.3: Interface of the CTM | User system.

In general, given two CCS expressions P and Q, the process $P \mid Q$ describes a system in which P and Q may proceed independently, or communicate via complementary channels

Action hiding The action hiding operator blocks the access of certain communication ports outside the scope of the process. This functions in a similar way as variables that have a delimited scope in block-based programming languages.

Previously we have composed a system with two parallel processes, CTM and User. If we append another parallel User to the system, we obtain the interface shown in Figure 2.4.

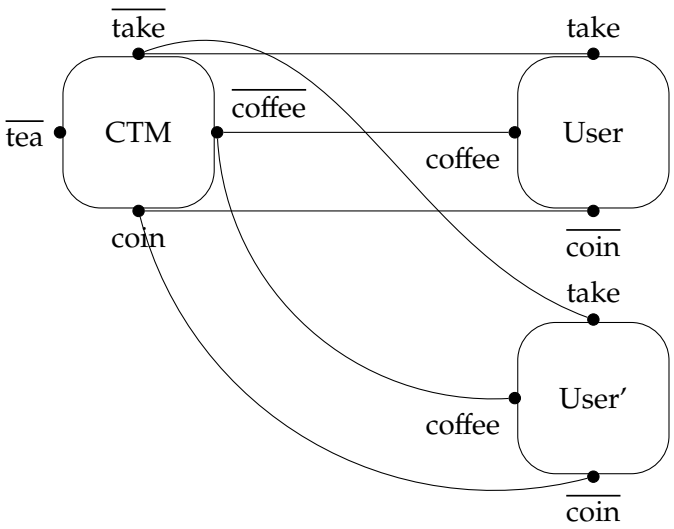

Figure 2.4: Interface of the (CTM | User) | User' system.

However, the first user might want the vending machine only for themselves. To achieve this we can delimit the scope of the channels to within the process (CTM | User):

$$\text{SelfishUser} \stackrel{\text{def}}{=} (\textbf{new}\ \text{coin}, \text{coffee}, \text{tea}, \text{take})(\text{CTM} \mid \text{User})$$

As illustrated in Figure 2.5, the second user no longer can communicate with the vending machine.

Action substitution The final operation of CCS allows the renaming of actions in a process. For example, if we want to change the vending machine from selling *coffee* and *tea* to *water* and *cola* we can do so by introducing two action substitution operations:

$$\begin{aligned}\{\text{water} \leftarrow \text{coffee}\}\{\text{cola} \leftarrow \text{tea}\}(\text{tea}.\overline{\text{pick}}.\text{CTM} + \text{coffee}.\overline{\text{pick}}.\text{CTM})\\ \{\text{cola} \leftarrow \text{tea}\}(\text{tea}.\overline{\text{pick}}.\text{CTM} + \text{water}.\overline{\text{pick}}.\text{CTM})\\ (\text{cola}.\overline{\text{pick}}.\text{CTM} + \text{water}.\overline{\text{pick}}.\text{CTM})\end{aligned}$$

More formally, considering for each process variable A a defining equation $A(x_1, \ldots, x_n) = P$ where the name variables $x_1, \ldots, x_n$ occur (bound) in P, the syntax of CCS process can be defined as:

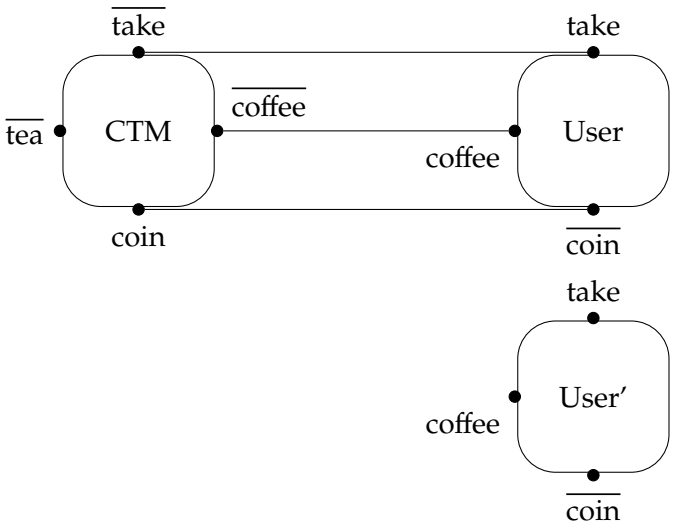

Figure 2.5: Interface of the (SelfishUser | User′) system.

$P, Q, R ::=$	Processes
$\mathbf{0}$	Empty process
$\mid A\langle a_1, \ldots, a_n \rangle$	process definition
$\mid \alpha.P$	action prefix
$\mid (\mathbf{new}\ a)P$	action hiding
$\mid P \mid Q$	parallel composition
$\mid P + Q$	(non-deterministic) choice

2.3.3 Labelled Transition Systems

One way to visualize and reason about the possible behaviour of a CCS process is through the representation in a Labelled Transition System (LTS). In this model, the processes are represented by the vertices of an edge-labelled graph and the transition relation. When an action is performed, the change in state is represented as moving along an edge, labelled by the action name, that goes out of that state.

Given a set of CCS defining equations $\mathcal{P}$ specifying a system, the transition relation of the system is defined by a set of triples:

$$\{\xrightarrow{a} \in \mathcal{P} \times \mathcal{P} \mid \ a \in \mathsf{Act}\}$$

Continuing with the previous example of a vending machine, we can define a CCS process model of the machine:

$$\mathrm{CTM} \stackrel{\mathrm{def}}{=} \mathrm{coin}.(\mathrm{coffee}.\overline{\mathrm{pick}}.\mathrm{CTM} + \mathrm{tea}.\overline{\mathrm{pick}}.\mathrm{CTM})$$

The LTS of the vending machine is shown in the Figure 2.6. The LTS provides a great view into how the system can behave. In other words, it shows the *potential* behaviour

of the machine. This is because, for the system to evolve, other processes are required in order to have communication with the vending machine. Without interaction from another process, the vending machine cannot have progress.

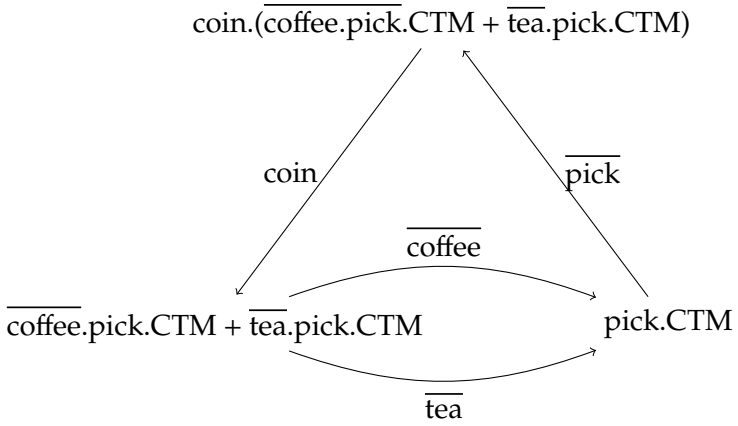

Figure 2.6: LTS of the CTM process.

In order to model the *concrete* behaviour that the system will exbibit it is necessary to introduce another process to the system, a process representing a user. For this example we will introduce a user that repeatedly orders coffee from the machine. The user is modelled as the following CCS process.

$$\text{User} \stackrel{\text{def}}{=} \overline{\text{coin}}.\text{coffee}.\text{take}.\text{User}$$

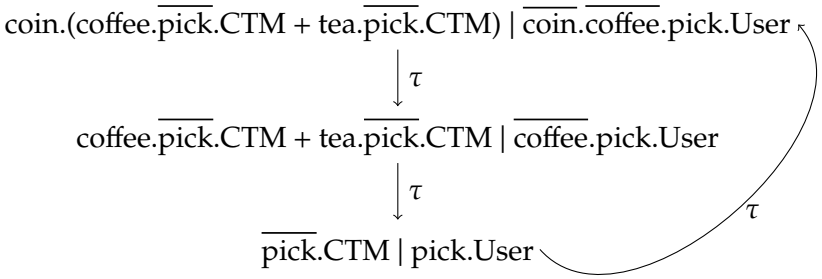

Figure 2.7: LTS of the CTM | User process.

In Figure 2.7 we can see the evolution of the system. The transitions with label τ represent the evolution of the system via an internal communication. This communication is a handshake between two processes and leads to a state transition that is not observable from the perspective of the environment. For example, in the first state transition, the CTM and User process perform an input and output action, respectively, over the channel coin. This represents the user placing a coin in the coin slot of the machine. While for each of the two processes this communication directly changes their state, for other possible

parallel processes in the system the communication is irrelevant and transparent w.r.t. their behaviour.

2.3.4 Actions of a Process

The actions of a process $\mathsf{Act}(P) \subseteq \mathsf{Act}$ is a set inductively defined by the following rules:

$$\begin{aligned}
\mathsf{Act}(A\langle a_1, \ldots, a_n\rangle) &= \{a_1, \ldots, a_n\} \\
\mathsf{Act}(\alpha.P) &= \{a\} \cup \mathsf{Act}(P), \text{ if } \alpha = a \text{ or } \alpha = \overline{a} \\
\mathsf{Act}((\mathbf{new}\, a)P) &= \{a\} \cup \mathsf{Act}(P) \\
\mathsf{Act}(P \mid Q) &= \mathsf{Act}(P) \cup \mathsf{Act}(Q) \\
\mathsf{Act}(P + Q) &= \mathsf{Act}(P) \cup \mathsf{Act}(Q)
\end{aligned}$$

2.3.5 Free and Bound Actions

The set of free actions of a process (i.e., the actions that are visible to all processes), is the set

$$\mathrm{fn}(P) = \mathsf{Act}(P) \setminus \mathrm{bn}(P)$$

The set of bound actions of a process (i.e., only visible to itself, within the scope of the action hiding), is the set $\mathrm{bn}(P) \subseteq \mathsf{Act}$, inductively defined by the rules:

$$\begin{aligned}
\mathrm{bn}(A\langle a_1, \ldots, a_n\rangle) &= \emptyset \\
\mathrm{bn}(\alpha.P) &= \mathrm{bn}(P) \\
\mathrm{bn}((\mathbf{new}\, a)P) &= \{a\} \cup \mathrm{bn}(P) \\
\mathrm{bn}(P \mid Q) &= \mathrm{bn}(P) \cup \mathrm{bn}(Q) \\
\mathrm{bn}(P + Q) &= \mathrm{bn}(P) \cup \mathrm{bn}(Q)
\end{aligned}$$

2.3.6 Structural Operational Semantics

Figure 2.8 presents the structural operational semantics of CCS. This set of rules rigorously define the behaviour of a program in terms of the behaviour of its parts. For example, the rule [Pre], has it has no premises (no transition above the horizontal line), the process can always afford the transition $\alpha.P \xrightarrow{\alpha} P$. The rule [L-Par], on the other hand, contains a premise. In this case, the rule indicates that the left side of the parallel composition can evolve form P into P' only if we can prove that P can afford the transition $P \xrightarrow{\alpha} P'$. Rule [L-Par] models the symmetric case. Rule [Alpha] deals with syntactically equal process with alpha conversion. Rules [Def] and [Res] model process definition and action hiding, respectively. Finally, [L-Sync] and [R-Sync] allows for processes to synchronize on dual actions, namely, a receive with send, or send with receive, respectively.

To illustrate the use of the SOS rules, In Figure 2.9 we present the proofs for every τ transition the process CTM | User can afford.

$$\frac{P_A\{\vec{a} \leftarrow \vec{b}\} \xrightarrow{\alpha} P'}{A\langle \vec{b} \rangle \xrightarrow{\alpha} P'} \; A(\vec{a}) \stackrel{\text{def}}{=} P_A \; [\textsc{Def}] \qquad \frac{}{\alpha.P \xrightarrow{\alpha} P} \; [\textsc{Pre}]$$

$$\frac{P \xrightarrow{\alpha} P'}{(\textsf{new}\, a)P \xrightarrow{\alpha} (\textsf{new}\, a)P'} \; \alpha \notin \{a, \overline{a}\} \; [\textsc{Res}] \qquad \frac{P \xrightarrow{\alpha} P' \quad P =_\alpha Q}{Q \xrightarrow{\alpha} P'} \; [\textsc{Alpha}]$$

$$\frac{P \xrightarrow{\alpha} P'}{(P \mid Q) \xrightarrow{\alpha} (P' \mid Q)} \; [\text{L-}\textsc{Par}] \qquad \frac{Q \xrightarrow{\alpha} Q'}{(P \mid Q) \xrightarrow{\alpha} (P \mid Q')} \; [\text{R-}\textsc{Par}]$$

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \; [\text{L-}\textsc{Sum}] \qquad \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'} \; [\text{R-}\textsc{Sum}]$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{(P \mid Q) \xrightarrow{\tau} (P' \mid Q')} \; [\text{L-}\textsc{Sync}] \qquad \frac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{(P \mid Q) \xrightarrow{\tau} (P' \mid Q')} \; [\text{R-}\textsc{Sync}]$$

Figure 2.8: Structural Operational Semantics of CCS.

$$\dfrac{\dfrac{}{\text{coin.(coffee.}\overline{\text{pick}}\text{.CTM + tea.}\overline{\text{pick}}\text{.CTM)} \xrightarrow{\text{coin}} \text{coffee.}\overline{\text{pick}}\text{.CTM + tea.}\overline{\text{pick}}\text{.CTM}}\,[\textsc{Pre}] \quad \dfrac{}{\overline{\text{coin}}.\overline{\text{coffee}}\text{.pick.User} \xrightarrow{\overline{\text{coin}}} \overline{\text{coffee}}\text{.pick.User}}\,[\textsc{Pre}]}{\text{coin.(coffee.}\overline{\text{pick}}\text{.CTM + tea.}\overline{\text{pick}}\text{.CTM)} \mid \overline{\text{coin}}.\overline{\text{coffee}}\text{.pick.User} \xrightarrow{\tau} \text{coffee.}\overline{\text{pick}}\text{.CTM + tea.}\overline{\text{pick}}\text{.CTM} \mid \overline{\text{coffee}}\text{.pick.User}}\,[\text{L-}\textsc{Sync}]$$

(a)

$$\dfrac{\dfrac{\dfrac{}{\text{coffee.}\overline{\text{pick}}\text{.CTM} \xrightarrow{\text{coffee}} \overline{\text{pick}}\text{.CTM}}\,[\textsc{Pre}]}{\text{coffee.}\overline{\text{pick}}\text{.CTM + tea.}\overline{\text{pick}}\text{.CTM} \xrightarrow{\text{coffee}} \overline{\text{pick}}\text{.CTM}}\,[\text{L-}\textsc{Sum}] \quad \dfrac{}{\overline{\text{coffee}}\text{.pick.User} \xrightarrow{\overline{\text{coffee}}} \text{pick.User}}\,[\textsc{Pre}]}{\text{coffee.}\overline{\text{pick}}\text{.CTM + tea.}\overline{\text{pick}}\text{.CTM} \mid \overline{\text{coffee}}\text{.pick.User} \xrightarrow{\tau} \overline{\text{pick}}\text{.CTM} \mid \text{pick.User}}\,[\text{L-}\textsc{Sync}]$$

(b)

$$\dfrac{\dfrac{\overline{\text{pick}}\text{.coin.(coffee.}\overline{\text{pick}}\text{.CTM + tea.}\overline{\text{pick}}\text{.CTM)} \xrightarrow{\overline{\text{pick}}} \text{coin.(coffee.}\overline{\text{pick}}\text{.CTM + tea.}\overline{\text{pick}}\text{.CTM)}}{\overline{\text{pick}}\text{.CTM} \xrightarrow{\overline{\text{pick}}} \text{coin.(coffee.}\overline{\text{pick}}\text{.CTM + tea.}\overline{\text{pick}}\text{.CTM)}}\,[\textsc{Def}] \quad \dfrac{\text{pick.}\overline{\text{coin}}.\overline{\text{coffee}}\text{.pick.User} \xrightarrow{\text{pick}} \overline{\text{coin}}.\overline{\text{coffee}}\text{.pick.User}}{\text{pick.User} \xrightarrow{\text{pick}} \overline{\text{coin}}.\overline{\text{coffee}}\text{.pick.User}}\,[\textsc{Def}]}{\overline{\text{pick}}\text{.CTM | pick.User} \xrightarrow{\tau} \text{coin.(coffee.}\overline{\text{pick}}\text{.CTM + tea.}\overline{\text{pick}}\text{.CTM)} \mid \overline{\text{coin}}.\overline{\text{coffee}}\text{.pick.User}}\,[\text{R-}\textsc{Sync}]$$

(c)

Figure 2.9: Example proofs of transitions in Figure 2.6.

3

Related Work

In this chapter, we start by providing an overview of previous studies of concurrency bugs and the usage of concurrency primitives. The first analysed bugs in C codebases, while the other three focused on Go projects. From these studies we can glean some important metrics in how concurrency is used in Go projects, and the characteristics of concurrency bugs that projects have. Two of the studies of Go concurrency bugs also provide the bug dataset, from which we will reference in order to build our collection of common Go code patterns with deadlock bugs.

Subsequently, we will explore some existing solutions for deadlock detection in Go programs.

3.1 Studies of Concurrency Bugs

Lu et al. [18] performed a comprehensive study where they analysed and categorized concurrency bugs and their respective fixes of 4 large-scale open-source applications. Of the 105 concurrency bugs analysed, 31 (30%) were deadlock bugs. The authors found that almost all (96%) of the concurrency bugs are guaranteed to manifest if the partial order between two threads is enforced, indicating that pairwise testing of concurrent threads can expose the vast majority of concurrency bugs. Of the 31 deadlock bugs the authors measured, 19 were fixed by removing the acquisition of the resource. This strategy however can lead to other non-blocking concurrency bugs. In some of the bug reports, programmers even explicitly said that would be the case and applied the patch, hoping that the probability of the non-deadlock bug to occur would be small enough. The approach, however, although useful, is clearly not sound.

Dilley and Lange [6] conducted an empirical study of the usage of message-passing concurrency in Go programs. The authors developed a tool-chain to automatically parse and collect metrics on the usage of various message-passing concurrency primitives in Go programs. They applied the tool-chain to 865 top-stared open-source Go projects from GitHub. The authors showed that 76% of the projects feature channel creation and 82% contain at least one goroutine creation. With regard to channels, almost all (94%) channel

creations have a capacity can be determined statically, with synchronous channels being the most common channel type with 61%. Of the asynchronous channels (33%), 75% have a capacity of at most 5. The study also discovered that projects generally structure the concurrent related code in a limited part of the codebase, with only 20% of the files containing concurrency related features. Overall this study showed that, in practice, Go projects make frequent use of concurrency primitives and that static analysis may be performed in a more modular way, analysing only the concurrent code instead of the program as a whole.

Tu et al. [29] conducted an empirical study on Go concurrency bugs from 6 widely used Go applications. In total the authors analysed 171 concurrency bugs. They categorized the bugs across two orthogonal dimensions: whether the bug was caused by misuse of shared-memory or message-passing, and whether the bug caused any goroutine to block indefinitely (blocking bugs) or not (non-blocking bugs). Overall they concluded that it is as easy to make concurrency bugs with shared-memory as with message-passing, with 58% of the blocking bugs being caused by message passing. The authors also evaluated and compared the execution of two implementations of gRPC, one written in C++ and the second one in Go. With the same workload, the Go implementation created multiple times more goroutines than threads in the C++ version. Together with a lower goroutine average execution time relative to threads, it can be observed that goroutines are shorter but executed more frequently. This is in line with the Go's design goals of making goroutines lightweight and easy to use, and thus incentivizing their usage in more projects. The study also found a high correlation between bug causes and the strategies used to fix them, that together with overall simple fixes suggests promising results for automated tools to fix blocking bugs in Go.

Yuan et al. [30] developed a benchmark suite of Go concurrency bugs found in 9 real world projects. The authors compiled two test suites of bugs: GoREAL and GoKER. For each of the 87 bugs in GoREAL the authors created a docker image with the buggy version of the software and a script to trigger the concurrency bug. GoKER contains a collection of 103 bug kernels. Each bug kernel is extracted and simplified from a real world bug, while preserving its bug-inducing properties. As part of the suite, the authors also introduced a taxonomy of Go concurrency bugs, classifying each bug according to their root cause. Together these two test suites provide two very different strategies to evaluate bug detection tools. The much smaller but still real-world based bugs in GoKER can be most useful in evaluating the coverage and efficacy of bug detection tools. While the GoREAL test suit is more appropriate to evaluate how the tools scale with large projects.

3.2 Prior Research and Tools

3.2.1 Built-in in Go runtime

Go's runtime includes a very basic but sound (i.e. does not give false positives) global deadlock detector [24]. The detector is implemented as part of the goroutine scheduler and as such is always enabled. The goroutine scheduler frequently checks if *all* goroutines are unable to make progress and if so the Go runtime panics, alerting the user that a global deadlock as occurred, and terminates the execution of the program. This is the main limitation of the built-in deadlock detector: it is only able to catch global deadlocks, where *all* goroutines are blocked.

As is incentivized by Go's philosophy, goroutines are very common in Go applications. Their very light overhead and ease of use leads to Go applications creating more Goroutines than comparable constructs in other programming languages [29]. All it takes is a single background goroutine that is not blocked to effectively render the deadlock detector inoperative. Of the 21 blocking bugs tested by Tu et al. [29], only 2 were caught by the built-in detector. This significantly reduces the effectiveness of the detector in real world applications. The detector only provides a stack trace of the blocked goroutines, leaving the resolution part to the developer.

3.2.2 GoAT

GoAT (Go Analysis and Testing), developed by Taheri and Gopalakrishnan [27], is a tool that exercises a mix of dynamic and static analysis to perform deadlock detection (Figure 3.1).

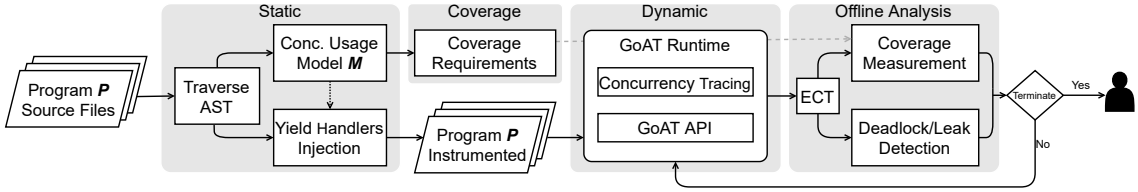

Figure 3.1: GoAT overview.

GoAT starts by traversing the abstract syntax tree (AST) of the input program extracting a concurrency usage model. The concurrency usage model is a table containing the location of every concurrency operation used in the program. For every different concurrency operation the authors defined a set of possible behaviours the operation can exhibit. For example, Go's channel send operation can be blocked, unblocking or NOP. The operation can be blocked when there is no corresponding receive operation ready to execute, can be unblocking if a receive operation is blocked and as such the send operation will unblock the receiver, or with buffered channels, the send operation can neither be blocked nor unblocking. With the collection of possible behaviours, the tool generates a set of coverage requirements for each concurrency operation in the program. In the Listing 5 and Table 3.1

we can see an example of the coverage requirements generated for a Go program. The GoAT tool will detect which coverage requirements have been met. These will serve as indicators to measure the quality and progress of the schedule-space exploration.

```
1  func main() {
2      container := &Container{stop:
   ↪  make(chan struct{})}
3      go Monitor(container)
4      go StatusChange(container)
5  }
6  func Monitor(cnt *Container) {
7      for {
8          select {
9          case <- cnt.stop:
10             return
11         default:
12             cnt.Lock()
13             cnt.Unlock()
14         }
15     }
16 }
17 func StatusChange(cnt *Container) {
18     cnt.Lock()
19     defer cnt.Unlock()
20     cnt.stop <- struct{}{}
21 }
```

Listing 5: Example of Go code with concurrent operations.

Concurrent operations		Coverage Requirements
Line	**Kind**	
3	go	covered
4	go	covered
8	select	c-recv-blocked
		c-recv-unblocking
12	lock	blocked
		blocking
13	unlock	unblocking
		no_op
18	lock	blocked
		blocking
19	unlock	unblocking
		no_op
20	send	blocking
		unblocking
		no_op

Table 3.1: Coverage requirements of the Go code in Listing 5.

Since in practice, during execution, programs tend to exercise only a small portion of the possible interleavings, GoAT inserts code that calls an yield handler before each concurrency primitive usage. The yield handler randomly chooses to call the `runtime.GoSched()` function up to a specified number of times. The `runtime.GoSched()` function tells the Go runtime scheduler to yield the execution of the goroutine, pushing the goroutine to the back of the global runnable goroutine queue, thus forcing other runnable goroutines to execute.

The Go runtime features an execution tracer that provides dynamic tracing of the programs execution, logging events during program execution, such as goroutine creation/blocking/unblocking and system call related-events. The resulting trace can then be used to analyse offline various aspects of the program's execution, such as goroutine latency and blocking behaviour. While the events are compressive enough to analyse many aspects of a program's execution like goroutine latency and blocking behaviour, they still lack many events to allow in-depth analysis of the program's concurrency model. The authors extended the Go's existing execution tracer package with 14 new events. These new events log the usage and behaviour of Channels, Mutexes, WaitGroups, Conditional variables, the select statement and scheduler.

With the execution concurrent trace (ECT), GoAT can analyse the program's execution and determine if any deadlocks have occurred. If a deadlock has occurred the tool generates visualizations of the execution interleaving and the goroutines tree. This information can then help the user fix the bug. The program can be repeatedly executed and analysed until a bug found or until the coverage metrics reach a certain threshold.

In the authors' evaluation it was very effective at detecting deadlock bugs, being able to detect all deadlock bugs in the GoKER benchmark. However, as it is relying on the analysis of the program's execution, it can only find deadlocks that occur in the execution paths tested. Without an exhaustive testing where every execution path is evaluated, false negatives can occur. Once again, the tool only focuses on detecting deadlocks and does not suggest any resolution to the user.

3.2.3 GCatch & GFix

Liu et al. [16] developed, to the best of our knowledge, the only tool capable of automatically resolving deadlock bugs in Go.

The work introduces two tools, GCatch and GFix. Both tools perform static analysis on the Go code, operating over the single static assignment (SSA) form, with the code transformations of GFix being achieved over the abstract syntax tree (AST).

GCatch execution is composed of two main components:

Disentangling. By disentangling, the authors refer to the process of extracting the code that interacts with a particular channel. For each channel the disentangling algorithm returns a scope and a set of related primitives that can be analysed separately. Since in most cases, the channels are only accessed by a small portion of a codebase, this process can significantly speed up the analysis. The authors measured that, on average, disabling disentangling leads to an 115X slowdown in the time necessary to analyse a codebase.

Constraint system. For each channel in the Go program, after disentanglement, GCatch gathers all possible execution path combinations within the scope returned by the disentanglement policy. When generating the path combinations, GCatch includes a set of heuristics to filter out paths that are infeasible to generate deadlocks. For example, when a function call is encountered that does not interact with any primitive in the set of related primitives, then the function call is simply ignored. In the case of loops, in order to better model the possible interactions between successive iterations, GCatch unrolls the loop up to two times, if the number of iterations cannot be determined statically.

For every generated path combination, GCatch calculates suspicious groups, with each group containing synchronization primitives that together can potentially block indefinitely.

GCatch will for each suspicious group, generate constraints modelling various aspects of the program:

- Constraints denoting the operation order imposed by the execution paths and goroutine creation,
- Constraints modelling buffered and unbuffered channels and their operations: send, receive, select statement and close.
- Constraints requiring each group operation to be unable to make progress.

In the end, GCatch invokes the Z3 [22] constraint solver with the conjunction of all the previous constraints. If Z3 is able to find a solution it means that the program might have a deadlock.

GCatch also includes 5 other traditional detectors capable of detecting traditional concurrency bugs such as locks without unlocks, double locks and the acquisition of two locks in conflicting orders. GCatch also models mutexes by converting them to buffered channels of buffer size one, with the locking being represented as a send operation and unlocking as a receive operation.

The second tool, GFix, is designed to generate minimal patches that resolve the deadlock bugs. GFix takes as input the bug found by the constraint system in GCatch. The scope of bugs that GFix can fix is as follows (Listing 6):

- Two goroutines, *Go-A* and *Go-B*, access a local channel *c*.
- *Go-B* is a child goroutine of *Go-A*.
- When the bug is triggered, *Go-A* fails to execute operation *o1* on channel *c*, causing *Go-B* to be blocked at operation *o2*.

```
1  func main() { // Go-A
2      c := make(chan bool)
3      go func() { // Go-B
4          ...
5          c <- true // o2
6      }()
7      select {
8      case <- c: // o1
9          ...
10     case /* other case*/ :
11         return
12     }
13 }
```

Listing 6: General pattern of bugs that GFix tries to resolve.

When a bug is found that satisfies the previous requirements, GFix tries to apply the following 3 strategies:

Strategy I: Increases the channel buffer size, preventing the *o2* send operation from blocking.

For example, in Listing 7 before the patch, if the select (line 8) receives a message from the channel `ctx.Done()` before the channel `outDone`, the Exec function will return and the child goroutine will block on line 6. In that case, the parent goroutine will never receive the value from the send operation at line 6 and, since the channel is synchronous the send operation will block.

By increasing the channel's buffer from zero to one, the send operation will no longer block as the buffer will be able to store the value. The select will behave the same. The channel will be empty and as such the receive operation (line 11) will block until a message is sent.

```
 func Exec(ctx context.Context) error {
-    outDone := make(chan error)
+    outDone := make(chan error, 1)
     go func() {
         ...
         outDone <- err // block
     }()
     select {
     case err := <-outDone:
         return err
     case <-ctx.Done():
     }
     return
 }
```

Listing 7: Example of a patch generated by GFix - Strategy I.

Strategy II: Fixes the cases when the goroutine *Go-A* returns (due to return or panic) without executing *o1* by using the **defer** keyword to defer the *o1* operation. Go's runtime automatically executes all deferred statements after the function returns or panics. With this patch, the *Go-A* is guaranteed to execute *o1* in all cases, and as such *o2* will never block.

In the Listing 8, it is possible for the function `TestRWDialer()` to exit without receiving the message from the child goroutine (by panicking at line 12). In this example, GFix removes the existing receive operation (line 14), and adds a **defer** statement after the channel creation, enforcing the execution of the receive operation.

Strategy III: This final strategy is the most invasive of the three. The patch introduces a new channel `stop`. This channel is used to notify goroutine *Go-B* that goroutine

```
 func TestRWDialer() {
     stop := make(chan struct{})
+    defer funct() {
+        stop <- struct{}{}
+    }()
     go func(stop struct{}{}) {
         ...
         outDone <- err // block
     }(stop)
     conn, err := d.Dial(...)
     if err != nil {
         panic("dial error")
     }
-    stop <- struct{}{}
 }
```

Listing 8: Example of a patch generated by GFix - Strategy II.

Go-A has failed. So that if *Go-B* is blocked and goroutine *Go-A* fails, *Go-B* can receive a message from the stop `stop` channel and unblock.

Listing 9 shows a function that dispatches a goroutine to read an input, line by line, sending each line through a channel to the parent goroutine. The fault here is when the `Interactive()` function receives an abort, the goroutine will continue to read lines, but will block trying to send them to the parent goroutine. The fix adds a new channel that is used to unblock the child goroutine and terminate it.

In an effort to reduce invalid patches that change the code semantics in undesirable ways, besides the requirements listed earlier, each strategy has it's considerable set of requirements that the code must meet in order for the patch to be applied. The most notable of them is the potential side-effects in the instructions after *o2*. Since the intent of GFix is to remove the blockage in *o2*, any instruction after that operation must not change the original semantics of the program. To that end, GFix performs an inter-procedural analysis on the instructions after *o2*, checking for library function calls, concurrency operations, or updates to variables defined outside *Go-B*.

The authors evaluated the solution on 23 Go open-source projects. The GCatch tool caught 149 true deadlock bugs with 46 false positives. Of the 46 false positives, 20 were due to infeasible paths, with 11 of these being due to the loop unrolling. Of the remaining, 17 and 14 were due limitations of the alias and call-graph analysis respectively. In order to evaluate coverage, the authors executed GCatch over a set of 49 deadlock bugs from [29]. The tool detected 33 bugs, leading to a false negative rate of 33%.

From 147 deadlock bugs, GFix correctly generated 124 patches that fixed the bugs. The authors assessed the corrects of each patch manually, checking if each patch fixed the bug

```
 func Interactive() {
     scheduler = make(chan string)
+    stop = make(chan struct{})
+    defer close(stop)
     go func() {
         for {
             line, err := Input()
             if err != nil {
                 close(scheduler)
                 return
             }
-            scheduler <- line
+            select {
+            case scheduler <- line:
+            case <- stop: return
+            }
         }
     }()
     for {
         select {
         case <- abort: return
         case _, ok := scheduler:
             if !ok {return}
         }
     }
 }
```

Listing 9: Example of a patch generated by GFix - Strategy III.

via code inspection and injection of random-duration sleeps around channel operations, and that the patch did not change the original program semantics.

While effective at resolving the bugs detected by GCatch, we find the approach taken somewhat ad-hoc. In order to reduce invalid patches the strategies taken are very specific to certain patterns of bugs, providing little generalization to other, more complex, patterns. The inclusion of more general strategies would likely be very challenging, specially with the requirement of producing correct patches that can be applied without modification from the developers. This results in a solution that can still leave many deadlock bugs unresolved.

3.2.4 MiGo/Gong

Lange et al. present a static verification framework to checks for liveness and channel-safety [15]. In this work, liveness denotes the ability of communication actions to always eventually fire, *i.e.*, the absence of deadlocks, while channel-safety regards the misuse

of closed channels, where closing an already closed channel or sending a message on a closed channel triggers a run-time panic.

To accomplish the aforementioned analysis, the authors introduce a new process calculus [2], dubbed MiGo (Mini-Go), a core language allowing to write message-passing Go (like) programs — the syntax closely mimics Go's communication-based constructs, featuring channels, goroutines, and the `select` statement. The paper presents a collection of illustrating examples.

The authors follow by introducing a typing system for the calculus that actually performs a translation of value-passing processes into processes that use channel names just to synchronize (called types). The translation basically erases message payloads and turn `if` statements into non-deterministic choices.

Then, the authors debut the concept of fencing, which imposes a syntactic restriction on channel usage: when a type is fenced, the program is guaranteed to be made up of finitely many different communication patterns, that may be repeated infinitely many times. This restriction is imposed by checking that the type does not contain any parallel composition, or that any recursive call over the type uses strictly fewer (non newly created) names. Next, the authors introduce an operational semantics for types, via a symbolic labelled transition system, which, for the types that are fenced, is finite state. Therefore, automated analyses (of namely, liveness and channel safety) are decidable. The paper presents a bounded model checking verification procedure of liveness and of channel safety.

Finally, the authors discuss liveness in respect to three classes of programs. For those that have a terminating path, or that do not contain infinitely occurring conditional branches, if the program is typable by a live type, then the program is itself also live. In the third class of programs, those that run infinitely and that contain recursive variables in conditional branches, extra care must be taken to prevent a mismatch between the program and type behaviours. To this purpose, the authors define a subclass, dubbed alternating conditional. In this subclass, if the program is well-typed with some live type, then the program must live.

This solution does not propose any resolution for the deadlocks it finds.

3.3 Francalanza et al.'s Approach

Francalanza et al. devised a compositional algorithm to statically detect deadlocked processes, and, in turn, unblock them, using one of two proposed strategies that refactor the original process [8]. This approach operates over a simpler version of CCS, excluding recursion, choice and channel scoping, that also enforces a linear use of channel names, meaning that an input or output action over a channel can only occur at most once [13]. Furthermore, the approach focuses only on the types of deadlocks we defined earlier as incorrect communication ordering.

The intuition behind the static analysis performed to find deadlocked processes is based on the construction of layers. Each layer is an environment that provides a partial map from channel names to permissions. Permissions denote the communication capability over a channel: whether an input, output, or both operations can be performed. With the layers of environments, an approximation of the prefixing found in the process is generated. For example, from the process $a.b.\mathbf{0} \parallel \bar{b}.\bar{c}.\mathbf{0} \parallel c.\bar{a}.\mathbf{0}$ the following list of layered environments is constructed:

$$\underbrace{(a :\downarrow, b :\uparrow, c :\downarrow)}_{\Gamma_1} ; \underbrace{(a :\uparrow, b :\downarrow, c :\uparrow)}_{\Gamma_2} ; \epsilon$$

While building the layers, if one is found where no synchronization can occur, and the top environment in the layer is top-complete, a deadlock verdict is emitted. An environment is top-complete if the inverse of every permission in the top environment is found in a sub-layer. This indicates that, since the processes are linear, if the co-actions of the actions found in the top layer are found in sub-layers, and the top environment can not synchronize, the actions in the top layer will never synchronize, and thus, a deadlock verdict can be emitted.

In order to unblock deadlocked processes, the resolution strategies *refactor* the original process into a form where the deadlock does not take place. The two deadlock resolution algorithms take as input the CCS expression of the program, and a list of the blocked actions when a deadlock is detected. Later we will present the formal definition for each algorithm. The first algorithm, moves the blocked actions into a parallel composition, allowing the resulting process to circumvent the problematic action. The second algorithm, takes the same approach when resolving a problematic output. In the case of a blocked input, the input is not parallelized, instead the corresponding output is placed in a parallel composition, located at the same level as the input.

Deadlocks of the missing communication type are presently a case where the resolution step cannot be reliably automated. Since these types of deadlocks are inherently unbalanced, the resolution requires either the addition of the corresponding action, or the removal of the problematic one. Neither can be performed automatically in a safe manner without changing the program semantics in potentially undesirable ways.

Unfortunately, this approach's CCS variant is very limited in terms of expressiveness. The lack of choice, recursion, and the restriction to linear processes greatly inhibits its utility in real world programs.

3.4 Almeida's Approach

Almeida, as part of their master thesis, developed a tool capable of automatically detecting and resolving deadlocks [1]. This solution is based on the approach by Francalanza et al., extending and redesigning the algorithms in order to add support for more feature-full CCS variant. This is grounded on the objective of analysing programs written in Go.

Almeida added support for choice and non-linear processes. In order to achieve this, the deadlock detection algorithm was considerably redesigned. With the original approach, when a non-linear processes is given, the algorithm is too eager to attribute a verdict, emitting a deadlock verdict even when synchronization with the deadlocked action is still possible. To overcome this limitation the revamped solution, implemented in OCaml, essentially simulates an execution of the Go program, reducing the CCS process as far as possible. When no reduction step is possible, a deadlock is detected.

Almeida's solution has some major flaws, both in terms of time complexity and expressiveness. When reducing the process, the tool checks if synchronization between two actions is possible, only returning the reduction of the first synchronization it finds. For example: in the CCS process $\bar{a}.\mathbf{0} \parallel a.\bar{a}.\mathbf{0} \parallel a.\mathbf{0}$, two possible synchronizations can take place, either $\bar{a}.\mathbf{0}$ with $a.\bar{a}.\mathbf{0}$ or $\bar{a}.\mathbf{0}$ with $a.\mathbf{0}$. During deadlock detection, only the reduction result of first synchronization is returned, and as such, this approach is not guaranteed to find all the possible reductions. In order to correctly model this type of non-determinism behaviour, Almeida's solution generates all the possible permutations of the parallel compositions before the first reduction and after every synchronization step. This technique is very inefficient, as it generates many permutations where the resulting reduction will be the same. The factorial time complexity of this approach limits its usage to programs with a small number of parallel compositions.

The second significant limitation of Almeida's solution regards the semantics of the choice operator. In his solution, the choice operator always behaves as an internal (non-deterministic) choice, where the reduction can always evolve to either branch. While this behaviour can be used to model the common conditional statements, it is not adequate to model Go's `select` statement. In Almeida's solution, the best representation of the Go program in Listing 3.2a, is show in Fig. 3.2b, which models the select statement using an internal choice. With internal choice, the CCS process can always reduce to either branch, acting as if every `case` in the `select` can always proceed. This does not match with the Go semantics, and, as so, an incorrect verdict will be given, claiming that the process has a deadlock when the second branch is chosen.

```go
func main() {
    a := make(chan struct{})
    b := make(chan struct{})

    go func() {
        a <- struct{}{}
    }()

    select {
    case <-a:
        return
    case <-b:
        b <- struct{}{}
    }
}
```

(a) The Go code.

$$\bar{a}.\mathbf{0} \parallel (a.\mathbf{0} + b.\bar{b}.\mathbf{0})$$

(b) CCS representation with internal choice.

Figure 3.2: Example of a deadlock-free Go program with a select statement.

4

Characterization of Some Real-world Deadlocks

We have analysed and compiled a collection of deadlocks[1], from the test suites of two papers [29, 30]. These two papers studied a large collection of bugs in Go projects, with the goal of analysing and categorizing all types of concurrency bugs. While these two papers provide great insight on the landscape of concurrency bugs in Go, for our purposes, we found them lacking when focusing on deadlock bugs over the message passing concurrency model. And so, we decided to perform a study focusing on these types of concurrency bugs.

4.1 Methodology

The deadlock bugs were present in high-profile open source projects, written in the Go programming language. For 16 deadlocks we categorized and grouped the errors over three major aspects. In first group we determined which Go constructs and concurrency features were involved in the deadlock. This group can be of use to determine the most common features of the language that a solution for deadlock detection and resolution should support. The second group indicates the cause of the deadlock, while the third group the types of changes on the code employed by the developers to resolve each deadlock. This can be useful to understand which changes are more common. For each deadlock we also extracted a code snippet that retains just the bug inducing part of the code that can be then used to evaluate existing deadlock detection and resolution tools.

4.2 Results

In Table 4.1 we present the first group, with the features and constructs of the Go language that the faulty code makes use of and are relevant for the deadlock. Each line corresponds to a bug from the collection, with the leftmost column indicating the project from where

[1] https://github.com/JorgeGCoelho/go-deadlock-bug-collection

the bug was sourced, and the identifier for the GitHub pull-request where the bug was disclosed and fixed. Column 1 signals the use of the `select` statement. Usually, a default branch is missing and thus, if all branches are blocked the `select` does not progress. Column 2 indicates that the faulty code makes use of the `close` operation over a channel. Not closing a channel after sending on it in an iteration also causes a deadlock (the receiver must be told no more values will arrive). Column 3 shows if the code features channel passing, that is, when a channel is passed as a value via another channel. Blocks in this case happen in the received channel. Finally, Columns 4 and 5 indicate whether synchronous or asynchronous channels were used. They block when at full capacity.

From Table 4.1 we can make some noteworthy observations: In this sample of errors, 80% of the deadlocks involve at least one synchronous channels, and only in one case the faulty code made use of channel passing.

Table 4.1: Go constructs involved in the deadlock.

Column	1	2	3	4	5
Bug Id	**Select**	**Closed channel**	**Channel passing**	**Synch channel**	**Asynch channel**
cockroachdb#13197		X		X	
cockroachdb#13755		X		X	
cockroachdb#18101	X			X	
cockroachdb#24808					X
cockroachdb#25456		X		X	
etcd#6857		X	X	X	
grpc-go#1275				X	
grpc-go#1424		X		X	
grpc-go#490		X			
kubernetes#25331					X
kubernetes#35672		X		X	
kubernetes#5316	X			X	
moby#21233				X	
moby#33293				X	
moby#33781	X	X		X	
moby#4395				X	

In Table 4.2 we present the two remaining groups. In the second group we present a classification of the cause of the deadlock, indicating if the cause of the error is due to a missing receive, send, or close operation over a channel. In the third group we studied and classified the changes that the developers applied to the buggy code in order to resolve the deadlock. In this group we identified 5 deadlock resolution strategies. The first operates by making asynchronous a previously synchronous channel. The second proposes the addition of a select statement or a new case in an existing select statement. This is often done to allow the deadlocked goroutine to unblock when it receives a message from another channel, or when the channel is closed. The next strategy involves adding

a communication operation or using the `defer` statement to introduce a communication operation in a code path that should perform that communication. Finally, the last strategy suggests the creation of a new channel.

From Table 4.2 we gather that, regarding the deadlock cause, very few cases were caused by a missing send. With respect to the fixing strategy, introducing a select statement or adding a new case to an existing statement are the most used strategies. Adding a new channel is rarely the action taken to resolve a deadlock. The remaining three strategies (Make channel asynchronous, Add/Defer missing communication, and Close channel), see similar amount of use, less than half of the previous two.

Table 4.2: Causes and fixes for the deadlocks in Go programs.

	Causes			Fix				
Bug Id	**Miss rcv**	**Miss snd**	**Miss channel close**	**Make channel asynch**	**Add select stmt**	**Add/Defer miss commun**	**Close channel**	**New channel**
cockroachdb#13197			X				X	
cockroachdb#13755			X				X	
cockroachdb#18101	X				X			
cockroachdb#24808	X				X			
cockroachdb#25456			X		X			
etcd#6857	X				X			
grpc-go#1275		X				X		
grpc-go#1424			X				X	
grpc-go#490			X		X		X	
kubernetes#25331	X				X			
kubernetes#35672			X		X	X	X	X
kubernetes#5316	X			X				
moby#21233		X						
moby#33293	X			X				
moby#33781	X			X		X		
moby#4395	X			X				

5

GoDDaR — CCS-based Deadlock Detection and Resolution

In this chapter we present our framework for detection and resolution of deadlocks in Go, named GoDDaR[1]. Leveraging on the work of Francalanza et al. [8] and Almeida [1], we developed a solution that, given the source code a Go program, can determine, without directly executing the program, whether it is possible for the program to deadlock. If a potential deadlock is found, GoDDaR will then propose how to refactor the program in an effort to resolve the deadlocks found, so that the refactored program will not deadlock.

5.1 Workflow

The high-level workflow of the tool is shown in Figure 5.1. With this workflow the analysis can be fully automatic, starting with the program's Go source code, and ending with the resolved CCS representation of the Go program[2].

5.1.1 Pipeline

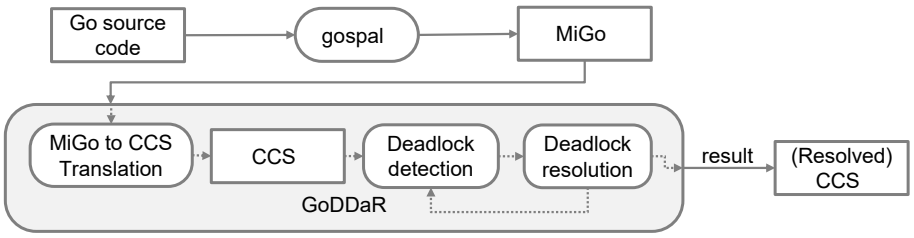

Figure 5.1: GoDDaR pipeline.

The Figure 5.1 shows the sequence of tools and intermediate representations that take part in the pipeline of our tool. In this diagram, the rounded boxes represent tools or a component of a tool, while the square boxes depict a representation of a program.

[1] https://github.com/JorgeGCoelho/GoDDaR

[2] How to use this process to provide feedback on how to patch the original Go code is a matter for future work.

Starting off with the source code in Go, the first step is to generate an intermediate representation in MiGo [15] using the *gospal* [23] tool. Our tool is then able to translate the MiGo representation into our CCS variant. Finally, from the CCS representation, our tool steps through the deadlock detection and one of the deadlock resolution algorithms, resulting in a CCS representation of the original program where the deadlocks might have been resolved[3].

In the rest of this chapter we provide a comprehensive description of each step and associated algorithms present in this pipeline.

5.1.2 Intermediate Representations

As shown in the pipeline, our approach, makes use of two intermediate representations: MiGo and CCS.

MiGo. The MiGo representation is mainly used as a stepping stone to obtain a CCS representation. Compared to the Go programming language, the MiGo abstraction provides a much simpler syntax from which we can translate into CCS. With regards to control flow, the higher level statements, such as `for` loops, `switch`, and `goto` statements, are simplified into `if` statements and recursive function calls. The data flow is mostly omitted, retaining only the assignment of new channels into variables. The communication operations also omit the values being sent or received.

CCS. Our approach is cast within a CCS variant, adapted from Almeida's work to provide a better model on which we can represent the behaviour of Go code. The syntax is shown in Figure 5.2. Our CCS variant differs in two main ways. First, the choice operator has been split into two more specialized operators. In this manner we can represent the two types of non-determinism, internal and external, in a more explicit manner that provides a better mapping to and from Go code. The second major change is the addition of recursion via the form of process replication. This addition is essential to deal with the classes of programs that feature loops and recursion. In sections 5.2.1 and 5.2.2 we describe these additions in more detail.

5.2 Deadlock Detection

The intuition behind the algorithm to detect deadlocks in CCS processes is that, by reducing the process as far as possible, we can find the process states where it cannot reduce any further. In these cases, if the process is not empty (or, in other words, has not terminated), then there are actions that are blocking the process from progressing further, and thus, the process is deadlocked.

[3]

$P, Q ::=$	Processes
$\mathbf{0}$	Empty process
$\mid \alpha.P$	Action prefix
$\mid P \mid Q$	Parallel composition
$\mid P + Q$	Internal choice
$\mid P \,\&\, Q$	External choice
$\mid a^*.P$	Process replication

Figure 5.2: Syntax of our CCS variant.

In our CCS variant we follow a subset of the semantic rules defined in Figure 2.8, rules [Pre], [Alpha], [L-Par], [R-Par], [L-Sync], and [R-Sync]. With these semantics, we define a process P that is deadlocked as the following predicate:

$$\mathsf{dlock}(P) \stackrel{\text{def}}{=} \left(\nexists Q \cdot P \longrightarrow Q\right) \quad \text{and} \quad P \not\equiv \mathbf{0}$$

where $\equiv$ is a structural equivalence relation.

Following the semantic rules, the algorithm explores all possible program states, finding those that satisfy dlock(). For each deadlocked state found, the algorithm collects the problematic actions. The problematic actions are the actions that are blocking the process from progressing, and thus the cause of deadlock. For example, the process $\overline{a} \mid a.(\overline{b}.c \mid \overline{c}.b)$ presents the following deadlock $\overline{b}.c \mid \overline{c}.b$. In this case, the problematic actions are $\overline{b}$ and $\overline{c}$. The union of the problematic actions found in each deadlock is passed to the deadlock resolution stage.

5.2.1 Internal/External Choice

Following the approach taken by Almeida, our solution does not make use of τ. As explored in [5], the interaction between τ and the choice operator + can be somewhat unclear. The choice operator can exbibit a mixture of two forms of non-deterministic behaviour, internal and external non-determinism. For the process $a.P + b.Q$, if it synchronizes on a then the process will act as P, but if it synchronizes over b, the process will act as Q. In this case, the choice operator is modelling external non-determinism, as the behaviour of the process depend on external synchronization. On the other hand, for the process $a.P + a.Q$, the choice is representing internal non-determinism. When synchronizing on a, the process can evolve into either P or Q, demonstrating a case of internal non-determinism, where the resulting behaviour of the choice is not dependent on external communication.

By omitting τ and replacing + with two new combinators, representing internal and external non-determinism, a simpler model can be obtained. In the context of our work, this CCS variant provides a clearer path for the mapping between CCS and Go code. For

example, in Listing 5.3a we have a Go program that makes use of both **if** and **select** statements, and in Figure 5.3b we show the equivalent program in CCS. We can see that, for the **if** statement, it was translated to an internal choice. In our analysis of the **if** statements, we assume that the program can always proceed into either branch, non-deterministically. The choice does not depend on any communication operation and so, an internal choice operator is used to represent this behaviour in CCS. For the **select** statement, it can also be easily be modelled in CCS with an external choice. For each **case**, the communication operation is prefixed to the contents of the **case**'s body.

```
a := make(chan struct{})
b := make(chan struct{})
if (...) {
    R()
} else {
    select {
    case <- a:
        P()
    case <- b:
        Q()
    }
}
```

(a)

$$R + (a.P \,\&\, b.Q)$$

(b)

Figure 5.3: Examples of internal/external choice mapping between Go and CCS.

The semantics of the internal and external choice are given in Figure 5.4.

$$\frac{}{P + Q \longrightarrow P'}\ [\text{L-IntSum}] \qquad \frac{}{P + Q \longrightarrow Q'}\ [\text{R-IntSum}]$$

$$\frac{P \xrightarrow{\alpha} P'}{P \,\&\, Q \xrightarrow{\alpha} P'}\ [\text{L-ExSum}] \qquad \frac{Q \xrightarrow{\alpha} Q'}{P \,\&\, Q \xrightarrow{\alpha} P'}\ [\text{R-ExSum}]$$

Figure 5.4: Operational semantics of internal and external choice.

5.2.2 Recursion

Another crucial addition to the deadlock detection algorithm was the support for recursive processes. This greatly improves the expressiveness of our solution, allowing the analysis of a much larger types of programs. Recursion was added via the introduction of a guarded process replication construction. Syntactically, we represent the construction as $a^*.P$, where a is an input action, and P a process. As show in Figure 5.5, whenever a synchronization occurs with the input action a, the $a^*.P$ is kept and a new P is instanced in parallel.

$$\frac{Q \xrightarrow{\overline{a}} Q'}{(a^*.P \mid Q) \longrightarrow ((a^*.P \mid P) \mid Q')} \text{ [Repl]}$$

Figure 5.5: Operational semantics of process replication.

With this construction recursive processes can be represented. Loops can also be modelled by placing an output action at the end of the process. For example, a simple clock can be represented as $\text{clock}^*.\text{tick}.\text{tock}.\overline{\text{clock}} \mid \overline{\text{clock}}$, evolving as follows:

$$\begin{aligned}
& \text{clock}^*.\text{tick}.\text{tock}.\overline{\text{clock}} \mid \overline{\text{clock}} \\
\longrightarrow\ & \text{clock}^*.\text{tick}.\text{tock}.\overline{\text{clock}} \mid \text{tick}.\text{tock}.\overline{\text{clock}} \\
\xrightarrow{\text{tick}}\ & \text{clock}^*.\text{tick}.\text{tock}.\overline{\text{clock}} \mid \text{tock}.\overline{\text{clock}} \\
\xrightarrow{\text{tock}}\ & \text{clock}^*.\text{tick}.\text{tock}.\overline{\text{clock}} \mid \overline{\text{clock}} \\
\longrightarrow\ & \vdots
\end{aligned}$$

However, with the current approach for the deadlock detection, the analysis might never terminate. For example, the program $a^*.\overline{a} \mid \overline{a}$ will always be able to reduce.

$$\begin{aligned}
& a^*.\overline{a} \mid \overline{a} \\
\longrightarrow\ & a^*.\overline{a} \mid \overline{a} \\
\longrightarrow\ & \vdots
\end{aligned}$$

This will result in an analysis that never terminates. To work around this problem, we store all the process states that have been explored, and whenever a previously explored state reappears, the repeated part of the process is omitted from the analysis. The repeated subprocess is effectively a livelock, and so, in our analysis to find deadlocks, these never terminating subprocess can be ignored.

To illustrate this strategy, we can analyse the program $a^*.\overline{a}.\overline{a} \mid \overline{a}$:

$$\begin{aligned}
& a^*.\overline{a}.\overline{a} \mid \overline{a} && (5.1) \\
\longrightarrow\ & a^*.\overline{a}.\overline{a} \mid \overline{a}.\overline{a} && (5.2) \\
\longrightarrow\ & \underbrace{a^*.\overline{a}.\overline{a} \mid \overline{a}.\overline{a}}_{(5.2)} \mid \overline{a} && (5.3)
\end{aligned}$$

We see that the process will reduce indefinitely, with each iteration instancing a new $\overline{a}.\overline{a}$ in parallel. By the third iteration the process (5.3) will contain a previously explored state (5.2), and so, it will be omitted and the $\overline{a}$ is analysed instead. The analysis of $\overline{a}$ will result in a deadlock verdict.

This example shows a case were for every output actions that synchronizes, two more are instanced. Even though is possible for each individual output action to eventually

synchronize, it is impossible for *every* action to synchronize. The process as a whole is showing progress, but since there will be actions that will never synchronize, we justify the deadlock verdict for these classes of processes. More formally, for recursive processes, we say that a deadlock occurs when an action can not synchronize within a finite number of steps.

Processes featuring only process replication are not considered deadlocks, since, in the context of analysis of Go programs, process replication is only used to define and generate new instances of processes.

5.3 Deadlock Resolution

With the problematic actions returned by the deadlock detection step, the deadlock resolution step uses one of two algorithms to attempt to resolve deadlocks. The intuition behind the two algorithms is that by parallelizing the problematic actions, the program can progress further, potentially synchronizing with the problematic action. In this manner the problematic action will no longer block the process and cause a deadlock.

The strategies are defined functions that recursively iterate through the process, rewriting it in order to resolve the deadlock. Both strategies attempt to resolve the deadlock by parallelization, but what sets them apart is the way the parallelization is performed.

The first resolution algorithm, shown in Figure 5.6, goes over the CCS process and, whenever it encounters a problematic action (as dictated by Γ), it is refactored into a parallel composition.

$$\mathbf{dr1}(\Gamma,\ \mathbf{0}) \stackrel{\text{def}}{=} \mathbf{0}$$

$$\mathbf{dr1}(\Gamma, P \parallel Q) \stackrel{\text{def}}{=} (\mathbf{dr1}(\Gamma, P)) \parallel (\mathbf{dr1}(\Gamma, Q))$$

$$\mathbf{dr1}(\Gamma, P + Q) \stackrel{\text{def}}{=} (\mathbf{dr1}(\Gamma, P)) + (\mathbf{dr1}(\Gamma, Q))$$

$$\mathbf{dr1}(\Gamma, P \,\&\, Q) \stackrel{\text{def}}{=} (\mathbf{dr1}(\Gamma, P)) \,\&\, (\mathbf{dr1}(\Gamma, Q))$$

$$\mathbf{dr1}(\Gamma, a^*.P) \stackrel{\text{def}}{=} a^*.(\mathbf{dr1}(\Gamma, P))$$

$$\mathbf{dr1}(\Gamma, a.P) \stackrel{\text{def}}{=} \begin{cases} a.\mathbf{0} \parallel P & \Gamma(a) = \downarrow \\ a.(\mathbf{dr1}(\Gamma, P)) & \text{otherwise} \end{cases}$$

$$\mathbf{dr1}(\Gamma, \bar{a}.P) \stackrel{\text{def}}{=} \begin{cases} \bar{a}.\mathbf{0} \parallel P & \Gamma(a) = \uparrow \\ \bar{a}.(\mathbf{dr1}(\Gamma, P)) & \text{otherwise} \end{cases}$$

Figure 5.6: First deadlock resolution algorithm.

The second algorithm (Figure 5.7) works asymmetrically on input and output actions. For outputs, the strategy from the first algorithm is applied. Whereas for inputs, they are not parallelized, instead the respective output action is placed at in a parallel composition located at input level.

$$\mathbf{dr2}(\Gamma, \mathbf{0}) \stackrel{\text{def}}{=} \mathbf{0}$$

$$\mathbf{dr2}(\Gamma, P \parallel Q) \stackrel{\text{def}}{=} (\mathbf{dr2}(\Gamma, P)) \parallel (\mathbf{dr2}(\Gamma, Q))$$

$$\mathbf{dr2}(\Gamma, P + Q) \stackrel{\text{def}}{=} (\mathbf{dr2}(\Gamma, P)) + (\mathbf{dr2}(\Gamma, Q))$$

$$\mathbf{dr2}(\Gamma, P \,\&\, Q) \stackrel{\text{def}}{=} (\mathbf{dr2}(\Gamma, P)) \,\&\, (\mathbf{dr2}(\Gamma, Q))$$

$$\mathbf{dr2}(\Gamma, a^*.P) \stackrel{\text{def}}{=} a^*.(\mathbf{dr2}(\Gamma, P))$$

$$\mathbf{dr2}(\Gamma, a.P) \stackrel{\text{def}}{=} \begin{cases} a.(\mathbf{dr2}(\Gamma, P)) \parallel \bar{a}.\mathbf{0} & \Gamma(a) = \downarrow \\ a.(\mathbf{dr2}(\Gamma, P)) & \text{otherwise} \end{cases}$$

$$\mathbf{dr2}(\Gamma, \bar{a}.P) \stackrel{\text{def}}{=} \begin{cases} \bar{a}.\mathbf{0} \parallel (\mathbf{dr2}(\Gamma, P)) & \Gamma(a) = \uparrow \\ (\mathbf{dr2}(\Gamma, P)) & \Gamma(a) = \downarrow \\ \bar{a}.(\mathbf{dr2}(\Gamma, P)) & \text{otherwise} \end{cases}$$

Figure 5.7: Second deadlock resolution algorithm.

5.4 Implementation

The work by Almeida served as the starting point for the implementation of our solution. Programming language chosen for the implementation of the tool was Ocaml, a general-purpose and primarily functional programming language. The language is a great fit for the implementation of our algorithms due to its recursive algebraic data types, that, together with recursive functions and pattern matching, allows for very compact and easy to follow implementation. With a garbage collector, it also greatly facilitates programming tools like ours, since troubleshooting incorrect allocations and freeing of memory would have been very troublesome.

In the end, we ended up reimplementing most of the tool, reducing the code line count by half, but more importantly, significantly simplifying the implementation of the algorithms. Particularly with the deadlock detection algorithms, our implementation resulted in a main recursive function that effectively applies the transition relation to the CCS program, with another recursive function responsible for keeping track of the explored and to explore program states, while also collecting the program states that exhibit a deadlock.

The previous implementation required the user to input the CCS process in the form of an AST, directly in the code. This is not very practical, so, after porting the program to the *dune* build system, we implemented a parser for CCS, greatly facilitating the use of our tool.

In the reimplementation of the deadlock detection algorithm we found and applied an optimization that resulted in a significant speed-up in the analysis of processes with a high number of parallel compositions. The previous deadlock detection implementation, when reducing the process, it would check if synchronization between two actions is possible, only returning the reduction of the first synchronization it finds. In our implementation, instead of generating and analysing all the permutations of the parallel compositions, in each reduction step, returns all the possible synchronizations the process can perform. In this manner, all the possible reductions resulting from non-deterministic synchronization are explored without redundant computations.

Another improvement we developed was related to the deadlock resolution. After the deadlock detection step of the analysis, the deadlock resolution algorithms are then tasked with rewriting the program, receiving as input the program and a set of problematic actions. If the problematic actions are identified only by their label and type (input or output), as it was previously implemented, it is possible that the resolution applies the refactoring to the wrong actions. To illustrate, the following process features a deadlock $\overline{a}.b.a \mid a.\overline{a}.\overline{b}$. After synchronization over a, the process deadlocks in the following state $b.\overline{a} \mid \overline{a}.\overline{b}$. The problematic actions are $\overline{b}$ and $\overline{a}$. However, if all it's given to the resolution algorithm is the output action over a, algorithm will not be able to differentiate between the two instances of $\overline{a}$, resulting in both actions being parallelized, $(\overline{a} \mid b.a) \mid a.(\overline{a} \mid \overline{b})$. To solve this problem, after parsing the CCS program, each action is tagged with a unique identifier. These unique identifiers follow the action during the deadlock detection analysis and are passed through to the deadlock resolution step. Then, during resolution, the identifiers are used to find the exact problematic actions involved in the deadlock.

6

Conclusion

In this chapter we present an evaluation of our approach, comparing with two other state-of-the-art solutions. Next we discuss the limitations of our approach, along with future work.

6.1 Evaluation

In order to evaluate our solution, the code snippets we assembled for our deadlock bug collection (§ 4) were used as a benchmark suit. Each code snippet is a small program, less than 50 lines of code, containing just the essential logic and code necessary to replicate the deadlock bug. Besides our solution, we also evaluated Gong [15] and GCatch [16]. For our solution and Gong we used *migoinfer* [23] to obtain the MiGo representation of the original Go program, which was then passed as input to each tool. GCatch takes as input the original Go program.

In Table 6.1 we summarize the obtained results. The columns contain the deadlock verdict for each tool, with ✓ indicating a positive deadlock verdict, while ✗ indicates a negative deadlock verdict. Since every line corresponds to a real deadlock bug, the perfect tool would report a positive verdict in every line. The cells containing a '—' mark, indicate that the tool is unable to give a verdict. For bugs with the [1] mark, the *migoinfer* tool was unable to generate a valid output, and so, the GoDDaR and Gong tools could not produce a verdict. In all three cases, it was due to the use of Go's `context` package, which the *migoinfer* tool seems incapable of representing in MiGo. Bugs with [2] mark make use of asynchronous channels, which our tool does not support. The bug with [3] employs the use of channel-passing, where a channel is sent through another channel, a feature that none of the tools support.

The results depicted in the Table 6.1 show that our tool performs considerable better than GCatch, while matching the results of Gong.

Table 6.1: Deadlock detection evaluation over our bug collection.

Bug Id	GoDDaR	Gong [15]	GCatch [16]
cockroachdb#13197 [1]	—	—	✗
cockroachdb#13755 [1]	—	—	✗
cockroachdb#18101	✓	✓	✗
cockroachdb#24808 [2]	—	✗	✗
cockroachdb#25456	✓	✓	✗
etcd#6857 [3]	—	—	—
grpc-go#1275	✓	✓	✓
grpc-go#1424	✓	✓	✓
grpc-go#490	✓	✓	✗
kubernetes#25331 [1, 2]	—	—	✗
kubernetes#35672	✓	✓	✓
kubernetes#5316	✓	✓	✓
moby#21233	✓	✓	✗
moby#33293	✓	✓	✓
moby#33781	✗	✗	✓
moby#4395	✓	✓	✓

6.2 Limitations and Future Work

With the addition of internal/external choice and process replication, our approach can analyse a much wider range of Go programs. Nevertheless, there are still several directions where we can improve our work.

Regarding expressiveness, our solution lacks support for some Go features, namely channel passing, channel closure, and asynchronous channels. Therefore, some potential faulty situations in Go code are not analysed. To deal with the first, since CCS does not support channel passing, a different approach would be required for the abstract model, almost certainly a form of π-Calculus [20]. However, as corroborated with our findings, an empirical study on the use of the message passing concurrency model in Go [6] also found that a very small number (6%) of projects made use of channels that carry other channels. On the other hand, channel closure and asynchronous channels exbibit a much higher degree of use, with channel closure being used in 46% of projects, and 33% of channels being asynchronous.

In this work we created an automated pipeline to analyse Go code, however, the results are provided to the user in CCS form. For the typical developer, this can create a high entry barrier due to, not only the unfamiliar representation, but also to the difficulty in determining the mapping between the CCS form and the original source code. To facilitate these tasks, future work could complete the cycle by translating the proposed solution back to the Go language, and potentially highlight the changes made to the original code, in a more user-friendly manner.

Deadlock resolution is also a prime target for future work. Taking inspiration on the deadlock bug collection, new resolution strategies can be developed, targeting common deadlock patterns, ideally with an emphasis on retaining, as much as possible, the original program's behaviour.

Bibliography

[1] A. Almeida. "Automatic Resolution of Deadlocks in Concurrent Programs". MA thesis. Universidade Nova de Lisboa, 2021. URL: https://run.unl.pt/handle/10362/138810 (cit. on pp. 5, 33, 39–41, 45).

[2] J. A. Bergstra, A. Ponse, and S. A. Smolka, eds. *Handbook of Process Algebra*. North-Holland / Elsevier, 2001. DOI: 10.1016/b978-0-444-82830-9.x5017-6 (cit. on p. 32).

[3] E. G. Coffman, M. Elphick, and A. Shoshani. "System Deadlocks". In: *ACM Computing Surveys* 3.2 (1971), pp. 67–78. ISSN: 0360-0300. DOI: 10.1145/356586.356588 (cit. on p. 2).

[4] *Data Races and Race Conditions*. URL: https://doc.rust-lang.org/nomicon/races.html (visited on 2022-09-30) (cit. on p. 8).

[5] R. De Nicola and M. Hennessy. "CCS without τ's". In: *Proceedings of the International Joint Conference on Theory and Practice of Software Development (TAPSOFT)*. Springer, 1987, pp. 138–152. ISBN: 978-3-540-47746-4. DOI: 10.1007/3-540-17660-8_53 (cit. on p. 41).

[6] N. Dilley and J. Lange. "An Empirical Study of Messaging Passing Concurrency in Go Projects". In: *IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2019, pp. 377–387. DOI: 10.1109/SANER.2019.8668036 (cit. on pp. 23, 48).

[7] P. Fonseca et al. "A study of the internal and external effects of concurrency bugs". In: *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. 2010, pp. 221–230. DOI: 10.1109/DSN.2010.5544315 (cit. on p. 1).

[8] A. Francalanza, M. Giunti, and A. Ravara. "Unlocking Blocked Communicating Processes". In: *Electronic Proceedings in Theoretical Computer Science* 188 (2015), pp. 23–32. DOI: 10.4204/eptcs.188.4 (cit. on pp. 5, 32, 39).

[9] A. Frumusanu. *The Ampere Altra Max Review: Pushing it to 128 Cores per Socket*. 2021. URL: https://www.anandtech.com/show/16979/the-ampere-altra-max-review-pushing-it-to-128-cores-per-socket (visited on 2022-09-30) (cit. on p. 1).

[10] P. Godefroid and N. Nagappan. *Concurrency at Microsoft - An Exploratory Survey*. Tech. rep. MSR-TR-2008-75. 2008, p. 4. URL: https://www.microsoft.com/en-us/research/publication/concurrency-at-microsoft-an-exploratory-survey/ (cit. on p. 2).

[11] D. Harel and A. Pnueli. "On the Development of Reactive Systems". In: *Logics and Models of Concurrent Systems*. Springer, 1985, pp. 477–498. ISBN: 978-3-642-82453-1. DOI: 10.1007/978-3-642-82453-1_17 (cit. on p. 14).

[12] Isloor and Marsland. "The Deadlock Problem: An Overview". In: *Computer* 13.9 (1980), pp. 58–78. ISSN: 1558-0814. DOI: 10.1109/MC.1980.1653786 (cit. on p. 2).

[13] N. Kobayashi, B. C. Pierce, and D. N. Turner. "Linearity and the Pi-Calculus". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21.5 (1999), pp. 914–947. ISSN: 0164-0925. DOI: 10.1145/330249.330251 (cit. on p. 32).

[14] T. Kulesza. *Go Developer Survey 2019 Results*. 2020. URL: https://go.dev/blog/survey2019-results (visited on 2022-09-30) (cit. on p. 4).

[15] J. Lange et al. "Fencing off Go: Liveness and Safety for Channel-Based Programming". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, 2017, pp. 748–761. ISBN: 9781450346603. DOI: 10.1145/3009837.3009847 (cit. on pp. 31, 40, 47, 48).

[16] Z. Liu et al. "Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2021, pp. 616–629. ISBN: 9781450383172. DOI: 10.1145/3445814.3446756 (cit. on pp. 27, 47, 48).

[17] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. ii).

[18] S. Lu et al. "Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics". In: *ACM SIGOPS Operating Systems Review* 42.2 (2008), pp. 329–339. ISSN: 0163-5980. DOI: 10.1145/1353535.1346323 (cit. on pp. 2, 23).

[19] R. Milner. *A Calculus of Communicating Systems*. Vol. 92. Lecture Notes in Computer Science. Springer, 1980. ISBN: 978-3-540-10235-9. DOI: 10.1007/3-540-10235-3 (cit. on p. 15).

[20] R. Milner. *Communicating and Mobile Systems: The π-Calculus*. Cambridge University Press, 1999. ISBN: 0521658691 (cit. on p. 48).

[21] A. Møller and M. I. Schwartzbach. *Static Program Analysis*. Department of Computer Science, Aarhus University. 2022. URL: http://cs.au.dk/~amoeller/spa/ (cit. on p. 5).

[22] L. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3 (cit. on p. 28).

[23] N. Ng. *gospal*. URL: https://github.com/nickng/gospal (visited on 2022-09-30) (cit. on pp. 40, 47).

[24] N. Ng and N. Yoshida. "Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis". In: *Proceedings of the 25th International Conference on Compiler Construction (CC)*. ACM, 2016, pp. 174–184. ISBN: 9781450342414. DOI: 10.1145/2892208.2892232 (cit. on p. 25).

[25] R. Pike. "Go at Google". In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH)*. ACM, 2012, pp. 5–6. ISBN: 9781450315630. DOI: 10.1145/2384716.2384720 (cit. on p. 8).

[26] *Rust Programming Language*. URL: https://www.rust-lang.org/ (visited on 2022-09-30) (cit. on p. 7).

[27] S. Taheri and G. Gopalakrishnan. "GoAT: Automated Concurrency Analysis and Debugging Tool for Go". In: *IEEE International Symposium on Workload Characterization (IISWC)*. 2021, pp. 138–150. DOI: 10.1109/IISWC53511.2021.00023 (cit. on p. 25).

[28] TIOBE. *TIOBE Index for January 2022*. 2022. URL: https://www.tiobe.com/tiobe-index/ (visited on 2022-09-30) (cit. on p. 3).

[29] T. Tu et al. "Understanding Real-World Concurrency Bugs in Go". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2019, pp. 865–878. ISBN: 9781450362405. DOI: 10.1145/3297858.3304069 (cit. on pp. 2, 24, 25, 30, 36).

[30] T. Yuan et al. "GoBench: A Benchmark Suite of Real-World Go Concurrency Bugs". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 187–199. DOI: 10.1109/CGO51591.2021.9370317 (cit. on pp. 24, 36).