Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

Verification of concurrent Go programs using Uppaal

Arne Philipeit
[redacted]

April 30, 2020

A Final Year Project submitted in partial fulfilment
of the requirements for the degree of
MAI (Computer Engineering)

Supervised by Dr. Vasileios Koutavas

Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

I have read and I understand the plagiarism provisions in the General Regulations of the University Calendar for the current year, found at `http://www.tcd.ie/calendar`.

I have also completed the Online Tutorial on avoiding plagiarism 'Ready Steady Write', located at `http://tcd-ie.libguides.com/plagiarism/ready-steady-write`.

Signed:  Date: **29.04.2020**

Abstract

This dissertation describes a way to model concurrent Go programs in Uppaal, a modeling and verification software for concurrent systems. Toph is presented as an automatic tool that translates Go code to Uppaal systems. Toph and Uppaal are able to verify channel safety and the impossibility of global deadlocks for a variety of test programs and make it easier for users than prior tools to examine failure cases.

Acknowledgements

First and foremost I would like to thank Dr. Vasileios Koutavas as my supervisor for his great support and many fruitful discussions.

I would also like to thank my family and friends for years of support that led me to this point.

Contents

List of Figures

List of Tables

Nomenclature

AST	Abstract Syntax Tree
CCS	Calculus of Communicating Systems
CFSM	Communicating Finite State Machine
CSP	Communicating Sequential Processes
FCG	Function Call Graph
FIFO	First-in-first-out
IR	Intermediate Representation
MPST	Multiparty Session Types
SSA	Static Single Assignment
TCP	Transmission Control Protocol

1 Introduction

Go is a modern programming language originally developed by Google with the goal of simplifying the development of concurrent software running in data centers(1). Go has seen widespread adoption and increasing popularity over the last decade (2). It is the first major programming language to build its concurrency paradigms around channels, an idea that originates in Formal Verification research. Since channels in a practically used programming language are relatively new, and Go is often for critical aspects of software systems, there is a need for development and verification tools around concurrency in Go.

In this dissertation, I present ways to model Go programs in Uppaal, a long standing program for the specification and verification of concurrent systems. These ideas are implemented in an automatic translation tool, called Toph, that converts Go programs into Uppaal systems following the developed models.

Chapter 2 Background provides an introduction to Formal Verification in general, details the existing research and tools for Go concurrency verification, and gives gives an overview of Uppaal.

Chapter 3 Methodology goes into detail on the semantics of channels in Go and presents the newly developed model for Go channels in Uppaal. The chapter also discusses how all other aspects of the Go programming language that are relevant to the problem are modeled in Uppaal.

Chapter 4 Implementation covers the development and implementation of Toph, including the design choices made and problems solved in the process.

Chapter 5 Evaluation examines the effectiveness of the used approach by testing Toph and Uppaal with 116 Go test programs.

And at the end, Chapter 6 Conclusion summarizes the gained insights and provides a perspective for future improvements.

For readers unfamiliar with Go, the following Section 1.1 offers a brief introduction to concurrency and channels in Go. Section 1.2 presents the motivating example referenced throughout. And Section 1.3 lists the contributions of this work.

1.1 Concurrency in Go

Go offers Goroutines for concurrency, which are similar to light threads in other languages. It is possible to run hundreds of Goroutines at the same time while the Go runtime schedules them between only a few operating system threads. The `go` keyword in front of any function call starts the called function in a new Goroutine.

"Do not communicate by sharing memory; instead, share memory by communicating."(3) is a fundamental principle of the Go programming language. In Go, channels implement this idea of sharing memory by communicating.
New channels get created using the builtin `make` function, which takes the type of the new channel as its argument. Different Goroutines can send and receive values via channels. Either the sending Goroutine waits for a receiver or vice versa. Syntactically, both operations use the <- operator, either pointing into (send) our out of (receive) the channel variable.

Channels can also be buffered. Senders fill the channel buffer, while receivers empty it. This provides for asynchronous communication via channels. However, if the buffer is full, additional senders have to wait for a receiver. If the buffer is empty, receivers have to wait for a new sender. Buffers preserve the order in which values are sent and received via a channel. The size of the channel buffer gets passed as an optional second argument to the `make` function. It is useful to think of an unbuffered channel as having buffer size 0.

And finally, Go channels can be closed with the builtin `close` function. Sending via a closed channel or closing a channel twice is not allowed and causes programs to crash. However, receiving via a closed channel is allowed and either returns remaining buffer elements or the default value for the channel value type, for example 0 for `int`.

Loops ranging over channels are a useful syntactic construct offered in Go. Each loop iteration processes one received value. The loop only exits when the channel gets closed and there are no more values to be processed. Otherwise it might block, waiting to receive the next value for processing.

Select statements are a powerful part of Go. They look similar to switch statements. However, the different cases in a select statement deal with different channel communication operations. The first case that can be executed gets chosen. If a default clause is present, a case has to be chosen immediately, or otherwise the default clause gets executed. Select statements without a default clause, on the other hand, are blocking.

1.2 Motivating Example: WordCounter

Figure 1.1 shows the motivating example - *WordCounter* - used throughout my dissertation. It is a working Go program that goes through each file in a given directory and counts all the words. Finding files, counting words in a file, and aggregating results happen in concurrently running Goroutines that communicate via channels. *WordCounter* is a simplified version of a similar program that computes MD5 hashes for files(4).

```
1  var errChan chan error = make(chan error, 3)
2  var abortChan chan struct{} = make(chan struct{})
3
4  func findFilesInFolder(root string,
5                         filesChan chan string) {
6      files, err := ioutil.ReadDir(root)
7      if err != nil {
8          errChan <- err
9          close(filesChan)
10         return
11     }
12     // toph: max_iter=3
13     for _, file := range files {
14         if file.IsDir() {
15             continue
16         }
17         select {
18         case filesChan <- file.Name():
19         case <-abortChan:
20             break
21         }
22     }
23     close(filesChan)
24 }
25
26 func countWords(filesChan chan string,
27                 wordCountsChan chan int) {
28     for file := range filesChan {
29         select {
30         case <-abortChan:
31             return
32         default:
33         }
34         content, err := ioutil.ReadFile(file)
35         if err != nil {
36             errChan <- err
37             break
38         }
39         text := string(content)
40         count := strings.Count(text, " ")
41         wordCountsChan <- count
42     }
43 }
44
45 func main() {
46     filesChan := make(chan string, 2)
47     wordCountsChan := make(chan int, 2)
48
49     root := os.Args[1]
50     go findFilesInFolder(root, filesChan)
51
52     waitChan := make(chan struct{})
53     doneChan := make(chan struct{})
54
55     // toph: min_iter=2, max_iter=2
56     for i := 0; i < 2; i++ {
57         go func() { // Helper 1
58             countWords(filesChan, wordCountsChan)
59             waitChan <- struct{}{}
60         }()
61     }
62     go func() { // Helper 2
63         // toph: min_iter=2, max_iter=2
64         for i := 0; i < 2; i++ {
65             <-waitChan
66         }
67         close(doneChan)
68     }()
69
70     totalCount := 0
71     for {
72         select {
73         case err := <-errChan:
74             close(abortChan)
75             fmt.Println(err)
76             return
77         case c := <-wordCountsChan:
78             totalCount += c
79         case <-doneChan:
80             fmt.Printf("counted %d words\n",
81                 totalCount)
82             return
83         }
84     }
85 }
```

Figure 1.1: Motivating example *WordCounter* concurrently counts words in files.

Figure 1.2 gives an overview of the concurrently running functions and Goroutines in

WordCounter and the six channels used between them. The first three channels are straightforward. `filesChan` transmits paths of files to be processed. `wordCountsChan` transmits word count values of individual files, which `main` adds up. And `errChan` is used to send any encountered errors.
For such a small program, it would usually be okay to exit while various Goroutines are still running. However, to illustrate a proper abort mechanism that would be used in a larger program, if the `main` function receives an error it stops the other Goroutines, signaled by closing `abortChan`.
The `main` function needs to know when all files have been processed so that it can then print the result. Therefore, it uses two anonymous inner functions, referred to as `Helper 1` and `Helper 2`, that communicate via `waitChan`. `Helper 2` informs the `main` function that the two invocations of `countWords` have no more files to process, allowing the `main` function to print the final result. The communication happens by closing `doneChan`. In real-world Go code, this would be done using a wait group instead.

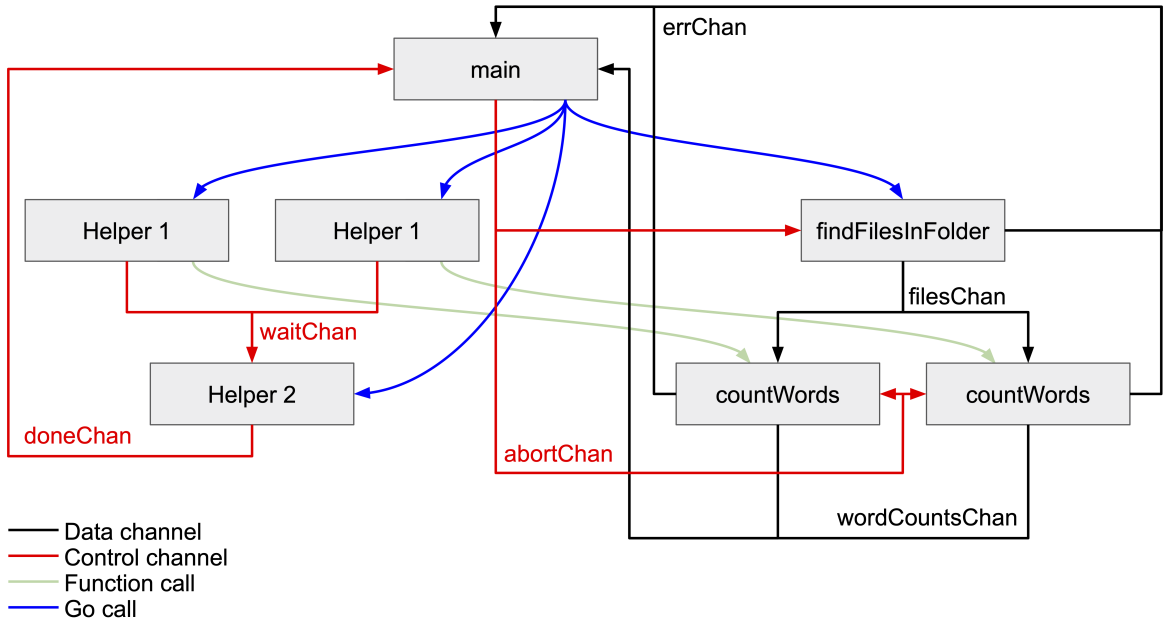


Figure 1.2: Communication between functions and Goroutines in *WordCounter*

This example shows many language constructs and paradigms used in real world Go programs. Some channels are used to transmit data, while others are used purely for program control. There are one-to-one, one-to-many, and many-to-one channel connections between sending and receiving Goroutines. The program closes `abortChan` effectively as a broadcast signal and uses the fact that reading from `abortChan` or `doneChan` succeed immediately if these channels are closed.

Uppaal is able to verify channel safety for all six channels using the system generated by Toph. In addition, Uppaal can verify that the program contains no global deadlocks where

one or more Goroutines would wait for a channel operation indefinitely.
These things are hard to reason about for regular programmers. In fact, my initial implementation of *WordCounter* had multiple issues that were revealed through Uppaal, such as potentially closing a channel twice, or a Goroutine not being able to abort because it is blocked, attempting to send a second error value. Even with unit testing it would have taken a lot of effort to catch such mistakes. Uppaal made it easy to find the exact causes by providing example traces for each issue.

1.3 Contributions

A Uppaal model for Go channels and channel operations was developed and tested (Section 3.2) based on a detailed analysis of the semantics of channels, loops ranging over channels, and select statements as implemented in the Go runtime (Section 3.1). In particular the model for select statements is more accurate than those in previous works with respect to the reachability of the default clause. Uninitialized channels, channel aliasing, buffered channels, and closing channels are supported. These were all issues with some approaches in prior work. Channel safety can easily be verified by checking that a channel's bad state is unreachable.

As far as I was able to determine, Toph is the first tool that can convert Go code into Uppaal systems. Toph provides more easily understandable models in Uppaal than prior tools for Go concurrency verification. Functions and control flow structures like if statements and for loops are converted directly to Uppaal models and are easily identifiable. This allows Go programmers to leverage the strengths of Uppaal, providing program traces for any failing queries.
Achieving this required devising Uppaal models for channel variables and function calls, including the spawning of Go routines in Uppaal (which does not directly support processes starting other processes), argument and result passing, and correctly capturing variables in closures. Toph correctly handles all of these throughout the translation process.
Additionally, Toph determines the maximum number of function invocations and created channels in Go programs to adjust the number of process instances used in Uppaal. This also helps users to navigate Uppaal systems more easily and means less work for Uppaal.

All of these efforts combined make it possible to analyse complex Go programs, such as the one presented in Figure 1.1, with Toph and Uppaal. Toph was tested with 116 test programs, including test suites used in prior work, and is able to translate 92 into Uppaal systems without any warnings of unsupported Go code. Toph translates all 116 programs, over 5000 lines of Go code, in a few seconds.
For all programs written explicitly to test the model for Go channels, range loops over channels, and select statements Uppaal is able to verify the expected reachability or

unreachability of code and channel safety. Uppaal is generally able to handle Go programs with large and complex state spaces, such as the classic dining philosophers program.

2 Background

This chapter provides an overview of Formal Verification, previous work on verification of concurrent Go programs, and Uppaal in order contextualize the approach described in this dissertation.

2.1 Formal Verification

Formal Verification is an area of Computer Science that is concerned with the correctness of hardware and software. In practice, it is often used to ensure that critical software in everything from routers and network switches (5) to the space shuttle (6) works as intended. It can also be used to ensure that algorithms, such as a sorting algorithm in a standard library, always return the correct result (7). The main focus in Formal Verification research around Go has been around its concurrency patterns, in particular Go routines and channels. The aim is to ensure that no bugs in the usage of channels and no deadlock or other common concurrency issues between Go routines running in parallel exist in a program.

There are known fundamental limits to the problems Formal Verification deals with. An algorithm that could correctly handle any Turing-complete input program in finite time would solve the Halting problem and can therefore not exist. Similarly, a mathematical model for programs in which proofs for the absence of all errors could be generated would violate Gödel's Incompleteness Theorem.

Therefore, different methods in Formal Verification are generally concerned with the following three properties (8):

- Soundness: The method can only indicate that a program is bug free, if there really are no bugs.
- Completeness: The method can only indicate that a program contains a bug, if there actually is a bug.
- Termination: The method is guaranteed to terminate and will not reason about the program indefinitely.

Another fundamental limit in Formal Verification is that there can never be a method that has all three properties. Different tools and methods choose only two properties, most commonly Soundness and Termination. This means that these tools can indicate in finite time if a program is bug free. However, if such a tool indicates an error, the program might still be correct. In that case it can sometimes be possible to partly rewrite the program or to provide additional information to help the verification tool. Toph uses comments starting with `toph:` for loop bounds for example.

One of the reasons Soundness and Termination are usually chosen as properties is that other approaches beyond Formal Verification can cover Completeness to some extent. Unit testing (testing individual functions in isolation), integration testing (testing multiple parts or an entire software system at once), and fuzz testing (testing a program with random inputs to provoke bugs) are commonly used tools to try to catch bugs. On the other hand, tools with the Soundness and Termination properties focus on ensuring the absence of bugs, which is something software testing can not do. If all tests pass, a program can still contain a bug.

Within Formal Verification there are two fundamental kinds of methods.
Deductive verification aims to translate a program into a mathematical logic system, such as first order logic or Hoare logic. With the examined program translated into a logic system, the correctness proof can then be done automatically.
Algorithmic verification also uses a model of the examined program, but within a decidability framework such as finite state machines or push-down automata. Within such a framework an algorithm can then explore the state space.

Not all methods in Formal Verification have the same intended use or audience. Some are highly theoretical and developed mainly to advance the field, while others can be much more targeted towards practical use by software developers. There is always a trade off between the two. A software engineer might be more interested in being able to investigate why a tool indicates that a program is incorrect, whereas a more theoretically minded approach can sometimes make simplifying assumptions or come up with new constructs. Go's channel based communication patterns are an example of how a more theoretical approach in Formal Verification (Communicating Sequential Processes) later informed the design of a programming language (3).

The following three ideas are commonly used in Formal Verification of concurrent programs, including by the previous work on Go, and are therefore briefly explained here.

Calculus of Communicating Systems (CCS)

A CCS systems consists of several communicating processes. Each process has several states and can change states by performing actions. Actions can either be silent or require

communication with another process, where one process sends and the other receives. Communication is used purely to synchronize processes and does not involve actual values. The behaviour of a process in a particular state is described by a behavioural type. Because processes can end up in earlier states, behavioural types can use recursive definitions. (9)

Communicating Sequential Processes (CSP)

CCS and CSP are very closely related. While CCS is a process algebra, CSP is a "highly influential proposal for a programming language" with the same paradigms (9).

Session Types

Session types are used to describe the communication behaviour of an entity. A session type consists of a sequence of instructions that can include recursive definitions and choice to allow for different branches.

2.2 Previous Work on Go Concurrency Verification

The channel-based concurrency paradigms of Go have seen some interest by the Formal Verification research community in the last five years. Over time, the tools and methods that were developed for Go have steadily grown in their capabilities and support for more and more complex language features and concurrency patterns. This section gives a chronological overview of these efforts, the chosen approaches, limitations, and contributions.

In (10), Ng and Yoshida present the first static deadlock detection tool for Go - *dingo-hunter* - based on session types and communicating finite state machines (CFSMs). *dingo-hunter* builds a local session type for each Goroutine in a program. Session types include the creation of channels, sending and receiving via channels, select statements, and the closing of a channel. For each local session type of a Goroutine *dingo-hunter* builds a CFSM. Because standard session types assume that channels are one-to-one, the tool also synthesises a CFSM for each Go channel. In the final step, *dingo-hunter* aims to build a global session graph by merging all Goroutine and channel CFSMs together. If this step is successful, their tool indicates that the original Go program is correct.
This approach has significant limitations. It can not deal with buffered channels and the corresponding asynchronous communication. It does not handle uninitialized channels with a *nil* value correctly (11). It does not support the creation of channels or Goroutines in a loop or conditionally in an if-statement. Some workarounds are proposed to address this. *dingo-hunter* also only flags closing a channel twice as a deadlock and not explicitly (11).

As far as I was able to determine, their model simplifies default clauses in select statements. The default clause is still reachable even if some other cases can also immediately succeed (see Section 3.1.3).

In (12), Stadtmüller et al. present an approach based on regular expressions, extended with a fork operator, and finite state automata (FSA) and *Gopherlyzer* as an implementation of these ideas. This work marks the first use of a symbolic deadlock detection tool. According to (11), this analysis is "extremely limited". It also does not support buffered channels. The ability to close channels is not modelled. Select statements with non-trivial case bodies and multiple calls to the same function are not supported.

In (13), Lange et al. develop an approach based on CCS (Calculus of communicating systems), a process calculus very similar to CSP, which inspired the design of channels in Go. They provide *GoInfer* (written in Go) as a tool to convert Go code to their MiGo process calculus language and *Gong* (written in Haskell) to perform liveness and channel safety analysis. Their work primarily addresses previous issues around the termination of programs by ensuring that finitely many communication patterns are used in a program that may otherwise be unbounded. This is also the first work that addresses buffered channels and asynchronous channel communication.
A remaining limitation is that more complex programs can lead to a state explosion, where a lot of possible states have to be examined to verify properties. In (11), the same authors also discuss lacking support for closures such as `Helper 1` and `Helper 2` in *WordCounter*.

In (14), Midtgaard et al. discuss general ideas around verification of programs with concurrent processes and use Go as a proof of concept. Similar to (10), they also first look at each Goroutine separately before they consider different interleavings using lattice theory, an area of mathematics dealing with partially ordered sets and equivalent algebra (15). However, their work only considers a very small subset of the Go language, which is necessary to ensure their method remains sounds. The tool they developed is written entirely in OCaml, including the scanning and parsing of Go code.
This approach only allows for a fixed number of top level processes and channels. Therefore, their contributions remain theoretical and are not currently applicable for almost any real-world Go program.

With (11), Lange et al. improve upon their previous work. *Godel Checker*, their new tool, uses a simpler workflow: first building behavioural types (similar to session types) and then analysing them separately with a model checker and a termination checker, where the termination checker is required to detect partial deadlocks. This approach resolves previous issues with closures and state explosion, which leads to faster verification. Their use of the μ-calculus for the model checker also theoretically allows for a lot of queries beyond channel safety and liveness analysis.

One remaining issue is that *Godel Checker* is still not able to handle Goroutines spawned in a loop. The authors work around this by unrolling a loop in one of their test cases.
Based on the description in the paper, it appears that the model for select statements uses the same simplification described above. The default clause is always reachable, even if other cases are possible to succeed immediately.

In the most recent work on Go concurrency verification (16), Castro et al. take a broader approach that looks at all lossless, ordered channel communication and also apply their work to TCP connections. They develop and apply the first theory of distributed multiparty session types (MPST) which is applicable beyond Go. This approach requires a global protocol specification from the user and automatically generates MPSTs for each Goroutine or communication end point.
Their approach allows them to model channel-over-channel passing which no prior work on Go concurrency verification can handle. Usually verification tools, including Toph with Uppaal, assume that only data will be passed over channels, which is the case in a lot of Go programs.

Overall, most work in this area starts with a theoretical framework and aims to apply it to Go. This leads to some of practical shortcomings and means that the majority of the presented tools could not handle *WordCounter* correctly as input due to missing support for buffered channels, closed channels, complex select statements, or closures among other issues.
Toph and Uppaal represent a less theoretical and more engineering focused approach. There is more emphasis on enabling users to investigate results rather than simply indicating success or failure to verify.
Chapter 5 goes into more detail on the strengths and limitations of Toph and Uppaal, including in-depth comparisons with the tools already in existence.

2.3 Uppaal

Uppaal is a verification tool for timed, concurrent systems, similar to those in CCS (17), that dates back to 1995 (18). It is written in Java, jointly developed and maintained by researchers at Aalborg University in Denmark and Uppsala University in Sweden and available online (see `http://www.uppaal.org`). Uppaal provides a graphical user interface to define processes, simulate systems, and verify queries against systems. Uppaal also defines open file formats for systems and queries.
The following is a brief introduction to Uppaal and the subset of features that are used to model Go programs. Figure 2.1 shows a simple made-up Uppaal process (part of a system) that serves as an example.

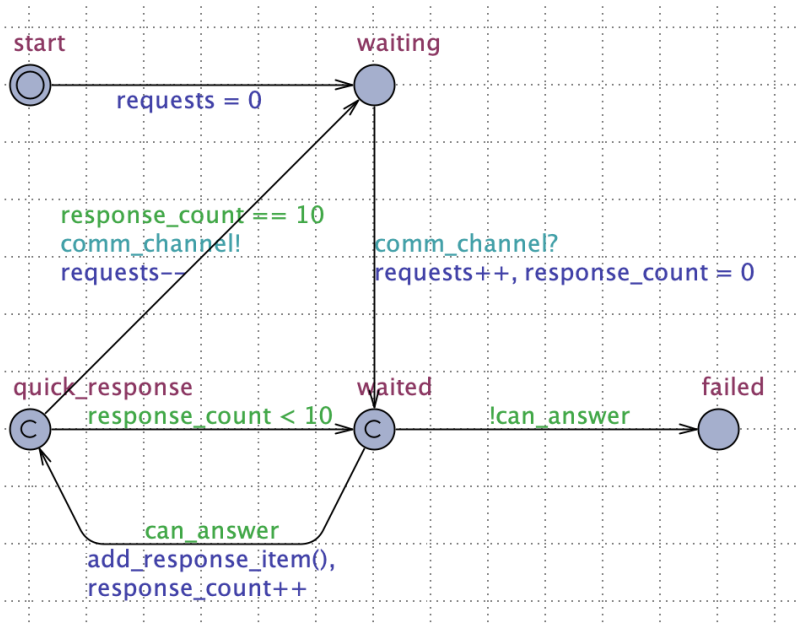


Figure 2.1: Simple made-up Uppaal process example.

A Uppaal system contains several processes. Each process consists of several states, including an initial state, and transitions between states. In Figure 2.1, the initial `start` state is in the top left corner with two circles. The names of states are shown in dark red/purple (■).

A system can have global declarations, and each process can have local declarations. Global and local declarations use a C-like syntax and can be constants, variables, arrays, structs, and functions. The global declarations of a system are accessible everywhere. The local declarations of a process are only accessible for that process and not visible elsewhere. Each process can additionally define parameters.

A system can contain several instances of a process, possibly instantiated with different parameters. Each process instance has its own set of local variables and can generally operate independently from other instances of the same process. All process instances exist and can operate from the start of the system simulation. It is not possible to dynamically create instances during simulation.

State transitions in Uppaal can define guards and updates. A guard is a boolean expression that can involve global and local variables. If a transition has a guard, the guard has to evaluate to true to enable the transition. Uppaal displays transition guards in green (■). In Figure 2.1, the transition from `quick_response` to `waiting` can only be taken when `response_count` (a variable) is equal to 10.
Updates are statements, for example variable assignments or function calls, that are executed when a transition gets taken. Uppaal shows updates in dark blue/violet (■). In Figure 2.1, the same transition as before subtracts one from the `requests` variable when

the process takes the transition.

Uppaal offers CCS-like channels which are different from Go channels. Uppaal channels are declared as global or local variables or arrays (if in aggregate) and do not need to be further initialized. They can only be used to synchronize processes, not to transmit or buffer data. It is not possible to close a Uppaal channel.
State transitions can require synchronization, either a send (!) or receive (?) on a Uppaal channel. In order for a process instance to take a transition that requires synchronization, a different process has to take a transition with the opposite channel operation on the same channel at the same time. A transition is disabled if no other process instance is able to synchronize. Synchronization requirements are displayed in turquoise (■). In Figure 2.1, transitioning from `quick_response` to `waiting` requires sending via `comm_channel` and transitioning from `waiting` to `waited` requires receiving via `comm_channel`.

In order to simplify readability, guards, synchronisation statements, and updates of transitions in this dissertation and generated by Toph are always listed in this order from top to bottom. Thinking about them in this order makes sense. First, a transition is disabled if the guard evaluates to false. Secondly, a transition of a process might need to synchronize with a different process. If that is not possible, the transition is also disabled. And finally, if a process, or two processes in synchronisation, take(s) a transition all transition updates get performed at the end.

A state can be marked as committed. A committed state has to be left as soon as possible. If one or more process instances of a system are in committed states, only transitions out of the committed states are considered as next steps in the simulation. Otherwise, all enabled transitions would be considered.
This means that a chain of committed states can be used for atomic behavior with respect to non-committed states. In Figure 2.1, the `waited` and `quick_response` states are committed. In a system with no other committed states, this process would have to leave the two states before any other process can make further transitions.

Uppaal provides a simple query language for systems. Conditions can include the values of variables, the states of process instances, and whether the system as a whole is in deadlock or can make further transitions. Queries can use one or more of the operators listed in Table 2.1. For example, the query `"A[] response_count <= 10"` checks that the globally declared variable `response_count` from Figure 2.1 never exceeds the value 10. Uppaal evaluates these queries against a system and indicates if they can be proven.

[1] see https://www.it.uu.se/research/group/darts/uppaal/help.php?file=RSL-Semantics.shtml

Table 2.1: Uppaal query operators, adapted from Uppaal Help[1].

Name	Operator	Explanation	Equivalence
Possibly	`E<> p`	It is possible to reach a state that satisfies p.	
Invariantly	`A[] p`	All reachable states satisfy p.	`not E<> not p`
Potentially Always	`E[] p`	It is possible to take a transition sequence where every state satisfies p.	
Eventually	`A<> p`	All possible transition sequences eventually reach a state satisfying p.	`not E[] not p`
Leads to	`p -> q`	If p holds eventually, q will hold as well.	`A[] (p imply A<> q)`

3 Methodology

The general idea is to only model the program control flow, meaning branching, loops, function calls, and concurrently running Goroutines, and not the actual processed data. This greatly reduces the model state space while still generally preserving the main properties of interest relating to global and partial deadlock. Where data does determine control flow in the examined program, for example through the condition of an if statement, the model over-approximates with non-deterministic choices. In the model, the program could always take any branch of an if statement.
In Go, channels heavily influence control flow because channel operations can block Goroutines and lead to complicated dependencies in control flow between Goroutines. Select statements and loops ranging over channels are two control flow structures in Go that deal with channels. Therefore, these are the main focus of the developed model.

Section 3.1 goes through the intricate semantics of channels, loops ranging over channels, and select statements in Go in detail. This analysis is important in order to be able to correctly model channels in Uppaal.

Section 3.2 introduces the devised model for channels, adhering to all the described channel semantics. The section also introduces how channel operations and select statements are modeled in Uppaal.

Section 3.3 explains how control flow structures, such as if statements and for loops, are modeled in Uppaal. This includes addressing issues such as infinite loops and the model for loops ranging over channels.

Section 3.4 then looks at modelling functions, function calls, the spawning of new Go routines using the go keyword, function arguments, return values, and variables captured in closures.

3.1 Channel Semantics in Go

3.1.1 Basics

Each channel has a type of values it can transmit. The default values for channel variables in Go is `nil`. Sending or receiving on, or closing a `nil` channel causes a runtime panic. Channels can be created with the built-in `make` function. A non-negative channel buffer size can be specified when a channel gets created with make. By default, channels have a buffer size of zero, i.e. are unbuffered and synchronous. The buffer size of a channel can not be changed after creation.

Figure 3.1 shows an extract of *WordCounter*. The channel variable `errChan` gets declared (line 1) and assigned a new channel with a buffer size of 3. On line 2, `abortChan` gets declared and assigned a new unbuffered channel. `errChan` transports `error` values (`error` is a type in Go). `abortChan` does not transport any data since `struct{}` refers to an empty structure of size zero.

```
1 var errChan chan error = make(chan error, 3)
2 var abortChan chan struct{} = make(chan struct{})
```

Figure 3.1: Creation of two channels (extract from Figure 1.1).

Channel buffers work as FIFO queues. The Go runtime implements them as ring buffers[1]. If a channel has a buffer with remaining capacity, a send statement can insert the transmitted data into the buffer immediately and is non-blocking. Similarly, if a channel has a non-empty buffer, a receive statement can remove the transmitted data from the buffer immediately and is also non-blocking.

`errChan` in Figure 1.1 and 3.1 has a buffer size of 3 so that `findFilesInFolder` and the two invocations of `countWords` can each send one error value without blocking. Both functions exit after finding an error.

If a channel is unbuffered or has a full buffer, a send statement (outside of a select case) is blocking. The go-routine requesting to send via the channel has to wait until a different go-routine is attempting to read via the same channel. Then the two go-routines can synchronize. If there are several go-routines waiting to send, one will synchronize at random.

Conversely, if a channel is unbuffered or has an empty buffer, a receive statement (outside of a select case) is blocking. The go-routine requesting to receive via the channel has to wait until a different go-routine is attempting to send via the same channel. Then, again, the two

[1] see `https://golang.org/src/runtime/chan.go`

go-routines can synchronize. If there are several go-routines waiting to receive, one will synchronize, again, at random.

If there are pending senders, any new receiver can immediately be matched. If there are pending receivers, any new sender can immediately be matched. This means a situation with pending senders and receivers can never arise.

Figure 3.2 shows `Helper 1` and `Helper 2` synchronising via `waitChan`. Each invocation of `Helper 1` sends once on the channel, while `Helper 2` receives once for each send, ensuring that `Helper 2` can only proceed to close `doneChan` after all `Helper 1` invocations have finished.

```
1  waitChan := make(chan struct{})
2  doneChan := make(chan struct{})
3  // toph: min_iter=2, max_iter=2
4  for i := 0; i < 2; i++ {
5      go func() { // Helper 1
6          countWords(filesChan, wordCountsChan)
7          waitChan <- struct{}{}
8      }()
9  }
10 go func() { // Helper 2
11     // toph: min_iter=2, max_iter=2
12     for i := 0; i < 2; i++ {
13         <-waitChan
14     }
15     close(doneChan)
16 }()
```

Figure 3.2: `Helper 1` and `Helper 2` synchronising via `waitChan` (extract from Figure 1.1).

Go channels can be closed with the built-in `close` function. Closing a channel multiple times causes a runtime panic. Attempting to send on a closed channel causes a panic, even if the send request has been pending before the channel was closed. Receiving via a closed channel is a non-blocking operation. It either returns the next remaining buffer element or the default value for the channel value type. A `chan int` would return 0, for example.

The `abortChan` and `doneChan` in Figure 1.1 use this fact. `findFilesInFolder` and `countWords` both attempt to read from `abortChan` in select statements and know to abort if they succeed. Therefore, closing `abortChan` is effectively used as a broadcast signal.

A receive operation can return a second, boolean value. This value indicates whether the received value comes from the channel buffer or a sender, or whether it is the default value for the channel type in case the channel is closed. Line 2 of the code on the righthand side of Figure 3.3 shows an example.

3.1.2 Range-Loops over Channels

Go offers loops that range over an array, slice, map, or channel. At the start of a range-loop over a channel, the program attempts to receive a value via the channel. If the channel got closed and the received value is the default value for the channel type, the loop exits. Otherwise, the loop body processes the received value.

In practice, range-loops over channels are just syntactic sugar. Alternatively, an infinite loop with a receive statement at the start of the loop body followed by a conditional break can be used.

Figure 3.3 shows the range-loop over `filesChan` in `countWords` and the equivalent for loop replacement. Each loop iteration processes one file path from `filesChan` until the channel gets closed and the buffer is empty. This is a very common pattern for functions that process or consume values from a channel in concurrently running Go routines.

```
for file := range filesChan {
    select {
    case <-abortChan:
        return
    default:
    }
    content, err := ioutil.ReadFile(file)
    if err != nil {
        errChan <- err
        break
    }
    text := string(content)
    count := strings.Count(text, " ")
    wordCountsChan <- count
}
```

```
for {
    file, ok := <-filesChan
    if !ok {
        break
    }

    select {
    case <-abortChan:
        return
    default:
    }
    content, err := ioutil.ReadFile(file)
    if err != nil {
        errChan <- err
        break
    }
    text := string(content)
    count := strings.Count(text, " ")
    wordCountsChan <- count
}
```

Figure 3.3: For loop ranging over channel (extract from Figure 1.1) and equivalent alternative code.

3.1.3 Select Statements

Select statements are a powerful language construct in Go. Similar to switch statements, select statements also contain different cases and an optional default clause, one of which gets executed. However, select statements deal with channel operations. Each select case

attempts to send or receive via a channel. If one or more cases can succeed immediately, one is chosen at random. If all cases have to wait, and there is a default clause, the default clause gets executed. If there is no default clause, the select statement blocks until one of its cases can succeed.

Therefore, a select statement with a default clause never blocks. And an empty select statement without any cases and without a default clause (not used in practice) blocks forever.

A select statement with a single case and without a default clause is equivalent to a stand-alone, basic channel operation followed by the body of the case. The Go compiler does this transformation as an optimization.

Figure 3.4 shows an example of a select statement without a default clause. The program either sends a file path via `filesChan` or receives via `abortChan`. In Go, unlike C, there is no implicit fall-through for cases in select and switch statements. Therefore, the `break` statement on line 4 refers to the enclosing loop, only shown in Figure 1.1, and not to the select case. And if sending via `filesChan` is the executed case, the program continues after the select statement.

```
1 select {
2     case filesChan <- file.Name():
3     case <-abortChan:
4         break
5 }
```

Figure 3.4: Select statements with different cases and without a default clause (extract from Figure 1.1).

Figure 3.5 shows an example of a select statement with a default clause. Here, the program can either immediately read from `abortChan` (because it was closed) and returns from the function, or otherwise enters the empty default clause and continues after the select statement.

```
1 select {
2 case <-abortChan:
3     return
4 default:
5 }
```

Figure 3.5: Select statements with one case and a default clause (extract from Figure 1.1).

In order to understand the full semantics of select statements, it helps to look at the runtime implementation of channels and select statements[2]. Channels are implemented as a

[2] see `https://golang.org/src/runtime/chan.go`

small data structure, containing a mutex and pointers to the channel buffer and sender and receiver queues among other things. A go-routine waiting to send or receive can add itself to the corresponding queue and then sleep. If a different go-routine matches the attempted operation, it can signal and wake all the sleeping go-routines. The go-routine to wake up first gets matched. In practice, the matches are random.

The runtime implements select statements without default clauses in two passes. In the first pass, when the select statement gets reached initially, the runtime looks at all cases in random order. If a channel operation can succeed immediately, the corresponding case gets executed. If no case can succeed immediately, the second pass gets reached. The runtime now adds the current go-routine as a pending sender or receiver to the queues of all channels in the select cases. The go-routine then sleeps, waiting to be signaled and woken up by a different go-routine. The implementation for select statements with a default clause is similar. However, instead of the second pass, if the first pass is unsuccessful the default clause gets executed.

This means that two go-routines with matching select cases, i.e. one go-routine trying to send, the other trying to receive on the same channel, in select statements that both have a default clause, can never synchronize. Neither go-routine has a way of knowing about the other because, during the first pass, neither go-routines modifies the channel (or its sender or receiver queues), which would indicate readiness to communicate to the other.

3.1.4 Experimental Analysis

In addition to studying the way the Go compiler and runtime code handle channels and select statements, I also wrote a set of 30 tests, in the form of short Go programs, to investigate semantics experimentally and determine under what circumstances channel operations can synchronize. The code for all of the tests is similar. The main function starts two separate functions in two different go routines. Then, the main function sleeps for a few seconds before the program exits. One of the two go routines attempts to send, the other to receive (in a loop).

There are five possible sender scenarios. The first is the stand-alone send. The other four are as a select case, with or without other select cases, and with or without a default clause. There are six possible receiver scenarios. The first five are the converse of the five send operations. The sixth is receiving via a loop ranging over a channel. This scenario was mainly added to confirm the semantic equivalence with unconditional loops with the receive statement inside the loop body, as discussed in Section 3.1.2.

Figure 3.6 shows the test code for a sender using a select statement with two cases and no default clause, and a receiver using a select statement without another case but with a default clause.

```go
func main() {
    chA := make(chan int)
    chB := make(chan int)
    close(chB)

    go func() {
        for {
            select {
            case chA <- 42:
            case <-chB:
                time.Sleep(50 * time.Millisecond)
                fmt.Println("send failed")
            }
        }
    }()
    go func() {
        for {
            select {
            case i := <-chA:
                fmt.Println(i)
            default:
                time.Sleep(50 * time.Millisecond)
                fmt.Println("receive failed")
            }
        }
    }()
    time.Sleep(1 * time.Second)
    fmt.Println("done")
}
```

Figure 3.6: Example test code for select statements without synchronization via `chA`.

There are several ways to implement a second case in a select statement. Obviously, the second case should be executable, so using a channel operation that will never match is not useful. The easiest way to ensure a select case can always execute is to read from a closed channel. As discussed above, this can always succeed immediately. A second option is to write to a buffered channel with remaining capacity, which can also always succeed immediately. A third option is to implement an operation that can synchronize with a different go-routine elsewhere.

The results of the experimental analysis are shown in Table 3.1. The results reflect and confirm the runtime implementation of channels and select statements described above. A select statement with a single case behaves exactly as if a stand-alone communication statement. A range-loop over a channel behaves exactly as the equivalent unconditional loop with a read and conditional break in its body. Select statements with two default

Table 3.1: Results from experimental analysis of channel synchronization semantics.

		Send	Send Case			
			No other Cases		With other Case	
			No Default	With Default	No Default	With Default
Receive		Synchronize	Synchronize	Synchronize	Synchronize	Synchronize
Receive Case	No other Cases: No Default	Synchronize	Synchronize	Synchronize	Synchronize	Synchronize
	No other Cases: With Default	Synchronize	Synchronize	Do not Synchronize	Synchronize in some Scenarios*	Do not Synchronize
	With other Case: No Default	Synchronize	Synchronize	Synchronize in some Scenarios*	Synchronize in some Scenarios*	Synchronize In some Scenarios*
	With other Case: With Default	Synchronize	Synchronize	Do not Synchronize	Synchronize in some Scenarios*	Do not Synchronize
Range Loop		Synchronize	Synchronize	Synchronize	Synchronize	Synchronize

clauses never synchronize.

There are five scenarios where the two go-routines may or may not synchronize, depending on the implementation of the other case(s) in the select statement(s). The two go-routines never synchronized when the other case was a receive operation via a closed channel or a send via a buffered channel with remaining capacity.
However, the two go-routines did sometimes synchronize if the other case also required communication. Figure 3.7 shows one example for this. In this modified scenario, which required removing sleep and print statements and adding a third go-routine, the two cases involving chA synchronize many times within the one second running time of the program.

```
1  func main() {
2      chA := make(chan int)
3      chB := make(chan int)
4
5      go func() {
6          for {
7              chB <- 69
8          }
9      }()
10     go func() {
11         for {
12             select {
13             case chA <- 42:
14             case <-chB:
15             }
16         }
17     }()
18     go func() {
19         for {
20             select {
21             case i := <-chA:
22                 fmt.Println(i)
23             default:
24             }
25         }
26     }()
27     time.Sleep(1 * time.Second)
28     fmt.Println("done")
29 }
```

Figure 3.7: Modified example test code for select statements to enable synchronization via chA.

The results involving select statements confirm the described runtime implementation. If the

other case is a receive operation via a closed channel or a send operation via a buffered channel with remaining capacity, that case will always be selected in the first pass. Only by introducing a synchronizing go-routine that also uses a select statement is it possible to get to the second pass.

3.2 Channel Model and Operations in Uppaal

Since the focus is on concurrency, it is not very interesting to look at the values transferred via channels. In most cases, channels transfer values or data that gets generated and processed by several go routines. While it is entirely possible in Go to have a channel that transports other channels, possibly as part of a larger data structure, it is rare in practice. Therefore, the decision was made not to model the values that get transferred via channels. This helps to significantly simplify the channel model.

3.2.1 Channel States

The Uppaal model for Go channels uses just two integer variables per channel: `counter` and `buffer`. `buffer` holds the size of the channel buffer. For unbuffered channels, `buffer` is 0. When the channel gets closed, `buffer` gets set to -1 to indicate this.
`counter` is initially 0. A new sender increases `counter`. A new receiver decreases `counter`. When there are pending receivers, `counter` is negative. When there are pending senders, `counter > buffer`. When `counter` is in the range `0 <= counter <= buffer`, there are no pending senders or receivers and the channel buffer is empty, partially filled, or full. Figure 3.8 visualises the state of the channel depending on which of the three possible ranges `counter` is in.

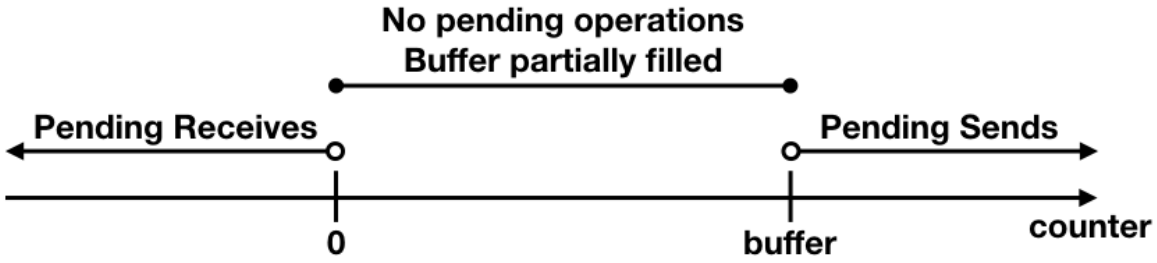


Figure 3.8: Channel state depending on counter value.

If the channel is unbuffered, `buffer` is 0, so `counter` has to be 0 when there are no pending senders or receivers. As before, a negative `counter` value indicates pending receivers. A positive `counter` indicates pending senders.

As a result of this definition, the effects of a sender synchronizing with a receiver cancel

each other out. If there are pending receivers and `counter` is negative, a new sender increases `counter` and synchronizes with and removes one pending receiver. If there are pending senders and `counter > buffer`, a new receiver decreases `counter` and synchronizes with and removes one pending sender.
In Go, a sender that does not synchronize with a receiver would add the transferred value to the channel buffer. In the Uppaal model this just means increasing `counter`. Conversely, in Go, a receiver that does not synchronize with a sender would remove the transferred value from the channel buffer. In the Uppaal model this just means decreasing `counter`.

3.2.2 Channel Process

In Uppaal, Go channels get modeled as a *Channel* process. Each process instance of *Channel* corresponds to one Go channel created with the built-in `make` function. Each instance has its `counter` and `buffer` as described above. *Channel* (modeling Go channels) deals with five Uppaal channels: `sender_trigger`, `sender_confirm`, `receiver_trigger`, `receiver_confirm`, and `close`.

In order to index and differentiate *Channel* instances, *Channel* has an index parameter *i*. `counter`, `buffer`, and the five Uppaal channels `sender_trigger`, `sender_confirm`, `receiver_trigger`, `receiver_confirm`, and `close`, are implemented as arrays of integers or Uppaal channels respectively, in the global system declarations. This is necessary so that all parts of a system can access them.
Each *Channel* instance can access its variables and Uppaal channels by accessing the *i*-th element of each array. Other processes have to store the integer indices of the *Channel* instances they deal with. This corresponds to channel variables in Go which are internally just pointers to an underlying structure in the Go runtime.
In Go, the built-in `make` function creates new channels. In Uppaal, all *Channel* instances exist from the start. A counter keeps track of how many *Channel* instances are already in use. The function `make_chan` in the global system declarations returns the next, previously unused index. It also initialises the `counter` and `buffer` used by the *Channel* instance.

Figure 3.9 shows all global declarations for channels. This system can simulate up to 10 *Channel* instances. Section 4.3.1 discusses how the required number of *Channel* instances can be determined.

Figure 3.10 shows the full *Channel* process state diagram. The process starts in the `idle` state. Only the `idle`, `closed`, and `bad` state are not committed.

Channel instances are responsible for the synchronization and blocking of other processes trying to send or receive. The two trigger channels, `sender_trigger` and `receiver_trigger` are used by regular processes to request confirmation for an operation.

```
int chan_count = 0;
int chan_counter[10];
int chan_buffer[10];
chan sender_trigger[10];
chan sender_confirm[10];
chan receiver_trigger[10];
chan receiver_confirm[10];
chan close[10];

int make_chan(int buffer) {
    int cid = chan_count;
    chan_count++;
    chan_counter[cid] = 0;
    chan_buffer[cid] = buffer;
    return cid;
}
```

Figure 3.9: Global declarations for channels in Uppaal.

Operations get confirmed by a *Channel* instance via the `sender_confirm` and `receiver_confirm` channels.

Depending on the state of a *Channel* instance (`counter` and `buffer`), when the triggering process is a pending sender or receiver, it might not be possible to confirm an operation immediately. Then, the trigger failed. In that case, the triggering process is forced to wait in its current state for confirmation.
There are two options when the trigger succeeds. If there are previously pending senders or receivers, two confirmations get sent: one for the triggering process and one for a previously pending process, i.e. one for the sender and one for the receiver. Any previously pending process can receive confirmation in Uppaal. This leads to different traces.
If there are no previously pending senders or receivers, which corresponds to an operation on the channel buffer, only a single confirmation for the triggering process needs to be sent.

This behaviour can be seen in the lower half of Figure 3.10. The lower left part shows the behaviour when a sender sends a trigger. The lower right part shows the mirrored behaviour when a receiver sends a trigger. All states that deal with triggers are committed, which means triggers either fail or succeed immediately, including all confirmations. The *Channel* process returns to its `idle` state before any process not involved in the trigger or a confirmation can take any transition. This makes *Channel* operations atomic with respect to regular processes.

The upper half of Figure 3.10 deals with closing channels. A *Channel* process attempts to enter its closed state when a regular process sends a signal via the `close` channel. Under

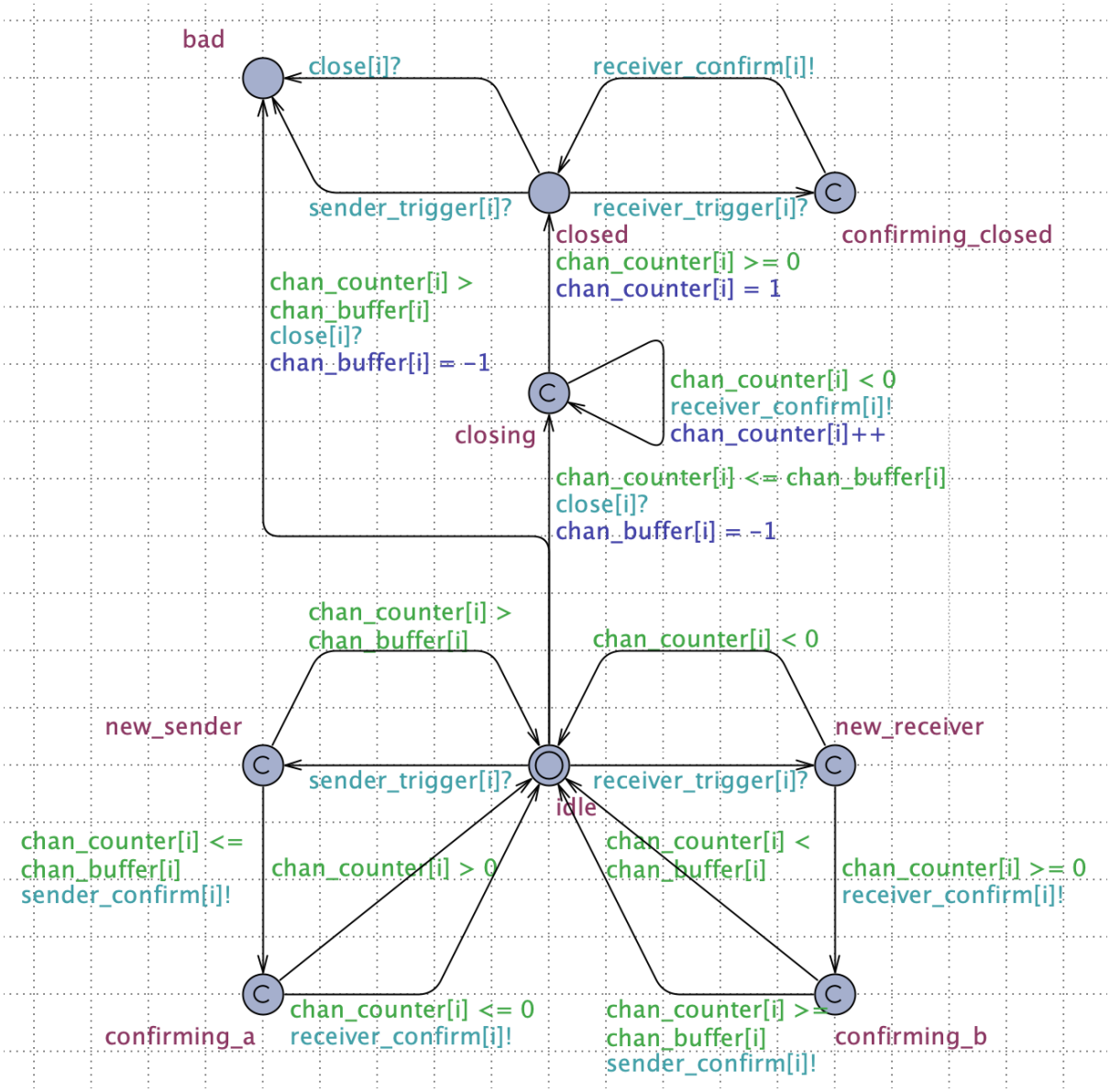


Figure 3.10: *Channel* process in Uppaal.

certain circumstances, corresponding to runtime panics in Go, a *Channel* may reach its `bad` state.

When a *Channel* instance is in the `idle` state and receives a signal via `close`, two things can occur. If there are pending senders (`counter > buffer`), the Go runtime would panic since, so the *Channel* instance transitions into its bad state.
Otherwise, a committed `closing` state gets reached. In this state, the *Channel* instance sends confirmations to all pending receivers. When there are no (more) pending receivers, the *Channel* instance reaches its `closed` state.

When a *Channel* instance is in the `closed` state and receives another signal via `close`, this corresponds to two calls to the built-in `close` function in Go. In that case the Go runtime would panic. Accordingly, the *Channel* instance reaches its `bad` state.

A *Channel* instance in the `closed` state also has to respond to triggers. If a sender sends a signal via the `sender_trigger` channel, this indicates a send attempt on a closed channel and therefore causes the *Channel* instance to reach its `bad` state.
If a receiver sends a signal via the `receiver_trigger` channel, confirmation can always be sent immediately. In Go, receiving from a closed channel always succeeds immediately.

3.2.3 Simple Channel Operations

Based on the previous description of the *Channel* process, the four simple channel operations, making a channel, sending and receiving via a channel (outside of a select case), and closing a channel, are implemented in Uppaal processes with the four state transition diagrams shown in Table 3.2.

In the Uppaal state diagrams in Table 3.2, `cid` is an integer variable and part of the local declarations of a process. Senders and receivers increment or decrement the `counter` used by a *Channel* instance with the same transition that sends the trigger signal. Once a trigger is sent, a sending or receiving process is stuck until it receives a confirmation signal.

3.2.4 Select Statements

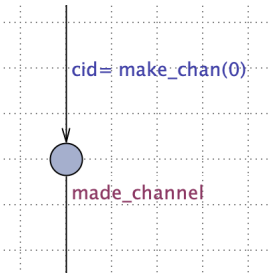
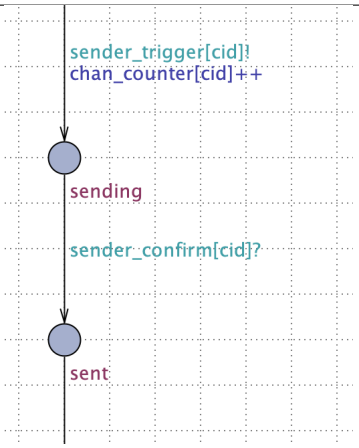
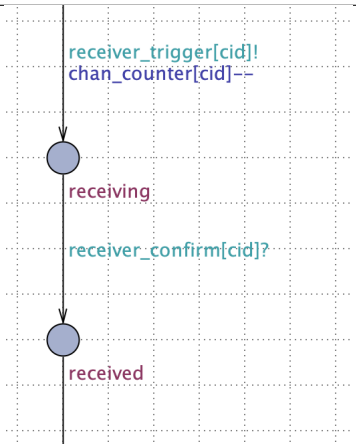
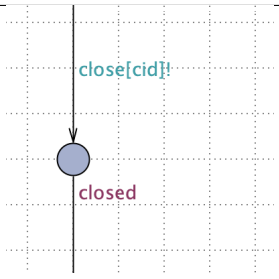
The Uppaal state diagrams for select statements are more complicated since they have to correctly model all the different scenarios described in Section 3.1.4. Synchronization of two select cases can occur in some scenarios but not in others. When a default clause is present, if no other case can be executed immediately, the default clause gets executed. When no default clause is present, the select statement can cause the go routine to block.

The state diagrams for select statements are inspired by the two pass implementation the Go runtime uses (see Section 3.1.3). Figures 3.11 and 3.12 show two examples of Go code with select statements and the corresponding Uppaal state diagrams.

The incoming transition in Uppaal increases or decreases the `counter` variables for all *Channel* instances referred to by the different select cases. In Figure 3.11, because the first case attempts to send (`chA <- 42`), the corresponding `counter` gets increased (`chan_counter[cidA]++`). Similarly, because the second case attempts to receive (`<-chB`), the corresponding `counter` gets decreased (`chan_counter[cidB]--`).

The incoming transition leads to the `select_pass_1` state. As the name implies, this state corresponds to the first pass in the runtime implementation. The first pass in the runtime immediately either selects a case or goes to the second pass or default clause (if present). To model this behaviour `select_pass_1` is committed. A system reaching `select_pass_1` has to leave it immediately.
The first pass can only select cases with channel operations that can immediately succeed.

Table 3.2: Uppaal state transitions corresponding to basic channel operations.

Operation	Go code	Uppaal state transitions
Make	`ch := make(chan int)`	
Send	`ch <- 42`	
Receive	`x := <-ch`	
Close	`close(ch)`	

```
select {
case chA <- 42:
// ...
case <-chB:
// ...
}
```

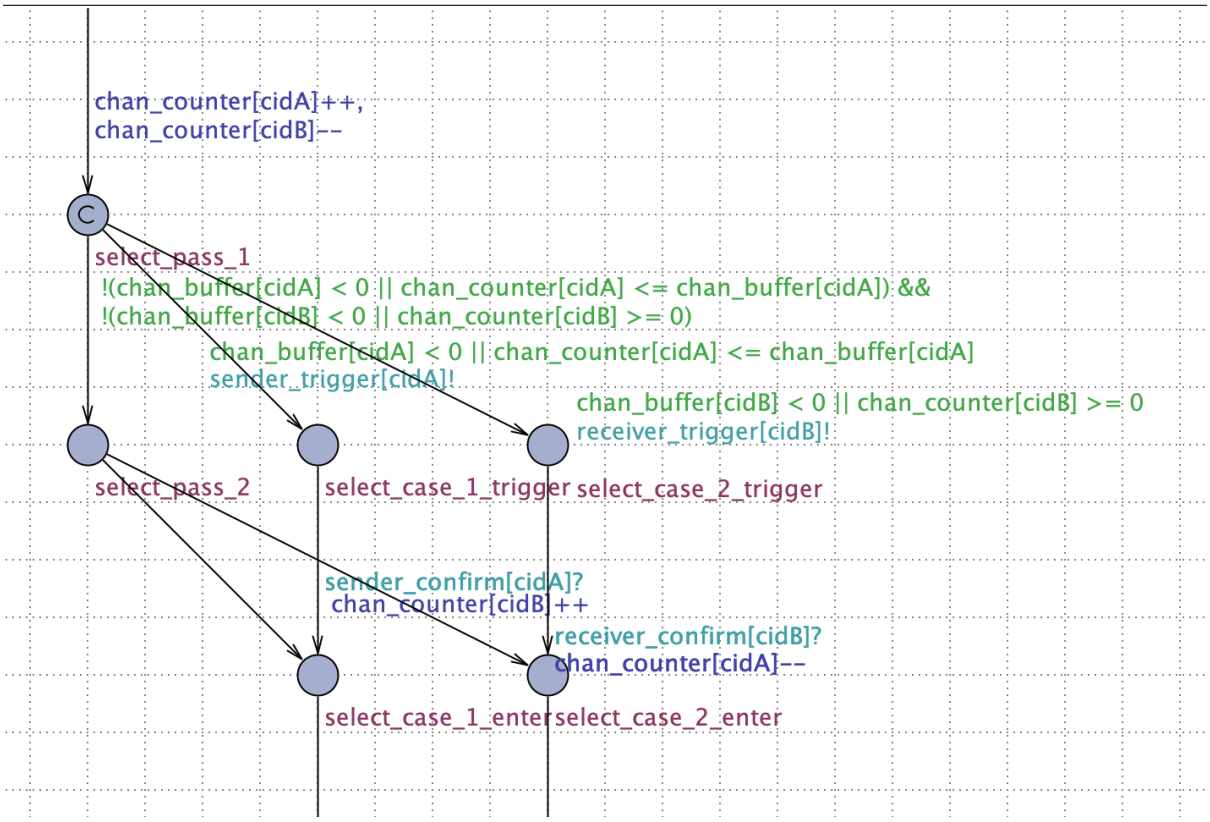


Figure 3.11: Go code and Uppaal state diagram for select without default clause.

The transitions from `select_pass_1` to the individual cases ensure this through guards. If a case tries to send via a channel, the channel must either have pending receivers or remaining buffer space. Conversely, if a case tries to send via a channel, the channel must either have pending senders or remaining elements in its buffer. Cases working with closed channels are always possible since reading via a closed channel immediately succeeds and writing via a closed channel causes the runtime to panic (leads to the `bad` state of a *Channel* process).

In Figure 3.11 the transition from `select_pass_1` to `select_case_1` has the guard `chan_buffer[cidA] < 0 || chan_counter[cidA] <= chan_buffer[cidA]`. The case is possible if the *Channel* instance with index `cidA` is closed or has pending receivers or free buffer space.

Because any case selected from `select_pass_1` has to succeed immediately, edges to the specific cases in Uppaal send a corresponding trigger signal to the *Channel* instance and then await confirmation. The guards described above in combination with the internals of

```
select {
case <-chA:
// ...
case <-chB:
// ...
default:
// ...
}
```

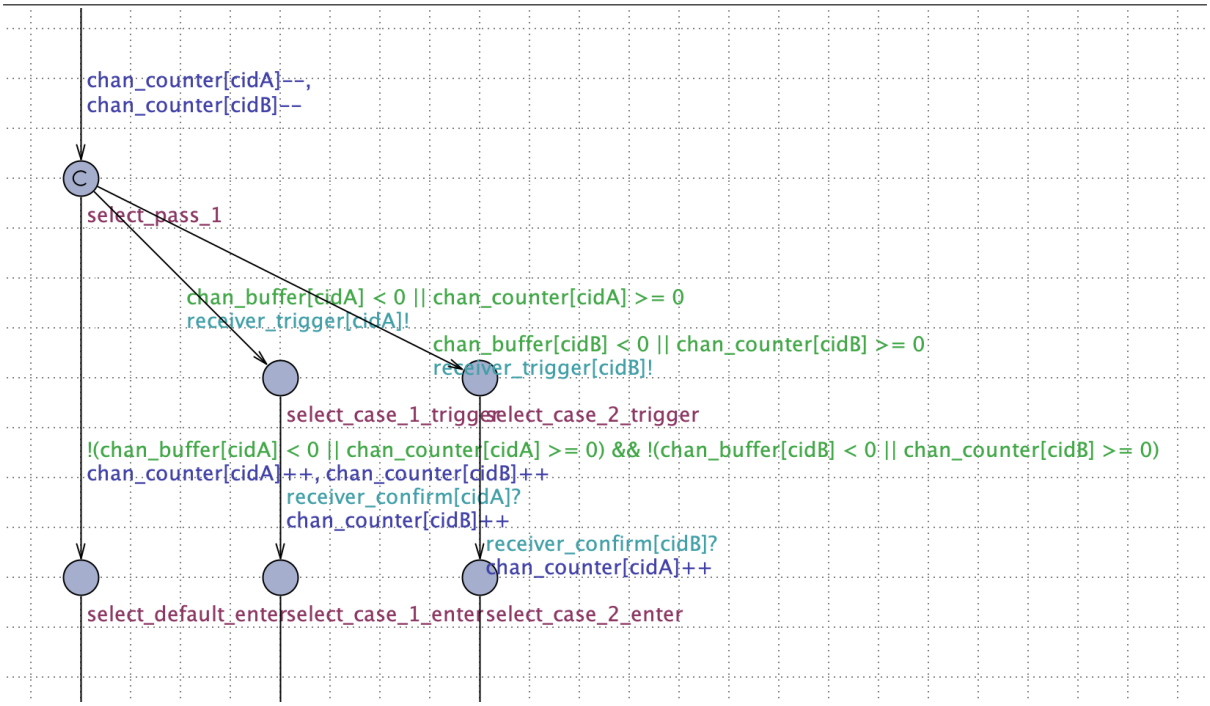


Figure 3.12: Go code and Uppaal state diagram for select with default clause.

the *Channel* process described in Sections 3.2.2 ensure that the triggering process is the only process waiting for confirmation for the triggered operation and that the confirmation gets sent immediately (because the responsible states in the *Channel* process are committed).
The transition from `select_pass_1` to `select_pass_2` or `select_default_enter` (if a default clause is present) also has a guard. The second pass or default clause can only be reached if none of the select cases can perform their channel operations immediately. The transition is therefore only enabled when all other transitions out of `select_pass_1` are disabled and the transition guard is set accordingly.

If the select statement has no default clause and the first pass failed because no case could immediately succeed, `select_pass_2` gets reached. During this pass, the runtime registers the current go-routine as a pending sender or receiver for all channel operations. The change of the corresponding `counter` variables in Uppaal already happens through the updates of the incoming edge, leading to `select_pass_1`.
After registering the go-routine with all channels, the runtime lets the go-routine sleep,

waiting for a different go-routine to attempt a matching channel operation or for a channel to close. A Uppaal process has to wait in the `select_pass_2` state in the same way. The edges out of the state all require synchronization via one of the confirmation channels. When a *Channel* instance, triggered by another process, signals confirmation, the corresponding select case can be entered.

When the default clause or any of the select cases get entered, the `counter` variables of all the channels via which communication did not occur need to be reverted.
In Figure 3.11, the first case can be entered (`select_case_1_enter`) either from `select_case_1_trigger`, when the case succeeded immediately in the first pass, or from `select_pass_2`, when the process had to await confirmation. In both cases, the transitions leading into `select_case_1_enter` increment the `counter` for the other select case: `chan_counter[cidB]++`. This undoes the decrement at the start.
In Figure 3.12, the transition leading to the default clause (`select_default_enter`), undoes changes to both `counter` variables.

3.3 Control Flow in Uppaal

As mentioned in the introduction to the chapter, the actual data that determines control flow in a Go program does not get modelled in Uppaal. Instead, in the model all possible control flow paths, agnostic to the processed data, are considered.

3.3.1 If Statements

An if statement in Uppaal starts with a process state with two outgoing edges to the `enter_if` start state of the if branch and the `enter_else` start state of the else branch. Following the two branch start states are the modeled statements from the original program. At the end, both branches lead back to an `exit_if` state from which the modelled program continues. There are no guards, synchronisation requirements, or updates associated with these transitions so that a Uppaal process can always take either path.
If the condition of an if statement contains function calls, channel operations or other constructs that are modelled in Uppaal, these appear immediately before the if statement.

Figure 3.13 shows a very simple if statement inside of a function and the corresponding model of the if statement in Uppaal. Since this if statement does not have an else branch, `enter_else` directly connects to `exit_if`.

```go
func test() {
    chA := make(chan int)
    if 42 == 24 {
        chB := make(chan int)
        close(chB)
    }
    close(chA)
}
```

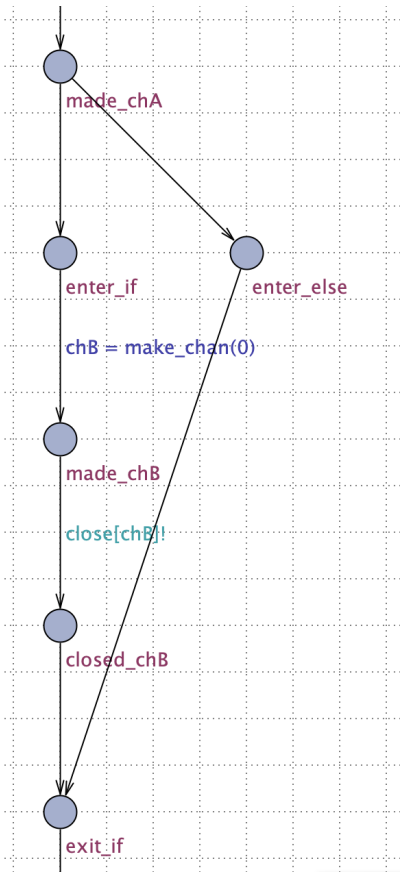


Figure 3.13: Go code and Uppaal state diagram for a simple if statement.

3.3.2 Regular Loops

Loops follow the same general pattern as if statements and use the five states: `enter_loop_cond`, `exit_loop_cond`, `enter_loop_body`, `exit_loop_body`, and `exit_loop`. Everything modeling the loop body is between the `enter_loop_body` and `exit_loop_body` states. From the `exit_loop_body` state, the program always returns to the `enter_loop_cond` state, with which it also enters the loop.

In Go, a regular loop can have an initialisation statement, a loop condition, and an increment statement, very much like in C programs.
The initialisation statement, if present, gets executed only once at the start of the loop. If it contains function calls, channel operations or anything else modeled in Uppaal, the model states and transitions appear immediately before the loop. `enter_loop_cond` is the first state of the actual loop that a Uppaal process reaches.
The loop condition, if present, gets evaluated at the start of each loop iteration. If it evaluates to false, the loop exits, otherwise the the loop body gets executed. Therefore, anything modeling interesting behaviour in the loop condition is placed at the start of the loop, between the `enter_loop_cond` and `exit_loop_cond` states.

There are two transitions out of the `exit_loop_cond` state - one to the `enter_loop_body` state, and one to the `exit_loop` state - corresponding to either executing the loop body or continuing after the loop in the original Go program. As with if statements, the choice is non-deterministic.
If no loop condition is present, the loop is infinite. In the corresponding Uppaal model, the transition from the `exit_loop_cond` state to the `exit_loop` state is missing for infinite loops.
And finally, the increment statement of a loop, if present, gets executed every time the loop body completes one iteration and the loop condition is about to be reevaluated again. Therefore, if the increment statements includes any interesting behaviour, this is modeled at the end of the loop body, immediately before the `exit_loop_body` state.

Figure 3.14 shows a function with a very simple for loop and the corresponding Uppaal model of the loop. Neither the initialisation statement (`i := 0`), nor the loop condition (`i < 5`), nor the increment statement (`i++`) contain any interesting behaviour that gets modelled in Uppaal. Therefore, the `enter_loop_cond` and `exit_loop_cond` states connect directly.

As already mentioned in Section 3.1.2, Go also offers range loops as a second kind of loop. Range loops take an aggregate data structure - an array, a slice, a map, or a channel - and provide one entry at a time via a loop variable to the loop body for processing. Generally, range loops are syntactic sugar and can be substituted for slightly longer code using only regular loops. The only exception are range loops over maps, because maps are unordered and do not offer successive indices.
Therefore, in Uppaal, all range loops that do not deal with channels can be modeled like regular loops without any interesting behaviour in their loop condition. Loops ranging over channels are more complicated and discussed separately in Section 3.3.3.

Because it is very useful for model checking in Uppaal to limit the number of loop iterations, Toph reads annotations for minimum and maximum iterations of loops from comments in Go code. In Uppaal, loops with such annotations use a variable (part of the local process declarations) that counts loop iterations, while transition guards enforce the iteration bounds.

`Helper 2` in *WordCounter* contains such an annotated loop. Figure 3.15 shows the loop in question as well as the top of the loop in the corresponding Uppaal process with the enforced the iteration bounds.

3.3.3 Range Loops over Channels

The Uppaal model for loops ranging over channels looks very similar to the model for regular loops. The main difference is that the loop condition gets replaced with the states and

```go
func test() {
    ch := make(chan int, 5)
    for i := 0; i < 5; i++ {
        ch <- i
    }
    close(ch)
}
```

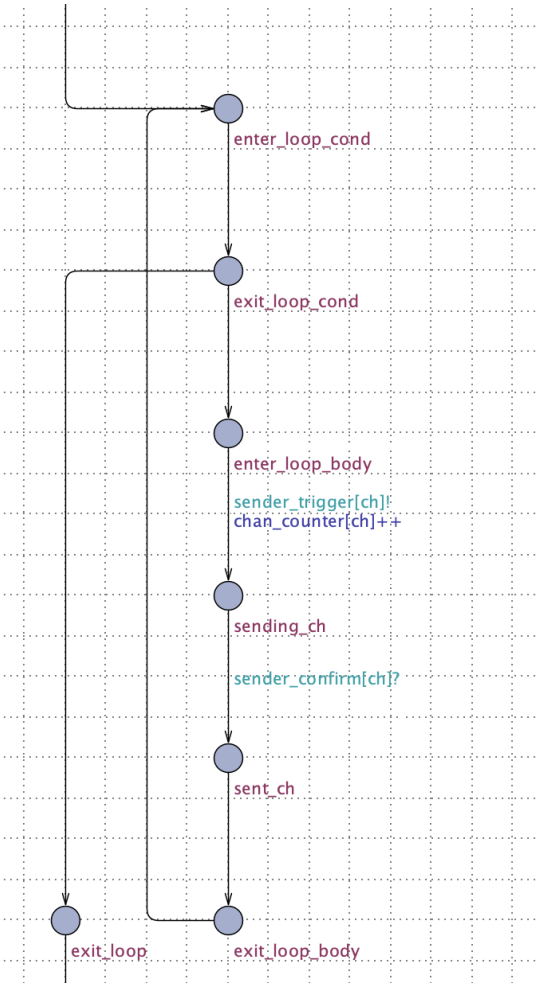


Figure 3.14: Go code and Uppaal state diagram for a simple for loop.

transitions for a channel receive operation. There are also guards on the transitions to enter the loop body or exit the loop that depend on the channel receive operation.

When a Uppaal process reaches a loop ranging over a channel, it first reaches the `range_enter` state (similar to the `enter_loop_cond` state in regular loops). The following two transitions to receive via the modeled Go channel are almost the same as for a standalone channel receive operation (discussed in Section 3.2.3). However, in range loops the states are called `range_receiving` and `range_received` and there is one additional update that happens with the triggering transition. After the process decrements the channel counter value for the modeled Go channel by one to register itself as a receiver (as usual), it also stores whether the counter value remained greater than or equal to zero with that operation. This is stored in the local boolean variable `ok` and very important information. In Go, the assignment `x, ok := <-ch` stores in the `ok` boolean variable whether the received value is valid or the default value. The `ok` variable in Uppaal is named accordingly.

As described in Section 3.1.2, a range loop over a channel still executes its loop body if the

```
1 // toph: min_iter=2, max_iter=2
2 for i := 0; i < 2; i++ {
3     <-waitChan
4 }
5 close(doneChan)
```

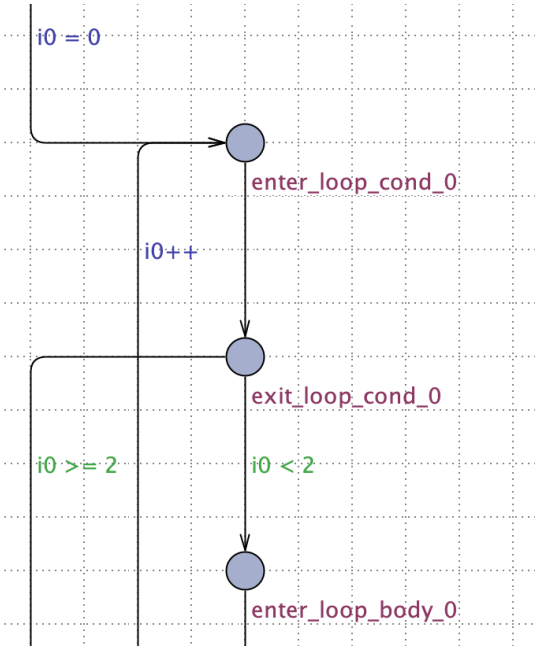


Figure 3.15: Go code and Uppaal state diagram for a for loop with iteration bounds.

channel is already closed but the received value is from the channel buffer and not the default value of the channel type. It would be inaccurate in the Uppaal model to simply check if a channel is closed to determine if the loop should be exited.
Instead, one of four scenarios can be the case when a modeled range loop gets confirmation for a received value and reaches the `range_received` state:

- The modeled Go channel is still open.
  This means that there was either a buffer element to read or a matching sender. In this case a valid value was received and the loop body gets entered.
- The modeled Go channel is closed but was open when the process with the range loop attempted to receive.
  Therefore, the process with the range loop was a pending receiver and had to wait in the `range_receiving` state when the channel was still open. The Uppaal Channel process sent confirmation for the receive operation when the channel got closed. In this case, the channel counter value in Uppaal had to be less than zero after the process with the range loop decremented it. This corresponds to receiving a default value and exiting the loop.
- The modeled Go channel is closed and was closed when the process with the range loop attempted to receive. There were buffer elements to read for the process with the range loop.
  In this case, the channel counter value in Uppaal had to be at least zero after the process with the range loop decremented it. This corresponds to receiving a valid

value out of the buffer of a closed channel and entering the loop body.

- The modeled Go channel is closed and was closed when the process with the range loop attempted to receive. There were no buffer elements to read for the process with the range loop.
  In this case, the channel counter value in Uppaal had to be less than zero after the process with the range loop decremented it. This corresponds to receiving a default value and exiting the loop.

This is where the aforementioned `ok` variable comes in. It is necessary to distinguish the third scenario from the second and the fourth. In the Uppaal model for a range loop over a channel `ch`, the guard on the transition to exit the loop is `chan_buffer[ch] < 0 && !ok` and the guard to enter the loop body is `chan_buffer[ch] >= 0 || ok` (the boolean complement). `chan_buffer[ch] >= 0` is the boolean expression to check if the modeled Go channel is still open. `ok` stores if the channel counter variable became negative when the process with the range loop decreased it.
It would not be sufficient to only look at the value of the counter variable relative to zero after the receive operation gets confirmed by the *Channel* process. The reason for this is subtle and can be explained by going back to the *Channel* process shown in Figure 3.10. When a channel with pending receivers gets closed, the *Channel* process increments its counter variable for each receiver getting confirmation. The last pending receiver would see a counter variable value of zero. This would be indistinguishable from a counter variable value of zero after a valid value was received and the counter was decremented.

Figure 3.16 shows the code of the `countWords` function from *WordCounter* and the top of the loop in the Uppaal model for it. The states and transitions for everything in the loop are below the `enter_loop_body_0` state.

3.4 Functions and Goroutines in Uppaal

Each function in Go translates to one process in Uppaal. Each function invocation in Go programs translates to one process instance in Uppaal.

Go treats functions as first-class citizens, meaning they can be stored in variables and passed as function arguments and results like any other type. One of the test cases even contains a map from strings to functions[3]. This challenges the idea that data and control flow are generally separable and that it suffices to only model control flow.
It was decided that it would be far too difficult to model functions as first-class citizens and dynamic function dispatch in Uppaal. Instead, the callee of each function call has to be easily and statically determinable.

[3]Toph generates several warnings for this program.

```go
func countWords(filesChan chan string, wordCountsChan chan int) {
    for file := range filesChan {
        select {
        case <-abortChan:
            return
        default:
        }
        content, err := ioutil.ReadFile(file)
        if err != nil {
            errChan <- err
            break
        }
        text := string(content)
        count := strings.Count(text, " ")
        wordCountsChan <- count
    }
}
```

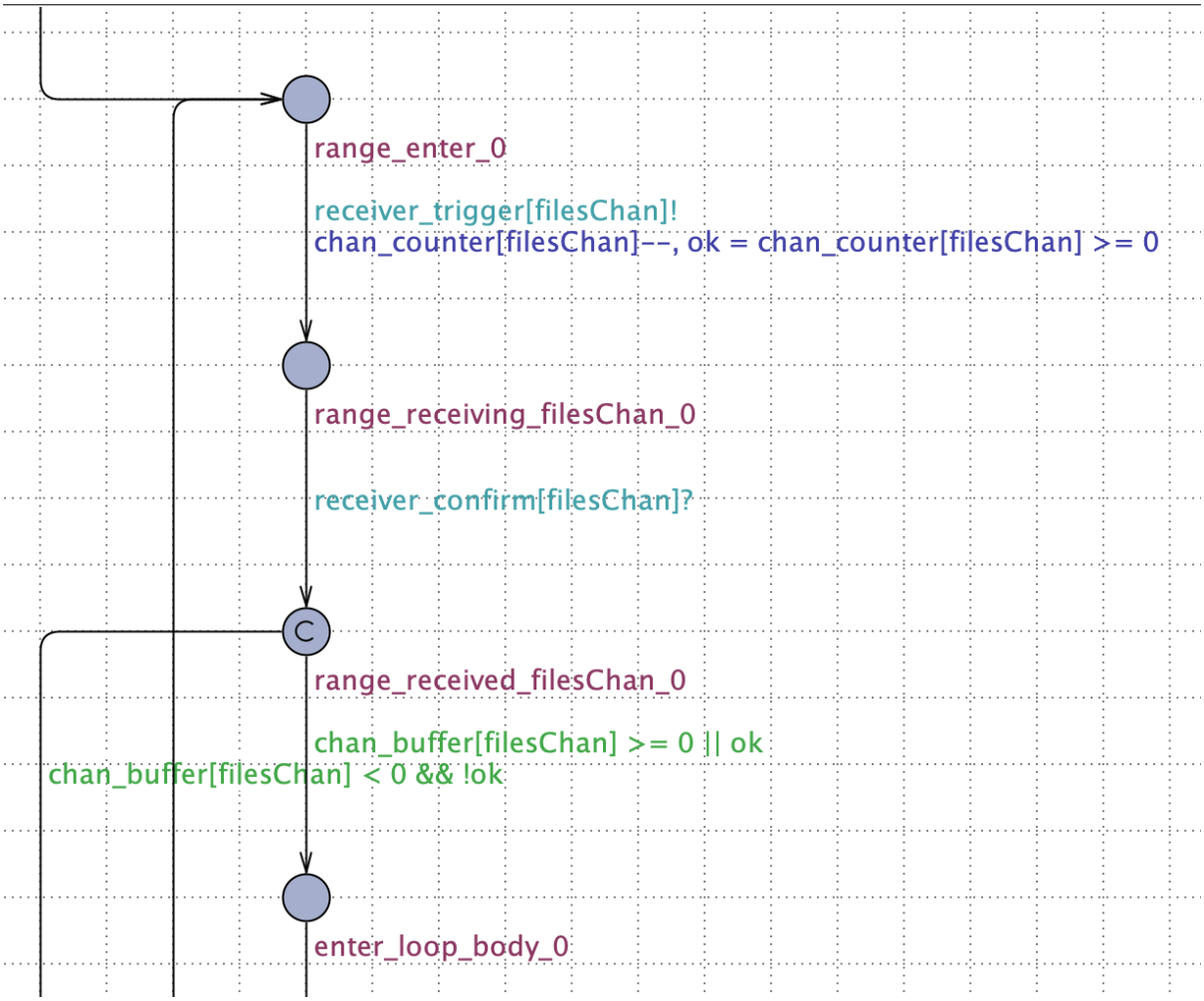


Figure 3.16: `countWords` function (excerpt from Figure 1.1) and the corresponding range loop state diagram.

In the devised model, different function process instances are identified by their process identifier (`pid`), an integer value. Similar to the `make_chan` function in Uppaal, which designates different *Channel* process instances, function process instances and their `pids` are managed by `make_...` functions too. For each function process, a counter keeps a record of how many instances are already in use.

3.4.1 Function Calls and go Calls

Each function process in Uppaal starts with a `starting` and a `started` state and ends with an `ending` and an `ended` state. Uppaal has no notion of processes starting other processes. Therefore, all function processes, except for the `main` function, require synchronisation via a Uppaal channel to transition from their `starting` state to their `started` state. This synchronisation represents a function call or `go` call, spawning a new Goroutine. With `go` calls, both the caller and callee (the new Goroutine) can continue in parallel. However, with regular function calls, the caller has to wait until the callee completes execution. This is also achieved through synchronisation via a Uppaal channel when the callee transitions from its `ending` to its `ended` state.
Each function process instance has two Uppaal channels associated with it. The `async` channel is used to model `go` calls. The caller sends, and the callee receives once to get the callee started. The `sync` channel is used to model regular function calls. As with `go` calls, at the start the caller sends, and callee receives once to get the callee started. At the end the callee sends, and the caller receives once to enable the caller to continue. While the callee executes, the caller is stuck waiting. The callee uses a locally declared variable to remember if it was called as a regular function call or as a `go` call.

Figure 3.17 shows the global declarations used by the `countWords` process instances. Because the function gets called twice during the entire run of the program, the `async_countWords` and `sync_countWords` Uppaal channel arrays have size 2.

```
1 int countWords_count = 0;
2 chan async_countWords[2];
3 chan sync_countWords[2];
4
5 int make_countWords() {
6     int pid = countWords_count;
7     countWords_count++;
8     return pid;
9 }
```

Figure 3.17: `make_countWords` function and global process declarations in Uppaal.

Figure 3.18 shows the `starting`, `started`, `ending` and `ended` states of the `countWords` process in Uppaal, including the different Uppaal channels used for regular function calls and

go calls.

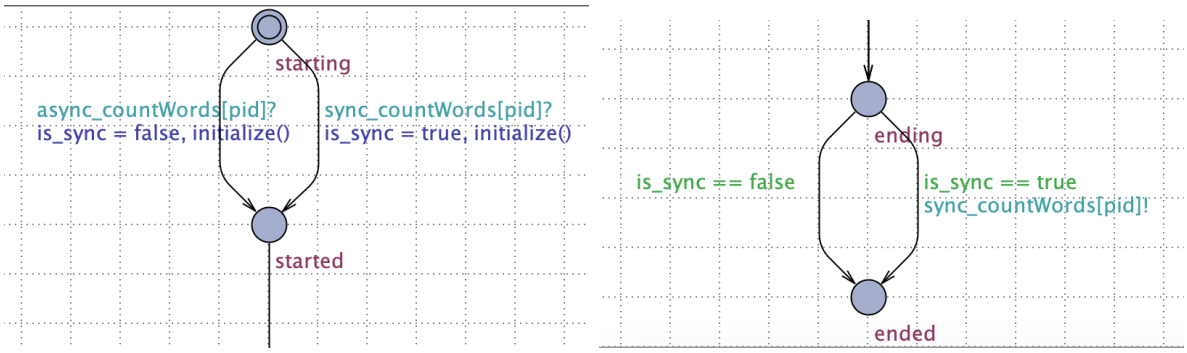


Figure 3.18: Initial and final states of `countWords` process in Uppaal.

3.4.2 Arguments and Results

In Go, functions can pass channels to each other through arguments and return values. In Uppaal, the arguments and results of functions are exchanged via arrays in the global system declarations, so that they are accessible to all processes. Like the arrays for the `async` and `sync` Uppaal channels, these arrays are also indexed by the `pid` of the callee. To call a callee with arguments, a caller first obtains a previously unused `pid` of a callee process instance through the `make_...` function. The caller then writes the values it wants to pass as arguments into the global argument arrays of the callee at the index of the new `pid`. And finally, the caller starts the callee, as discussed before. The callee initializes some of its local variables by copying values form the global argument arrays. The same happens in reverse when a callee returns values to a caller.

Figure 3.19 shows how `Helper 1` calls and passes arguments to `countWords` via the global argument arrays. Figure 3.20 shows the definitions of the global argument arrays and how the `countWords` process copies arguments into its local variables.

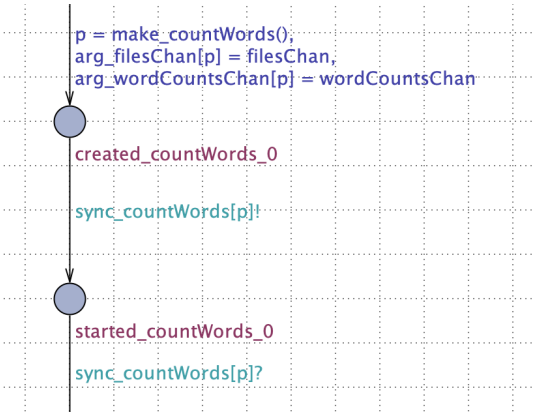


Figure 3.19: `Helper 1` calling `countWords` in Uppaal[4].

[4] In *WordCounter*, `filesChan` and `wordCountsChan` are captured variables. This is not reflected here to simplify the diagram.

```
int arg_filesChan[2];
int arg_wordCountsChan[2];
```

```
int filesChan = -1;
int wordCountsChan = -1;

void initialize() {
    filesChan = arg_filesChan[pid];
    wordCountsChan = arg_wordCountsChan[pid];
}
```

Figure 3.20: Global argument arrays and local variables and initialisation function of `countWords` function process in Uppaal.

3.4.3 Captured Variables

In Go, anonymous inner functions like `Helper 1` and `Helper 2` in *WordCounter* can use local variables of their enclosing functions. In fact, `Helper 1` and `Helper 2` both use `waitChan` defined in the `main` function enclosing them. This makes `waitChan` a captured variable. In this case it would be possible to simply pass `waitChan` by value as an argument to both functions.
However, anonymous inner functions can write to captured variables of the enclosing function. In C/C++ terms, the variables effectively get passed by reference. This presents a problem because Uppaal does not offer pointers that are capable of modeling this easily. Normally, channel variables in Go programs get modeled as local variables of the corresponding function process in Uppaal. It is not possible to pass pointers to these variables to other Uppaal processes. Therefore, all captured variables are instead modeled as arrays in the global declarations of a system. As with arguments and results, each function process instance can access its variables by using its `pid` as the array index.

This means that anonymous inner functions need to know the `pid` of their enclosing function invocation, which can simply be passed by value like all other function arguments and gets stored in a global `par_pid` array for each anonymous inner function. When an anonymous inner function needs to access a captured variable, it first determines the `pid` of its enclosing function, which then gets used as an index into the global array for the captured variable. For example, when `Helper 2` closes `doneChan` (a captured variable), the transition update is: `close[doneChan[par_pid_main_helper2[pid]]]!`. This approach can even be chained for nested anonymous inner functions.
If functions were supported as first-class citizens, anonymous inner functions would have to be curried, setting the parent `pid` before passing a reference to the inner function to other functions which provide the remaining arguments. This would make it complicated to call the same stored anonymous inner function multiple times.

4 Implementation

I developed Toph as the translation tool that automatically generates Uppaal systems for Go programs, implementing the methodology discussed in Chapter 3.

Toph consists of several parts that work in sequence. Figure 4.1 illustrates this pipeline.
First, Go code gets turned into an abstract syntax tree (AST) by the Go scanner and parser. These packages are available as part of the Go standard library and are also used by the Go compiler itself.
Then, Toph's builder component traverses the Go AST and builds Toph's own intermediate representation (IR) from it. This process extracts the important parts for modeling, such as channel operations, control flow and function calls, from the program and ignores data operations that do not get modeled.
Using the generated IR, Toph's translator component generates the final Uppaal system and all its processes, declarations, and queries. The results are written into files in Uppaal file formats.

Section 4.1 discusses the developed intermediate representation (IR) used in Toph. Section 4.2 explains the details of Toph's builder component. And finally, Section 4.3 covers the translation from Toph IR to Uppaal systems.

4.1 Intermediate Representation

Toph's IR, defined in the `ir` package, broadly follows a tree structure. The root is an `ir.Program` entity holding all `ir.Functions` and the global `ir.Scope`. An `ir.Scope` stores `ir.Variables` and can look up names. Each `ir.Function` contains an `ir.Body`. Each `ir.Body` has its own `ir.Scope` and a list of `ir.Stmts` (statements). Each `ir.Scope` has a pointer to its super scope, except for the global scope which has no super scope.

The following are all `ir.Stmts`:

- `ir.AssignStmt` represents a variable assignment.
- `ir.MakeChanStmt` represents creating a channel.

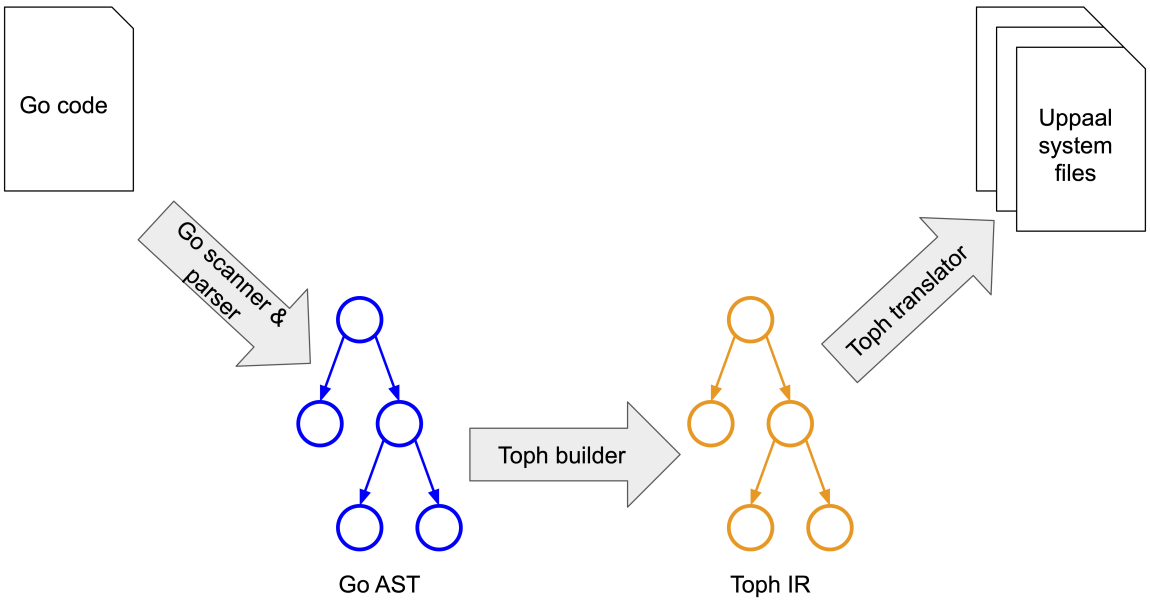


Figure 4.1: The different processing steps of Toph.

- `ir.ChanOpStmt` represents sending or receiving via a channel, or closing a channel.
- `ir.IfStmt` represents an if statement.
- `ir.ForStmt` represents a regular for loop or range loop not ranging over a channel.
- `ir.RangeStmt` represents a loop ranging over a channel.
- `ir.BranchStmt` represents a `break` or `continue` in a loop.
- `ir.SelectStmt` represents a select statement with all its different cases.
- `ir.CallStmt` represents a regular function call or a `go` call.
- `ir.ReturnStmt` represents a `return`, including the returned values.

Some of these, such as loops, have their own `ir.Bodys`. Some statements can reference variables stored in an `ir.Scope`.

Unlike the Go AST, Toph's IR has no notion of expressions, such as adding two values. An `ir.MakeChanStmt` for example, simply stores the channel variable that is assigned the newly created channel, as well as the buffer size of the new channel. The source of an `ir.AssignStmt` is an `ir.RValue`, which is either a plain value (an integer) or an `ir.Variable`. The destination is always an `ir.Variable`.

Each `ir.Variable` can be marked as captured, if it is defined in a function and used in an anonymous inner function of that function (see Section 3.4.3). Each `ir.Function` has a pointer to its enclosing function (possibly `nil`). This is used to detect captured variables

during lookup with `ir.Scope`.

Each `ir.Function` identifies its arguments and results by index. The `findFilesInFolder` function for example takes two arguments: a string and a channel. The string argument is not represented in the IR. But the corresponding `ir.Function` stores the channel argument at index 1. `ir.CallStmts` and `ir.ReturnStmts` use the same indexed mapping for arguments and results.

Each `ir.Variable` and `ir.Function` has a unique index that identifies it. This helps with debugging. The indices are also used in names in the generated Uppaal system to avoid naming conflicts.

4.2 Conversion of Go code to Toph Intermediate Representation

The `builder` package in Toph generates the intermediate representation. It uses the `go/token`, `go/parser`, `go/ast` and `go/types` packages[1] of the Go compiler to build an AST from the input Go program files. It builds the IR by traversing the AST.

Alternatively, it would have been possible to work with a static single assignment (SSA) form of Go using an internal package of the Go compiler[2]. Static single assignment is a very powerful and well understood representation used by modern optimizing compilers, but much closer to assembly code. All variables are turned into constants that get assigned once, hence the name. Functions are broken up into blocks. Control flow always begins at the start of a block. Jumping to other blocks or returning from a function is only possible at the end of a block. If a variable in the original code gets assigned different values in different branches, ϕ-nodes at the start of merging blocks define a new constant value depending on the incoming edge that was taken to the block.
SSA uses far fewer concepts than the original source code. In the process of translating from the AST to SSA, the Go compiler handles issues such as function argument evaluation order, replacing select statements with calls to runtime functions, and other semantics that have to be checked for manually with Toph's approach. For a kind of black box verification tool, which are the majority of those presented in Section 2.2 on prior work, it is easier to work with SSA.
However, it takes expertise to identify elements of the original program in SSA form. If statements, loops, and select statements are turned into connected blocks. By starting from the AST and not SSA form, Toph is able to build Uppaal system diagrams that are much closer to the original code. Loops and if statements are easily identifiable, which empowers

[1] see `https://golang.org/pkg/go/`
[2] see `https://golang.org/pkg/cmd/compile/internal/ssa/`

users to investigate and understand different simulation traces of a Uppaal system.
The main drawback of working with the AST is that there are a lot more language constructs to support and details to consider. As an example, I manually investigated when and in which order expressions for values sent via channels in select cases are evaluated, by using calls to functions with side effects (printing to the terminal) as value expressions.

As mentioned above, Toph's `builder` package recursively descends down the Go AST, building the IR in the process. It finds all operations that can be modelled in Uppaal, even if those are part of larger constructs than can not be modelled. For example, in the statement `var x int = <-ch + 5`, the variable declaration and the addition can be ignored, while the channel read operation gets turned into an `ir.ChanOpStmt` and added to the enclosing `ir.Body`.
Toph uses the `go/types` package to help identify the types of variables and check that function calls refer to known functions. In some cases Toph expects to be able to evaluate expressions, such as in an assignment of a channel variable or when encountering a channel operation. If Toph fails to identify such an expression as an element in the IR, it generates a warning indicating that the generated model is incomplete.
A few common Go packages (`flag`, `fmt`, `math`, `rand`, `sort`, `strconv`) and some functions in the `time` package are known to Toph to be okay to ignore in the Uppaal model. `time.After(time.Duration)` is a very commonly used function which returns a channel for timeouts. If Toph encounters it in a program, it adds two functions to the IR that behave equivalently and adds an `ir.CallStmt` to them instead.

4.3 Conversion of Toph Intermediate Representation to Uppaal systems

The `translator` package in Toph takes an IR and builds a Uppaal system from it. Toph uses its own internal `uppaal` package to help with naming and generating output files in different formats. Every generated Uppaal system contains the *Channel* process presented in Section 3.2. For each instance of the *Channel* process, Toph adds a query, e.g. `A[] not Channel0.bad`, to ensure that the bad state of the Channel is never reachable.

Each *ir.Function* becomes its own process in Uppaal. Toph's `translator` recursively descends down the IR tree, generating all process states and transitions and adding all variables either to the global system declarations or local declarations.
During this phase Toph also generates the layout for all states. For each `ir.Body` there is a start state from which all the translated `ir.Stmts` follow below and to the right. This allows Toph to calculate a rectangular bounding box around the states of each body and enables nesting loops, if statements, select statements etc. together.

When Toph generates the pending `receiving` or `sending` state for a channel operation or the `select_pass_2` state of a select statement, it adds a query, e.g. `A[] not (deadlock and countWords_0.sending_errChan_0)`, to ensure that the program can never reach a deadlock where the process is stuck waiting in that pending state.

4.3.1 Determination of Required Uppaal Process Instances

One interesting detail in the translation from IR to Uppaal systems is how Toph determines the required number of process instances in Uppaal for functions and the *Channel* process. Uppaal does not allow for the creation of new instances during simulation. Originally, Toph simply defined a constant number of instances of each process - 10 per function process and 100 *Channel* instances. This approach generally worked because most programs did not use all instances during simulation. If they did exceed the limit, the simulation would fail with an `IndexOutOfBounds` exception. However, having lots of unnecessary, unused instances presents extra work for Uppaal and clutters the simulator.
Therefore, I developed an algorithm to find an upper bound on the number of function invocations in Go programs. This upper bound is used to determine the number of function process instances required in Uppaal. The upper bound for the `make` function for channels determines how many *Channel* instances are needed. The algorithm and associated data structures are part of the `analyzer` package of Toph.

The algorithm consists of two phases. In the first phase, each `ir.Function` gets analysed separately. During the second phase, the results are combined to build and analyse a function call graph. The algorithm uses a maximum upper bound value of 500.

The goal of the first phase is to determine how many times an `ir.Function` can call other functions. The problem is solved recursively for each `ir.Body`, by processing all its statements:

- An `ir.CallStmt` adds one to the count for the callee.
- An `ir.MakeChanStmt` adds one to the count for the `make` channel function.
- For an `ir.ForStmt` or an `ir.RangeStmt`, the results get determined for the loop body and multiplied by the maximum number of loop iterations. If there is no upper bound, everything called in the loop reaches the maximum upper bound value.
- For an `ir.IfStmt` or an `ir.SelectStmt`, the results get determined separately for each branch or case. Then the maxima between the branches get determined.
- All other `ir.Stmts` get ignored.

The second phase starts by building a function call graph (FCG), which is a directed graph representing functions as nodes and calls as edges from caller to callee. Toph then

determines the strongly connected components of the FCG using Tarjan's algorithm(19). A strongly connected component (SCC) is a (maximum) subset of nodes in a directed graph such that there are paths from each node to each other node in the subset. The SCCs of a directed graph form a directed acyclic graph with the edges between them. This means that the SCCs can be topologically sorted. Tarjan's algorithm finds SCCs in reverse topological order.
SCCs help to identify call cycles, which Toph can not find bounds for. An `ir.Function` that does not call itself recursively and is the only member of its SCC is not part of a call cycle.
Toph traverses the found SCCs in topological order, starting with the SCC of the `main` function. If a call cycle is found, every function called from the SCC gets set to the maximum upper bound value. Otherwise, it is known how many times the one function in the current SCC gets called. This number gets factored into all the call counts for calls out of the function.

Figure 4.2 shows an example program and the corresponding FCG. In the first phase of the algorithm, Toph determines individual call counts. For example, the `main` function calls `f` 3 times and `i` calls itself recursively an unbound number of times.
When Toph enters the second phase of the algorithm, it finds the SCCs and determines that `i` is in a call cycle and that `g` and `h` are in a call cycle.
At the start of the traversal of SCCs in topological order, when the `main` function gets processed, the total call count for `f` gets set to 3, and for `g` and `i` to 1.
When the SCC of `g` and `h` gets processed later, due to the detected call cycle the total call counts for `g`, `h`, and `f` get set to the maximum upper bound (500).
Similarly, when the SCC of `i` gets processed, the total call counts for `i` and `j` also get set to the maximum upper bound.

Toph correctly determines the call counts and required number of Uppaal process instances for *WordCounter* - `findFilesInFolder`: 1, `countWords`: 2, `main`: 1, `Helper 1`: 2, `Helper 2`: 1, *Channel* process: 6.

```go
func main() {
    // toph: max_iter=3
    for i := 0; i < 3; i++ { f() }
    g(20)
    i(30)
}
func f() {}
func g(x int) {
    if x % 3 == 0 { h(x/3) }
}
func h(x int) {
    if x % 2 == 0 { g(x/2) }
}
func i(x int) {
    for x % 5 == 0 {
        x /= 5; i(x)
    }
    j()
}
func j() {}
```

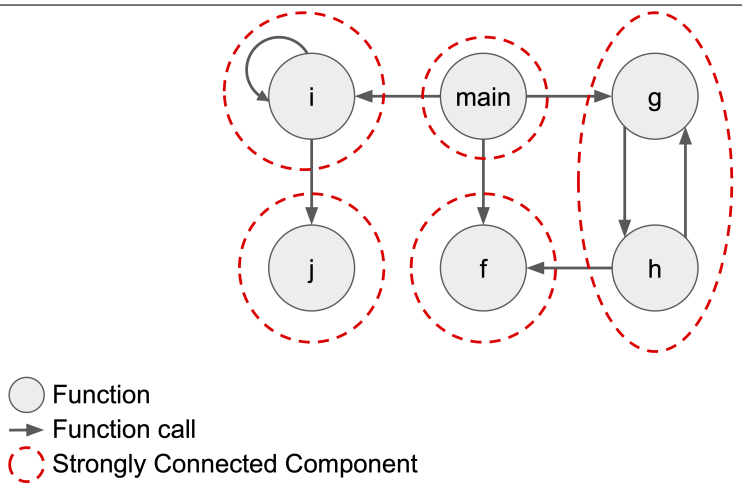


Figure 4.2: Go code and corresponding function call graph (FCG).

5 Evaluation

116 test programs in total were used to test Toph and investigate the generated Uppaal Systems. The programs are:

- *WordCounter* from Figure 1.1.
- 7 other programs with example code for concurrency in Go.
- 5 test cases written mainly to test how Toph handles different language constructs.
- 30 test programs that were written to experimentally analyse the semantics of select statements (see Section 3.1.4).
- 1 test program that was written to test the model for loops ranging over channels (see Section 3.3.3).
- 65 test programs used in (10) to test their *dingo-hunter* tool[1].
- 7 test programs from an open source repository showcasing Go concurrency patters[2].

The 65 test programs used originally for *dingo-hunter* are a mixture of classic academic concurrency examples (e.g. the dining-philosophers problem, distributed computation of Fibonacci numbers, and prime sieve), examples developed specifically to test modeling capabilities (e.g. dynamically spawning of Go routines in a loop), and real-world case studies for bugs.

The evaluation is split into two parts. Section 5.1 discusses how Toph is handling the test programs and what causes issues in the translation process.

Section 5.2 examines the generated Uppaal systems, the verification process in Uppaal, and whether the results for deadlock detection and channel safety are as expected.

All testing was done on a 2019 MacBook Pro, with a 2.4 GHz Quad-Core Intel Core i5 and 8GB of RAM. Due to this setup and unpredictable limitations like thermal-throttling, reported running times can only be interpreted as general indications of performance.

[1] see `https://github.com/nickng/dingo-hunter/tree/master/examples`
[2] see `https://github.com/stillwater-sc/concurrency`

5.1 Toph

Toph translates all 116 test programs, over 5040 lines of Go code, in a few seconds. No single program took more than 20ms to translate. Toph's running time appears to be roughly linear in the size of the input programs.

92 of the 116 test programs get translated without any warnings. The other 24 test programs generated 149 warnings in total.
61 warnings across 8 test programs indicate that Toph was not able to resolve an expression for a channel. One of these occurred because a test program defined a type alias for a channel type. All others are the result of channels being stored in data structures, namely structs and slices.
79 warnings across 16 test programs indicate that Toph was not able to resolve the callee of a function or go call. In a few cases, functions are stored in variables or in a map structure. In some cases, the callees are methods of a type. It would not be too difficult to add support for methods. Some callees are Go library functions Toph does not know about or deliberately does not model, such as `os.Exit(int)`, which terminates a Go program, or the `panic` function, which is part of the language and indicates critical unexpected failure such as a failed assertion. `panic` in Go works similarly to `throw` in C++ or Java but is meant to be used more rarely.
2 warnings in 2 similar test programs indicate that Toph gets confused by an outer channel variable `x` that gets shadowed by an inner integer variable `x`. Toph skipped the declaration of the integer variable and expected `x` to still refer to the outer channel variable. However, the `go/types` package, which Toph uses to double check, indicated otherwise. The generated Uppaal systems are still correct.
1 warning indicates that Toph does not currently translate switch statements. It would not be difficult to add support for switch statements. But this is an example where working with SSA form instead of the AST would have saved work (see Section 4.2).
5 warnings across 2 test programs indicate that Toph does not support defer statements. Defer statements schedule a function call to occur at the end of the current function. The classic example is to defer closing a file to ensure this happens no matter how the function exits. Defer statements can occur in loops or branches of if statements. This makes modelling them in Uppaal about as difficult as modelling dynamic function dispatch in general, which is not supported.
And finally, 1 warning indicates that a test program does not have a `main` function. It is still possible to look at the Uppaal processes of the other functions in the program. But without a `main` function the system can not take any transitions at all.

Uppaal automatically checks for syntax errors when it reads system files and query files. No syntax errors were detected in any of the 348 files Toph generated for Uppaal.

5.2 Uppaal systems

Through an automatic testing setup, Uppaal's query checker was run against all 116 systems and sets of queries. For 84 systems, the verification process completed successfully. In some cases it was necessary to annotate loops with iteration bounds to help the query checker. Using static analysis frameworks for loops can help to improve Toph in this regard in the future.

Table 5.1 lists 49 test cases used to test the *dingo-hunter* tool, shows the Uppaal query checker running times for each simulated test program, and lists whether the query checker completed or not. The full results for all 116 test programs are attached separately.

For 32 of the 116 generated systems, verification was aborted for some queries. One cause for this are systems that use more than the provided number of process instances for channels or function invocations, exceeding the maximum upper bound on function calls and channel creations mentioned in Section 4.3.1. This leads to an `IndexOutOfBoundsException` in Uppaal's query checker.
Some of the test programs for select statement semantics in particular run into an issue where a Uppaal process reads from a closed *Channel* in an infinite loop. This keeps decreasing the `counter` variable of the *Channel* instance until it eventually exceeds the minimum value for integers in Uppaal. There are ways to improve the channel model in Uppaal in the future to avoid this issue.
9 systems model test programs with call cycles in them. For all of them the query checker fails. This is an indicator of the difficulties associated with handling recursion with the chosen approach.

Most queries can be checked very quickly. Only a few programs lead to large search spaces for the query checker to handle. This can be the case when a lot of processes with a lot of states operate mostly independently of each other. In that case Uppaal has to consider all possible interleavings for taken transitions. These can grow worse than exponentially with the number of processes and states.

For all test programs that were manually inspected, the results from the query checker for channel safety were correct. Some test programs, such as the branch-dependent-deadlock program from *dingo-hunter*, hinge on program data to avoid deadlock. Because the Uppaal model is an over-approximation with regards to if statements, the model allows for simulation traces that would not be possible in the original program. This leads the query checker to find deadlock scenarios that are not possible in practice and report incorrect results. However, this still means that Toph and Uppaal appear to be a sound approach to concurrency verification for Go. I was not able to find a test case where neither Toph nor Uppaal falsely indicated that there are no issues.

Table 5.1: Query check results for *dingo-hunter* test cases.

Test	Duration	Verification	Remarks
altbit	2.4s	Completed	
branch-dependent-deadlock	0.2s	Completed	
channel-scoping-test	0.1s	Completed	
commaok	0.1s	Completed	
cond-recur	0.1s	Completed	
deadlocking-philosophers	0.4s	Failed	Uses Methods
dining-philosophers	0.4s	Failed	Uses Methods
factorial	1.8s	Failed	Call Cycle
fanin-pattern	0.6s	Completed	
fanin-pattern-commaok	0.5s	Completed	
fcall	0.1s	Completed	
forselect	0.1s	Completed	
giachino-concur14-dining-philosopher	2946.2s	Completed	
giachino-concur14-factorial	2.7s	Failed	Call Cycle
github-golang-go-issue-12734	0.1s	Completed	
golang-blog-prime-sieve	612.8s	Completed	
infinite-prime-sieve	223.5s	Completed	
issue-10-close-wrong-migo-chan-name	0.1s	Completed	
issue-11-non-communicating-fn-call	0.1s	Completed	
jobsched	0.5s	Completed	
local-deadlock	0.1s	Completed	
local-deadlock-fixed	0.1s	Completed	
loop-variations	0.1s	Completed	
makechan-in-loop	0.1s	Completed	
md5	0.1s	Completed	
multi-makechan-same-var	0.1s	Completed	
multiple-files	0.1s	Completed	
multiple-timeout	0.1s	Completed	
parallel-buffered-recursive-fibonacci	4.4s	Failed	Call Cycle
parallel-recursive-fibonacci	3.3s	Failed	Call Cycle
parallel-twoprocess-fibonacci	0.1s	Failed	Call Cycle
philo	11.5s	Completed	
powsers	167.7s	Failed	Call Cycle
producer-consumer	0.1s	Completed	
ring-pattern	0.4s	Completed	
russ-cox-fizzbuzz	2.8s	Completed	
select-with-continuation	0.1s	Completed	
select-with-weak-mismatch	0.1s	Completed	
semaphores	0.1s	Completed	
send-recv-with-interfaces	0.1s	Completed	
simple	0.1s	Completed	
single-gortn-method-call	0.1s	Completed	
spawn-in-choice	0.1s	Completed	
squaring-cancellation	3.7s	Completed	
squaring-fanin	7.8s	Completed	
squaring-fanin-bad	9.1s	Completed	
squaring-pipeline	2.5s	Completed	
struct-done-channel	0.1s	Completed	
timeout-behaviour	0.2s	Completed	

While the introduced annotations for loop bounds can represent an under-approximation, letting a loop body execute fewer times than in the original program, they had no impact on soundness in practice. They still require careful consideration by the user and should only be used if the system can also show issues with only a few iterations.

Overall, Toph and Uppaal handle channel safety well while there is more room for improvement regarding the detection of total or partial deadlocks. Partial deadlocks in particular can not currently be queried for.

6 Conclusion

In this dissertation I presented the use of Uppaal as a new approach to modeling and verifying concurrent Go code. I thoroughly investigated the details of channels and the surrounding language constructs in Go and developed ways to model them accurately and correctly in Uppaal. In addition, I found ways to model important aspects of the larger Go programming language in Uppaal and made it easy to draw connections between Uppaal systems and Go code.

Toph is a substantial effort to implement these ideas in an automatic tool. Developing Toph required solving a variety of problems, from designing a suitable IR to function call graph analysis. Working with the AST and not SSA form meant more work during the implementation but a more approachable tool for users.

Applying Toph and Uppaal to a range of existing and new test programs was successful and sound for most test programs. The most significant encountered limitations were call cycles, infinite loops, large state spaces for some test programs, no modelling support for data structures, and no dynamic function dispatch.
Annotations are a viable solution to the issue of infinite loops. The other limitations appear to be inherently tied to the chosen methodology and are not easy to work around or overcome.
However, for a lot of practical uses of Go, Toph and Uppaal can provide a powerful tool set for programmers. They help spot issues around channel safety and deadlocks that are very hard to reason about manually. The development of *WordCounter*, as described in the introduction, is such a use case.

Future Work

I am planning to undertake a summer research internship to continue working on Toph and the Uppaal model towards a publication at a conference. There are already a lot of ideas for improvements:

- The *Channel* process and channel operations can be adjusted such that reading from a

closed channel does not reduce the `counter` value below 0.

- There are remaining language features, such as switch statements, methods, and structs that Toph could support.
- By using a static analysis tool, Toph could determine loop bounds automatically. Other tools on Go concurrency verification already employ similar methods.
- Recursion as an issue could be addressed by forcing choices towards branches that break call cycles at a set recursion depth. This could be implemented through function inlining in Toph's IR or possibly as part of the simulation in Uppaal.
- Mutexes and wait groups, two other commonly used Go concurrency constructs, can also be modelled in Uppaal.
- There is a lot potential to improve the detection of partial and global deadlocks in Uppaal.

Bibliography

[1] RobPike. Go at google: Language design in the service of software engineering. `https://talks.golang.org/2012/splash.article`. Accessed: 2020-04-30.

[2] Todd Kulesza. Go developer survey 2019 results. From The Go Blog. `https://blog.golang.org/survey2019-results`. Accessed: 2020-04-30.

[3] Andrew Gerrand. Share memory by communicating. From The Go Blog. `https://blog.golang.org/codelab-share`, 2013. Accessed: 2020-04-26.

[4] Sameer Ajmani. Go concurrency patterns: Pipelines and cancellation. From The Go Blog. `https://blog.golang.org/pipelines`. Accessed: 2020-04-30.

[5] Y. Li, X. Yin, Z. Wang, J. Yao, X. Shi, J. Wu, H. Zhang, and Q. Wang. A survey on network verification and testing with formal methods: Approaches and challenges. *IEEE Communications Surveys Tutorials*, 21(1):940–969, 2019.

[6] Judith Crow and Ben Di Vito. Formalizing space shuttle software requirements: Four case studies. *ACM Trans. Softw. Eng. Methodol.*, 7(3):296–332, July 1998. ISSN 1049-331X. doi: 10.1145/287000.287023. URL `https://doi.org/10.1145/287000.287023`.

[7] S. Gouw, Jurriaan Rot, Frank Boer, Richard Bubel, and Reiner Hähnle. Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case. pages 273–289, 07 2015. ISBN 978-3-319-21689-8. doi: 10.1007/978-3-319-21690-4_16.

[8] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2 edition, 2004. doi: 10.1017/CBO9780511810275.

[9] Luca Aceto, Anna Ingólfsdóttir, Kim Guldstrand Larsen, and Jiri Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, USA, 2007. ISBN 0521875463.

[10] Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 174–184, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342414. doi: 10.1145/2892208.2892232. URL https://doi.org/10.1145/2892208.2892232.

[11] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. A static verification framework for message passing in go using behavioural types. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1137–1148, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356381. doi: 10.1145/3180155.3180157. URL https://doi.org/10.1145/3180155.3180157.

[12] Kai Stadtmüller, Martin Sulzmann, and Peter Thiemann. Static trace-based deadlock analysis for synchronous mini-go. *CoRR*, abs/1608.08330, 2016. URL http://arxiv.org/abs/1608.08330.

[13] Julien Lange, Nicholas Ng, Bernardo Toninho, and Nobuko Yoshida. Fencing off go: Liveness and safety for channel-based programming. *SIGPLAN Not.*, 52(1):748–761, January 2017. ISSN 0362-1340. doi: 10.1145/3093333.3009847. URL https://doi.org/10.1145/3093333.3009847.

[14] Jan Midtgaard, Flemming Nielson, and Hanne Riis Nielson. Process-local static analysis of synchronous processes. In *Static Analysis*, Lecture Notes in Computer Science, pages 284–305. Springer, 2018. ISBN 9783319997254. doi: 10.1007/978-3-319-99725-4_18.

[15] Matt Insall and Eric W. Weisstein. Lattice. From MathWorld–A Wolfram Web Resource. https://mathworld.wolfram.com/Lattice.html. Accessed: 2020-04-27.

[16] David Castro, Raymond Hu, Sung-Shik Jongmans, Nicholas Ng, and Nobuko Yoshida. Distributed programming using role-parametric session types in go: Statically-typed endpoint apis for dynamically-instantiated communication structures. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi: 10.1145/3290342. URL https://doi.org/10.1145/3290342.

[17] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[18] Kim G. Larsen, Paul Pettersson, and Wang Yi. Model-Checking for Real-Time Systems. In *Proc. of Fundamentals of Computation Theory*, number 965 in Lecture Notes in Computer Science, pages 62–88, August 1995.

[19] R. Tarjan. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 114–121, 1971.

A1 Appendix

The source code for Toph is available at `https://github.com/arneph/toph`. Running Toph requires an installation of Go. Uppaal is available at `http://www.uppaal.org`.