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

School of Computer Science and Statistics

Vtrace: Uppaal trace verification for Go programs

Brian Neville

A dissertation submitted in partial fulfilment of the degree of M.A.I.
(Computer Engineering)

Supervised by Dr Vasileios Koutavas

Submitted to the University of Dublin, Trinity College, May 2021

Declaration

I, Brian Neville, declare that the following dissertation, except where otherwise stated, is entirely my own work; that it has not previously been submitted as an exercise for a degree, either in Trinity College Dublin, or in any other University; and that the library may lend or copy it or any part thereof on request.

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: 11th of May 2021

Abstract

This dissertation presents Vtrace (from the words ‘verify trace’). Vtrace is a tool which takes traces from an abstract model of a Go program and determines whether these traces correspond to real errors in the original program. Vtrace makes use of Uppaal to generate traces through the abstract model, and Toph for Go-to-Uppaal translation. To verify traces, Vtrace automatically builds and runs deterministic tests for each trace and can force Uppaal to exhaustively regenerates new traces until an error is confirmed real.

Vtrace presents the output of any tests to the user in the form of readable log files, allowing the user to see the exact execution of the program which led to the error (complete with concurrent goroutine interleavings).

Vtrace is shown to be capable of finding errors in small Go programs but struggles to deal with programs where shared memory is used across multiple goroutines. Evaluation also shows that performance scales quite poorly, and that this tool is realistically unsuitable for large-scale programs.

Acknowledgements

Firstly, I would like to acknowledge my supervisor Dr Vasileios Koutavas, for his assistance, time, and commitment to helping me with my project.

I would also like to acknowledge my family and thank them for their support over the years. In particular, I am thankful for my dad who has been incredible throughout my entire life.

It has been a pleasure to make so many new friends in college, and so I would like to acknowledge everyone who helped make the last five years amazing.

Contents

List of Figures

List of Tables

1. Introduction

1.1 Overview:

Section 1 of this dissertation is an introduction, to introduce the reader to the Go programming language, familiarize the reader with the idea of trace reconstruction, highlight the value that Vtrace brings, and give a summary of the overall contributions of this project.

Section 2 of this dissertation contains details on background of this project, covering some of the important tools used as part of this project, as well as some of the related research in the field of trace reconstruction.

Section 3 explains the methodology behind this project, firstly giving an overview of the entire algorithm used, before afterward going into detail on each of the separate components.

Section 4 of this dissertation describes the implementation of this algorithm, going into detail on all major parts of the implementation, with example code to help where needed.

Section 5 covers the evaluation of Vtrace, explaining how and why performance scales the way that it does at each part of the algorithm, as well as covering the impact of some of the optimisations made.

Section 6 goes into detail about the limitations of the implementation and offers potential solutions in future work.

Section 7 is a short conclusion on the entire project.

concurrency and channel operations in Go.

The Go programming language was released in 2009, with work on the language design starting at Google in 2007[1]. With the prevalence of multicore systems and the increasing need to write safe, simple concurrent code in large scale systems, Go was designed to allow concurrency to be introduced into a program with ease. It features CSP-style concurrency (communicating sequential processes), with the 'go' keyword being used to run a function call concurrently in a 'goroutine' – a lightweight user-space thread, managed by the Go runtime scheduler.

For example, the line:

```
go serveConnection(conn)
```

will run the function serveConnection with the argument 'conn' in a concurrent goroutine.

Go also offers channels, which can send values to other goroutines or receive values from other goroutines. The use of channels for communication between goroutines is highly encouraged, and idiomatic Go code tends to make use of channels to share information, rather than trying to synchronise reads or writes to some shared memory location.

For example, a goroutine X that executes the line:

```
channelA <- result
```

will try to send the value of the result variable into channelA.
If the channel is buffered (meaning that it can store a buffer of sent items), then X will continue past this line, but if the channel is unbuffered, then this line will block until another

[1] *Go FAQ*, https://golang.org/doc/faq (last accessed 2021-02-14)

goroutine tries to receive from the same channel. To continue the example, when a separate goroutine Y executes the line:

```
resultX := <- channelA
```

This line receives from the channelA, and assigns the value received from the channelA to the variable resultX. After this line, both goroutine X and goroutine Y get unblocked and can progress.

State spaces, model abstractions and goal of trace reconstruction:

Concurrent systems are notoriously difficult to manually analyse. As the number of concurrent actors increases, so does the number of possible states that the program can have at any point in time. This complexity makes concurrent systems susceptible to reaching unintended/bad states (such as deadlocks or runtime errors). Additionally, unless constant logging is done for each concurrent actor, if a program is found in a bad state it can be difficult to determine the exact sequence of concurrent interleavings which produced this state.

To check for errors before a program is run, a model checker can be used. This takes a representation of the source program (which is abstracted to increases verification speed) and will provide a trace (a sequence of states and state transitions) that leads to a bad state when possible. However, use of abstract model for this verification means that traces can be reported which would not be possible in the source program.

Prata in [1] describes the 'state' of a program as the "set of values of all of the variables at any given point in program execution". From this, the 'state space' of a program is defined as the set of all possible states that a program can have. It is clear then, that as the number of variables in a program increases, the size of the state space will increase exponentially. This 'state space explosion' problem outlined by Clarke et al. in [2] makes it difficult to perform model checking on programs if the model checking were to involve exploring the entire state space.

Therefore, to make model checking feasible, an abstract model with a smaller state space can be used. This abstracted model of the program can be constructed in a number of ways. For example, in [3], Stadtmüller et al. restrict their work to using a subset of the Go language with limited features and channels are only used for synchronization, with the messages that they pass being ignored. Similarly, the Toph translator used by Vtrace (explained in section 2) builds an abstracted model of the program by disregarding portions of the program which are not pertinent to the programs channel operations (it also only reduces the channels to be for synchronization, not for message-passing).

As mentioned above, use of an abstracted model leads to inaccuracies, since the abstraction process may have removed some of the logic which was in place to prevent certain conditions from occurring, allowing the model checker to then report that states can be reached which would never occur in the source program. This inconsistency of abstract model checkers prompts the need for tools which can verify whether or not a trace from a model checker corresponds to a valid execution of the source program.

1.2 Project objective and research questions

The principle objective of this project is to develop a tool which is capable of reconstructing a trace in a unit test form, where this trace is extracted from a Uppaal model of a Go program, and executing this reconstruction will allow the user to understand whether or not the trace corresponds to a real error.

This objective is derived from the following research question which resides is at the heart of this project:

"Given a trace through an Uppaal model of a Go program, is it possible to instrument the Go program such that the trace can be determined to be reproducible?"

This question arose following the development of the 'Toph' tool (Philipeit, 2020) [4] for translating Go programs into Uppaal models. Toph is discussed further in the background section. A review of the current domain of trace reconstruction and Go verification also revealed that while there is much research done in both of these domains, there remains to exist a tool which would otherwise accomplish this project's research objective.

1.3 Contributions

The major contributions are made with this project:

- A parser which instruments Go code to support execution following a prescribed sequence of state transitions.
- An approach for capturing the sequence of states from an Uppaal trace, and turning this sequence into a unit test
- A test runner that automatically run unit tests, with the ability to regenerate these unit tests using another unique sequence of states if the test fails to reproduce the error.

The above functionality has been implemented in the Vtrace tool, which currently resides in a GitHub repository at https://github.com/brianneville/vtrace. A fork of the Toph translator with modifications that support Vtrace is found at https://github.com/brianneville/toph.

2. Background

2.1 Uppaal

Uppaal is a verification tool which facilitates all stages of model verification – from model specification, to simulation and verification. It was developed in collaboration between Aalborg University in Denmark, and Uppsala University in Sweden. Originally released in 1995, Uppaal is continuously updated and improved with the current snapshot version (4.1.24) used for this project being released only recently in November 2019.

As a tool for modelling systems, Uppaal allows modelling of concurrent processes, where each process is created from the beginning of the simulation, and processes can synchronize with each other through channels or shared variables. Unlike channels in Go, Uppaal's channels are only used for synchronization, and they unable to pass data from one process to the other. Additionally, channels in Uppaal cannot be buffered, meaning that a 'send' operation on a channel will be blocking until a corresponding 'receive' operation is performed on that same channel (and vice-versa). Shared state can also be used for synchronization (for example, a state can be made to take a transition when a global integer variable increases beyond a certain threshold), through declaration of global variables which all processes can read or write to. Processes can also declare local variables, which are only accessible within that processes scope.

Models in Uppaal are composed of states, with transitions between these states dictating program flow based on either global variables, local variables, or channel synchronization. These transitions also have an 'update' capability, where a statement can be declared which is executed upon taking the transition. This statement could call a function or update global or local variables to some new value. Variables in Uppaal[2] can have types which are either 32-bit signed integers, booleans, channels, clocks (which evaluate to real-numbers and increment synchronously), or arrays or structs which contain these fields as types.

Models are accompanied by queries; logical formulae defined by the developer, which are expected to be true across the entire state space. These logical formulae[3] can adopt any of the following forms:

A[] p – "for all paths p always holds"
A<> p – "for all paths p eventually holds"
E[] p – "there exists some path where p holds"
E<> p – "there exists some path where p eventually holds"
p --> q – meaning "whenever p holds, q eventually holds"

Where p and q are state properties, written as expressions[4].

Models and their respective queries are saved in an XML format, and Uppaal also provides a GUI to interpret this format, aiding in the visualization, construction and debugging of these models. For verification and trace generation, Uppaal also provides a Java API to its C++

[2] *Uppaal Types*, https://docs.uppaal.org/language-reference/system-description/declarations/types/ (last accessed 2021-05-09)
[3] *Semantics of logical formulae in Uppaal*, https://docs.uppaal.org/language-reference/requirements-specification/semantics/ (last accessed 2021-05-09)
[4] *Expressions for properties of states in Uppaal*, https://docs.uppaal.org/language-reference/expressions/ (last accessed 2021-05-09)

verifier backend. This API allows programs to be written in which models can be instantiated and have their queries run on the verifier. When running queries, the API allows for parameters to be specified controlling aspects such as the search order (breadth-first, depth-first, or random-depth first), or the type of trace to return (some random trace, the shortest trace, or the fastest trace).

2.2 Toph

Toph (Philipeit, 2020) [4] is a tool which translates Go programs into Uppaal models. Toph supports translation of the following aspects of a Go program:

- branching statements (if/else statements, switch statements, select statements)
- function calls, including calls that will run in new goroutines
- panic/recover calls
- return statements
- for loops, continue or break statements
- loops ranging over a channel, array, slice or map
- calls to create a new struct (using the ‘new’ function)
- calls to create a new array, slice or map
- calls to copy a slice (using the ‘copy’ function)
- deletion from a map
- calls to create a new channel (buffered or unbuffered), send/receive over a channel, or close an existing channel
- calls that cause the program to exit (such as os.Exit)
- mutex lock/unlock operations (for sync.Mutex and sync.RWMutex)
- wait group operations (for sync.WaitGroup)
- calls to run functions in sync.Once.Do

While this translation process results in Uppaal models being constructed such that the interactions between all of these aspects of the program is modelled, the actual values used are not included in the model (i.e. the values stored in slices, maps, or arrays, or those which are sent/received over channels, or used to branch into if/else statements). This loss of information is inevitable as a consequence of the limited types supported by Uppaal. However, by only mapping the control flow of the program in this fashion, Toph creates an abstracted version of the program which allows for much faster verification, as the model’s state space is greatly reduced.

Upon translation, Toph also generates queries as specified by the user. These queries can check for the following properties of a system:

- goroutines exiting with a panic
- channel safety
- sync.Mutex and sync.RWMutex safety
- sync.WaitGroup safety
- channel related deadlocks
- sync.Mutex and sync.RWMutex related deadlocks
- sync.Once related deadlocks
- sync.WaitGroup related deadlocks
- parts of code are reachable or not reachable (requires the desired part of code to be annotated)

Additionally, Toph can generate queries related to the resources used by the verifier itself, namely:

- queries to check that resource bounds are never exceeded
- queries to check that resource bounds for particular resources are never exceeded

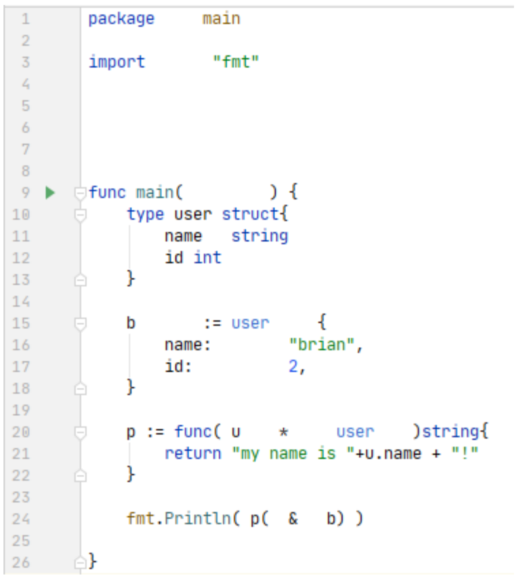
To specify in formal verification terminology, the models produced by Toph are *complete* models – that is, they are models which contain all errors that exist in the original program, but which also contain false errors.

2.3 Gofmt

Gofmt is a tool provided as part of the Go distribution[5]. Gofmt is a formatting tool which standardizes the format of valid Go files, modifying whitespace and indentation. This tool was provided by the Go authors so that Go programs could have a consistent, familiar style on first read. It also allows for developers to write code without worrying about formatting, and across projects using source control it allows reviewers to see exactly what parts of their code have been modified (i.e. No developer's individual formatting style can contribute to the modifications).

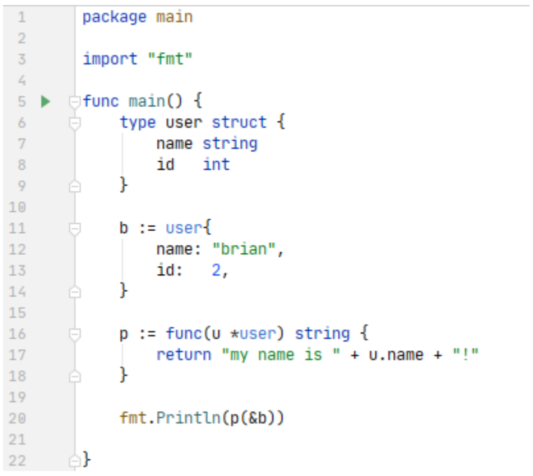
As of 2013, 70% of Go code written used gofmt[6], and gofmt plugins exist which can automatically run gofmt on file-save for editors such as Visual Studio Code, Emacs, Vim, and GoLand.

The following figures show an example of the gofmt tool being used:



```
package     main

import       "fmt"





func main(         ) {
    type user struct{
        name   string
        id int
    }

    b       := user     {
        name:        "brian",
        id:          2,
    }

    p := func( u    *     user    )string{
        return "my name is "+u.name + "!"
    }

    fmt.Println( p(  &   b) )

}
```

Figure 2.1: File with haphazard formatting



```
package main

import "fmt"

func main() {
    type user struct {
        name string
        id   int
    }

    b := user{
        name: "brian",
        id:   2,
    }

    p := func(u *user) string {
        return "my name is " + u.name + "!"
    }

    fmt.Println(p(&b))

}
```

Figure 2.2: File formatted with gofmt

2.4 Related work in trace reconstruction and verification

The need to be able to debug code which fails on very rare occasions or for obscure reasons has driven a lot of research and development in the domain of trace reconstruction. For example, in [5], Srinivasan et al introduce a system called Flashback, with which a Linux

[5] *Gofmt source code*, https://golang.org/cmd/gofmt/ (last accessed 2021-05-09)
[6] *Gofmt blog post*, https://blog.golang.org/gofmt (last accessed 2021-05-09)

process that encounters an error can be rolled back to some prior stable state, and the actions which led to the error can be replayed. As the process executes, calls to a custom kernel function (which are either inserted by the programmer or automatically) inside a modified Linux 2.4.22 kernel save the state of the process in a suspended ‘shadow-process’. System calls made by the process are captured, so that any information exchanged with the environment that affects the process (such as interprocess communication, network sending or receiving, or bytes read from a file) is logged. When replaying an error, the shadow process is forked and resumed, and all system calls are replayed to match the logged versions that led to the error.

Since Flashback operates on the kernel level, the order of thread scheduling is also logged – meaning that multithreaded processes can be replayed by forcing the kernel to interleave the threads in a certain order.

Flashback offers a much stronger replay engine than Vtrace, and its capacity to reproduce system calls and environment information is a testament to the power of kernel-space code. Its use of kernel-space code however means that Flashback is more complicated and less safe to run (compared to Vtrace which runs entirely in user-space). By comparison, Vtrace cannot detect if an external stateful object existed and this may lead to testcase behaviour changing if such an object was a dependency and was modified in during testing of multiple Q<num>.go.txt files.

A user-space alternative to Flashback is offered by Saito in Jockey [6] a shared object file which can be linked into a program to enable logging for deterministic replay. As with Flashback, jockey captures checkpoint states of the process. System calls in the libc[7] library are modified so that they log any returned values, which can be returned again later if the program is replayed. However, as this code does not run in the kernel, it cannot record thread-scheduling order, and so multi-threaded programs cannot be replayed.

However, both Flashback and Jockey feature no verification aspect, and so it can only hope to find errors or undesirable states in programs through continuous execution of the program. This means that (depending on the size of the program and the number of possible states) they may not be useful until a program has been running for a long time. With no verification aspect, the developer has a choice of arbitrarily stopping testing their program after a period of testing, or running the program with these tools enabled, which will incur some performance overhead (see section 6 in [5] pp. 9-13 and section 5 in [6] pp. 8-9).

Deels et al. present a system which is capable of verifying properties of traces in Friday [7]. Friday leverages liblog [8] to capture checkpoints and logs for a program’s execution. Liblog is a user-space shared library that (similar to Jockey), catches and logs calls made to libc functions. A custom scheduler ensures that only one thread at a time is running, and a thread can only be suspended when encountering another system call – at which point that thread’s context is logged. This allows for multithreaded applications to be logged, without any modifications to the kernel. In Friday, liblog allows any trace to be deterministically replayed in a distributed system.

To aid developers in debugging these traces, Friday introduces watchpoints and breakpoints. Breakpoints functions similarly to GDB breakpoints, where the program

[7] *libc(7) — Linux manual page,* https://man7.org/linux/man-pages/man7/libc.7.html (last accessed 2021-02-08).

surrenders execution to the debugger upon encountering. Watchpoints allow for variables in a program to be monitored during execution, and global watchpoints can be set that monitor the value of one or multiple variables across many nodes in the distributed system. This allows for global invariants to be written which can be evaluated to verify properties of the trace as it is being replayed. This invariant is specified in a script, using python to write scripting commands to a built-in interpreter.

This approach differs from the approach used by Vtrace in a number of ways. Firstly, the properties of a trace can only be verified when replaying the trace, while Vtrace depends on a verifier (Uppaal) to priorly provide a trace which will lead to a certain property. Secondly as with Jockey and Flashback, Friday's replay system (liblog) is able to reproduce the results of system calls made during the trace, which Vtrace is unable to emulate. Friday is also designed to support debugging across multiple processes, while the use of channels in Vtrace to schedule traces means that only traces from one process can be replayed.

Following Friday, the work of Liu et al. in [9] introduces the D^{3}S checker for distributed systems. D^{3}S does not feature a replay system for traces, and rather focuses on catching errors while the program is running. A developer supplies a number of functions that contain predicates for the system, which must hold for a part of the system or for the system as a whole. D^{3}S uses a separate verifier process to check this predicate, with the verifier process receiving state information from all other processes in the system. In the event that a predicate fails, D^{3}S then reports the states which led to that failure. Compared to Uppaal and Vtrace, D^{3}S's but its lack of trace reconstruction capability means that errors may be more difficult to investigate.

The Pensieve tool outlined by Zhang et al. in [10] takes a different approach to failure reproduction than previously mentioned work. Pensieve uses 'event chaining' (described in [10] section 3, pp. 4-9) to infer the minimal lines of relevant code that are required to reproduce the trace. In this case, the 'trace' is a combination of "log files output by the failure execution, the system's bytecode, a list of supported user commands, and a description of the symptoms associated with the failure (typically a user-selected subset of error messages in the log files)" (see [10] p.2). The control flow and the program data which are deemed to be dependencies for this trace are then constructed into a unit test. This unit test is executed three times to confirm that it can reproduce the exception. In the event that any of these attempts fail, Pensieve re-constructs the unit test using a slightly modified trace (e.g. by taking alternate branch conditions). Pensieve's ability to deal with faulty unit tests in this way is a guaranteed by the fact that it will never receive false traces (unlike Vtrace). Therefore, this approach of iteratively building minimal unit tests could not be applied to Vtrace, as Vtrace needs to be able to determine whether a trace is real or not.

The latest approach to trace verification in Go is Dara, described by Anand in [11]. Dara uses a concrete model checking approach, meaning that no abstraction of the system is created in order to check its properties. Dara instead instruments the source code to allow for a global scheduler to control execution as it verifies properties, making use of a custom Go runtime package which communicates with this scheduler. Similar to D^{3}S, desired properties are declared through functions that get checked during execution. In the event that a property fails to hold, Dara returns a trace of the sequence of states that led to the undesired state. These traces can be replayed, leveraging the same global scheduler and runtime infrastructure. To deal with the state space explosion during verification, Dara offers a selection of strategies for prioritizing path selection during the exploration stage.

Compared to Vtrace, Dara’s approach is potentially a lot slower due to the lack of an abstract model. However, performing concrete model checking in this fashion means that unlike the Toph models used by Vtrace, it will never deal with false traces. Dara is also arguably less transparent than Vtrace, with much of its implementation hidden from the user in a modified runtime environment and scheduling infrastructure. By comparison, use of the native Go runtime in Vtrace, and the use of channels to enforce scheduling provides a slightly clearer way for users to follow the trace’s execution in the testcases.

3. Methodology

3.1 General algorithm overview

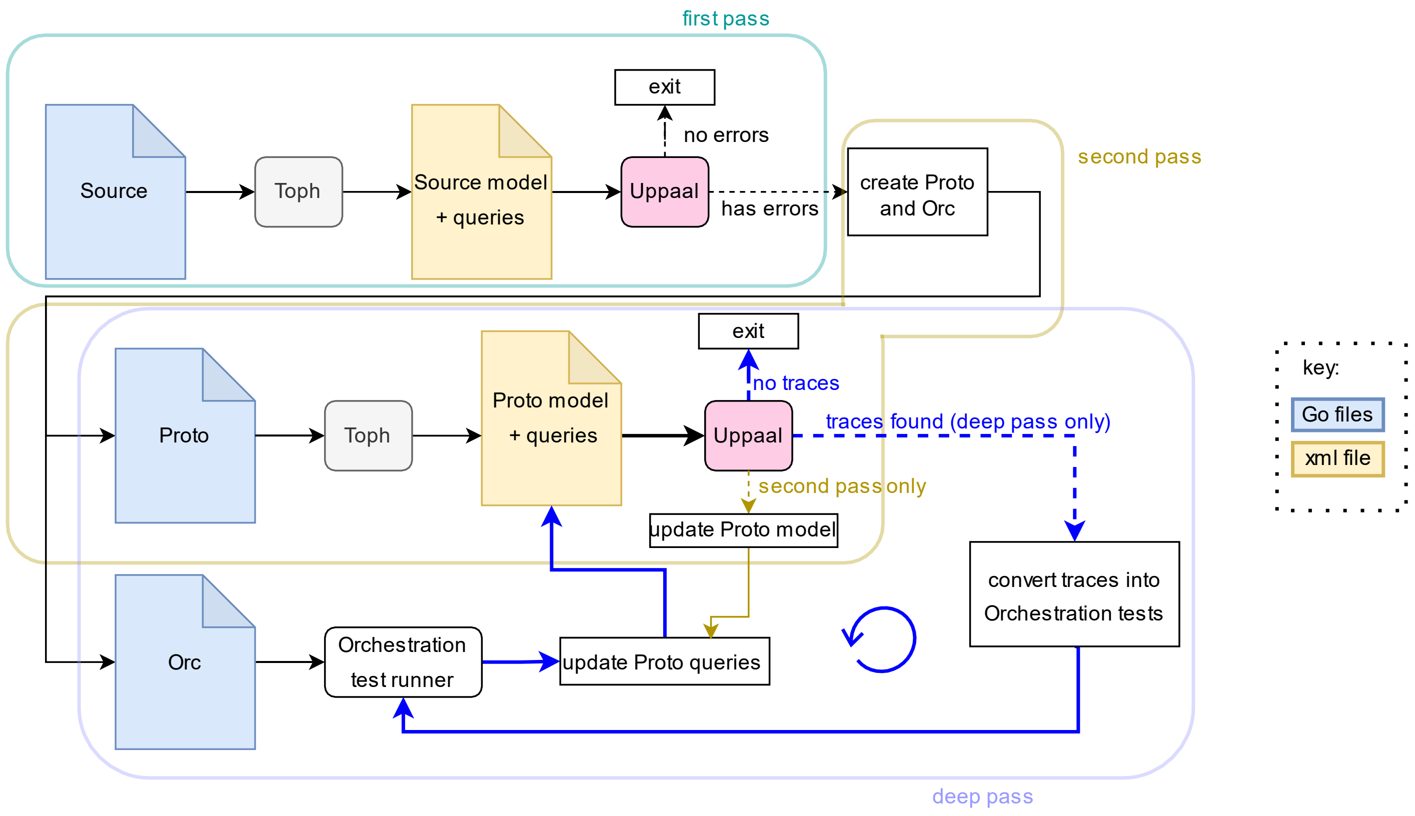


Figure 3.1: Vtrace general algorithm overview

The general algorithm for trace reconstruction, unit-test generation and verification which is applied in this project will be presented here in a brief overview, with following subsections providing a further in-depth explanation for individual components.

Figure 3.1 provides a visual reference for the algorithm, with the gold arrows representing the typical program flow, black lines indicating the reading or writing of files, and grey lines showing the places at which the program will exit. The original program (shown labeled as *Source* in figure 3.1) is a directory containing Go files and/or Go packages.

Beyond this point the following terms begin to appear, and so they are defined for the reader here for clarity:

Real error: A real error is an error which exists in the source program, and which will occur through some combination of interleavings of multiple goroutines in the source program. It is a state that the program can reach where upon reaching this state, the program will exhibit or has exibited some behaviour which is undesirable (for example, reaching this state may mean that the Go runtime panics and/or the program exits undesirably). For a real error, a trace can produced by Uppaal where the sequence of transitions can be reconstruct into a test for the source program, and this test can be run to completion.

False error: A false error is an error which does not exist in the source program, and which will never occur for all possible interleavings of goroutines. A false error exists only as a byproduct of the abstraction that Toph applies when translating the Go program into a

Uppaal model. False errors are produced by false traces – Uppaal traces which report a path through the model that cannot be reproduced in the source program. For a false error, the test which attempts to reproduce the trace will fail.

Orchestration test: A Go file containing an Orchestrate function, where this function instructs Orc (a modified version of the original Source program), how it should proceed so that some trace can be reproduced.

Trace disallowing: Given a query and a trace for an error produced by that query, 'trace disallowing' means modifying the query, such that if it is run again, it will not produce the exact same trace as before.

The algorithm is composed of three stages; the first pass, the second pass and the deep pass. Each of these stages executes sequentially, and will be explained in detail now:

The first pass

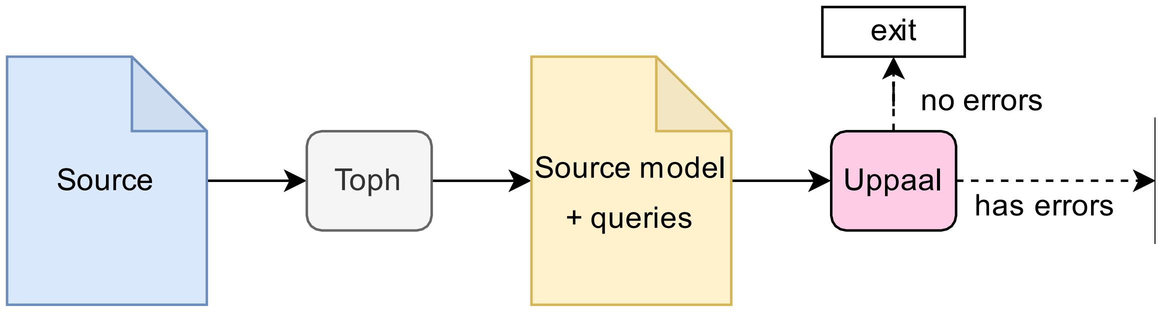


Figure 3.2: Isolated diagram of the first pass

In the first pass, the Source program is translated with Toph, producing an Uppaal model known as the Source model, and some queries relating to the model. Using Uppaal, these queries are then checked for the Source model. If no queries fail, then Vtrace can exit, as this indicates that the Soure program has no errors. However, if any queries fail, then Vtrace must proceed so that these errors can be investigated and determined real or false.

While first pass is useful, traces that come from errors are not particularly useful and cannot be reconstructed within Source, since the transitions between states would proceed non-deterministically. Therefore, it is necessary to produce a new version of Source, through which program flow can be finely controlled and traces can be reproduced.

The second pass

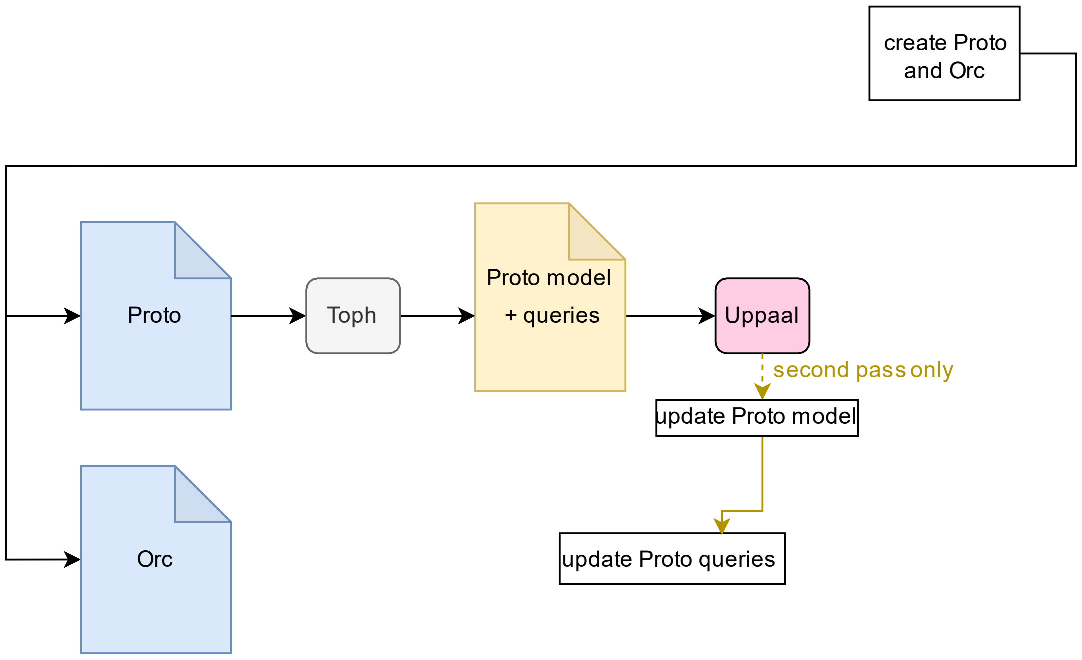


Figure 3.3: Isolated diagram of the second pass

At the start of the second pass, two new versions of the Source program are created, Orc and Proto. Orc (short for Orchestrated) is the version of Source through which program flow can be precisely controlled. Proto (short for Prototype) is a version of Source which is used to generate traces that can be built into Orchestration tests to run with Orc.

While Orc and proto are created at the same time, the second pass stage only makes use of Proto, with Orc being used afterwards in the deep pass stage. In this second pass stage, Toph is used to translate Proto into an Uppaal model, and to generate queries for this model's properties. It should be noted that the properties checked by the queries for the Source model will also be checked by the queries generated for the Proto model. However, the nature of Proto (i.e. the modifications that it contains with respect to Source) will mean that Toph also generates a large amount of additional queries for the Proto model.

Next, the queries for the Proto model are checked for the model using Uppaal. After the queries have been checked, the Proto model is updated to prepare it for the deep pass stage. This updated version of the model contains functionality that allows for trace disallowing to be performed later in the deep pass. Finally, any queries which did not produce errors are removed from the set of queries that accompany the Proto model. This query removal process ensures that the deep pass does not spend time searching for traces from queries that will never produce errors.

The deep pass

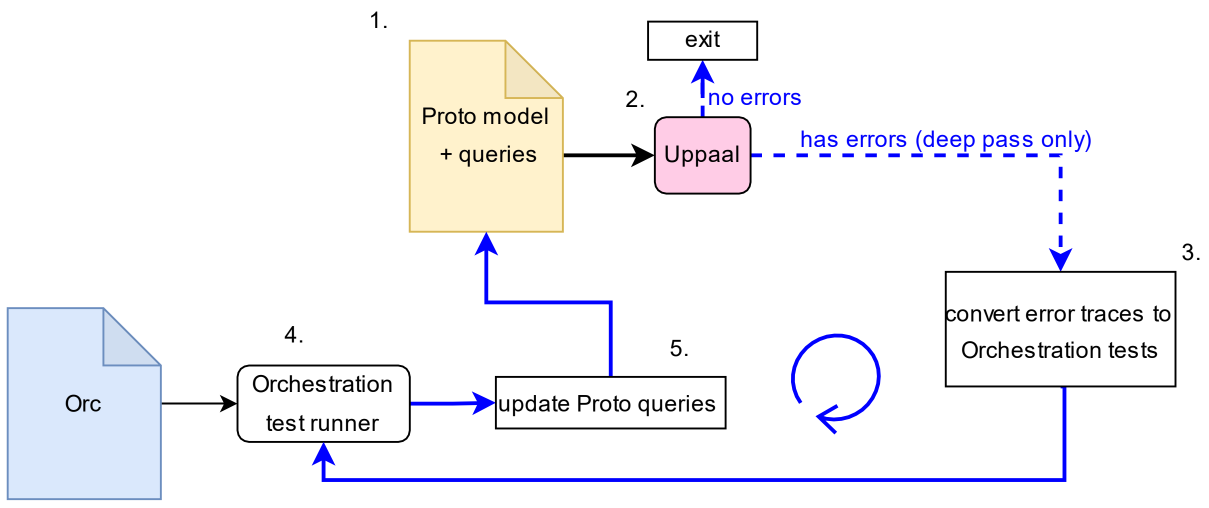


Figure 3.4: Isolated diagram of the deep pass

The deep pass is the main loop of Vtrace. It begins at after the end of the second pass stage, with a Proto model that can perform trace disallowing, and has queries that are known to produce errors. These queries are again run on the model using Uppaal. If no errors are produced, then Vtrace exits (note that for the first iteration of this loop, errors will alwys be produced). The traces from these errors are then converted into Orchestration tests. These Orchestration tests then get combined with Orc, compiled into executibles and run by the test runner, which will report to the user whether the trace reconstructed in the Orchestration test corresponded to a false or real error.

Next, the queries for the Proto model are updated so that any queries which correspond to real errors are removed, and any queries which correspond to false errors have their traces disallowed. This main loop then repeats to some bound defined by the user, or until no more errors are produced (which will also occur if all queries for this model have been removed). On each iteration, the trace disallowing means that a different, distinct trace is generated.

Since trace disallowing eliminates paths to an error, it is possible to run this deep pass until all traces to the error that Uppaal can produce have been produced and proven to correspond to false errors. In this case, the error is known to be false. In practice however, models tend to have so many possible traces to any given error that running the deep pass to verify that all of them are false is impractical or infeasible.

3.2 Algorithm components

Source

Source is an entire Go program – a collection of Go files under a package structure. Specifically, the root directory for Source is the ‘main’ package for a Go program and contains a file with the program entry point (i.e. a file containing the function ‘main’). All Go files which exist within this directory and in any sub-directories are considered to make up Source. External packages (packages not existing under the main package) can be imported, as can packages included in the standard library[8].

Toph translation

All Uppaal models used by Vtrace are created for Uppaal version 4.1.24, and are created using Toph. Each model is contained inside an XML file, with this XML file also specifying the queries that Toph has generated for this model.

Each model is a system of smaller sub-models, where each of these sub-models emulates some functionality from the original Go program. Specifically, sub-models either emulate some function within the Go program (for example, the main function will be a sub-model), or some behaviour of a type which offers concurrency to the program. The following types have their behaviour emulated within an Uppaal model: channel, sync.WaitGroup, sync.Mutex.

By default, Toph generates as many queries as possible given the contents of the model. Toph supports flags that allow for only certain queries to be generated (for example, a flag can be set so that only channel-safety queries are generated). Vtrace in turn has a flag allowing users to specify flags that they wish to pass into Toph during model generation.

Toph also allows for flags to be set which signal Toph whether or not some optimisations should be performed when generating Uppaal models. One of these flags instructs Toph to optimise the Go program’s intermediate representation (IR), and the other instructs Toph to optimise the Uppaal models produced. Allowing Toph to perform these optimisations leads to a model with a reduced number of states and thus, faster verification times. For Vtrace, these optimisations are used when creating the Source model, allowing for the first pass to complete faster. Since Proto and Orc use the states in the first pass model in their construction, use of these optimisations for the first pass also leads to a reduction in the number of states in the Proto model (see below detailing Proto).

These optimisations however, are not used when translating Proto into the Proto model, as their use led to the removal of some necessary elements of the Proto model, as well as some other states which are required to exist for trace disallowing.

Proto

Proto is a Go program which is functionally indisguishable from Source. This is to say that when either Source or Proto are compiled, both programs will have the same functionality and produce an identical output. Every every line of code in the Source program exists in Proto, as well as almost every comment.

Despite this, Proto has two key differences from Source. The first difference is the inclusion of **Guards**; blocking channel operations placed over every line from the Source program which corresponds to a state in the Source model. The second is the inclusion of the **Scheduler** function; a function that gets started in a separate goroutine at the beginning of

[8] *Go Standard library packages*, https://pkg.go.dev/std (last accessed 2021-05-10)

the main function. This Scheduler function contains a loop with a select statement[9], where every Guard in Proto has a corresponding case statement in this select statement. The Scheduler function ensures that if program flow encounters a blocking Guard operation it will immediately become unblocked and proceed beyond it, meaning that the programs execution is not hindered by the inclusion of these Guards.

As an example of this, take a line from an example Source program which tells the program to make a channel 'chA' as shown in (3.1).

```
chA := make(chan string)
```

(3.1)

In the Source model, calls to create a channel are represented as states, meaning that in Proto, this line should have a Guard before it. As a result, the following Guard line shown in (3.2) will exist in Proto:

```
<-PROTO_scheduler_autogen.ChPROTO_main_1
chA := make(chan string)
```

(3.2)

Here, the Guard is the blocking channel operation attempting to receive from the ChPROTO_main_1 channel (recall that in Go, an arrow out of a channel denotes receiving, while an arrow into a channel denotes sending). When running Proto, this guard will be unblocked by the Scheduler function declared in a file with contents shown in (3.3):

```
package PROTO_scheduler_autogen

var ChPROTO_main_0 = make(chan struct{})
var ChPROTO_main_1 = make(chan struct{})
var ChPROTO_main_2 = make(chan struct{})
var ChPROTO_main_3 = make(chan struct{})
var ChPROTO_main_4 = make(chan struct{})

func Scheduler() {
  for {
    select {
    case ChPROTO_main_0 <- struct{}{}:
    case ChPROTO_main_1 <- struct{}{}:
    case ChPROTO_main_2 <- struct{}{}:
    case ChPROTO_main_3 <- struct{}{}:
    case ChPROTO_main_4 <- struct{}{}:
    }
  }
}
```

(3.3)

In this entire example program for which (3.3) was used, five states were present, and each state is blocked with a different Guard. The Scheduler function can be seen to constantly attempting to unblock each of these Guards through its select statement.

In some sense, the Scheduler function can be thought of as a program-wide hook. If Proto runs non-deterministically, some certain sequence of Guard unblocks will be performed by entering certain cases inside Scheduler's select statement, with this sequence of Guard unblocks corresponding to a path taken through the program.

[9] In Go, a select statement selects one viable channel operation to perform when given a number of possible blocking channel operations. These possible channel operations are provided in the form of 'case' statements. See https://tour.golang.org/concurrency/5 (last accessed 2021-02-10)

As this hooking exists within Proto and exists due to channel operations which Toph will never abstract away during translation, it will also exist within the Proto model. This means that traces produced from the Proto model will include information about the sequence of Guard unblocks necessary to reproduce the trace. Additionally, traces will include information about which function (which could be a goroutine) was allowed to progress upon unblocking any Guard. As such, the Proto model produces traces with the necessary information to build Orchestration tests for Orc.

Orc

Orc is a Go program which supports the execution of an Orchestration test. That is to say, Orc has all of the necessary lines for an Orchestration test to interact with when deterministically reproducing and verifying a trace. When supplied with any Orchestration test, Orc gets compiled into an executable and run by Vtrace's Orchestration test runner (although the user may manually perform this compilation if they wish), with the result of its execution indicating the validity of the trace that produced the Orchestration test.

Where Proto starts a Scheduler function in a goroutine at the beginning of the program, Orc makes a call to start a function called Orchestrate in a goroutine instead. This Orchestrate function is declared in an Orchestration test and is used to deterministically manage program flow in Orc.

Orc includes Guards at the same positions as Proto, since Orchestration tests are created from the traces provided by the Proto model and will need to unblock the same sequence of Guards to progress. However, Orc also notably features **OrcGuards**, goroutine-specific channels which perform blocking and unblocking operations. OrcGuards are placed before every Guard, and after every block of code that is executed when a Guard is unblocked.

In this way, OrcGuards can serve two purposes. Firstly, by placing them before Guards, an Orchestration test can control which goroutine is able to progress to unblock that Guard. Secondly, by placing OrcGuards at the end of blocks of code in between Guards, Orchestration tests can ensure that a goroutine has completed the 'transition' from one Guard to the next (analogous to transitioning between states in the Proto model).

As well as OrcGuards, Orc also features additional code that logs the control flow of the program as an Orchestration test is being played out. The logging will log the current line that the program has reached or is about to reach (depending on the context it may be preferable to log either before or after), as well as the name of the goroutine that has executed that line.

Logging lines in this fashion helps the user to understand exactly how the program has executed, and in the event that an error is found, this logging is a lot easier to interpret than the Proto, Orc or Orchestration test. This allows the user to distance themselves from the fine details of Vtrace's operation, ignore the complexity of the autogenerated Proto, Orc, or Orchestration tests, and to instead focus on the Vtrace's output and the errors it describes.

As in (3.1) before, the following example lines in (3.4) from a Source program can be used to illustrate the changes that are made in Orc.

```go
chA := make(chan string)
fmt.Printf("created new channel of type %T", chA)

// lock some mutex (this will be the next state)
mtx.Lock()
```

(3.4)

In Orc, the call to make a new channel will have a Guard (the same as was used in Proto), as well as logging before the make statement, and an OrcGuard before the Guard and after the code along the transition (i.e. before entry to the next state. Logging will also be done for all lines in the code (using the orcutil.PrintControlFlow function). Note that the line mtx.Lock() will correspond to a state in the Source model, and so reaching this state marks the end of the block of code that lies on the transition between states. The section of code in Orc relating only to the channel creation state and the transition to the following state is shown below in (3.5). The section of Orc relating to the mtx.Lock() line has been removed for clarity, to emphasise where the transition starts and ends.

```
<-ChORCHESTRATION
<-PROTO_scheduler_autogen.ChPROTO_main_1
orcutil.PrintControlFlow(ChORCHESTRATION, "chA := make(chan string)")
chA := make(chan string)
fmt.Printf("created new channel of type %T", chA)
orcutil.PrintControlFlow(ChORCHESTRATION, "fmt.Printf(\"created new channel of type %T\", chA)")

// lock some mutex (this will be the next state)
<-ChORCHESTRATION
```

(3.5)

Here OrcGuards are the blocking receive operations using the ChORCHESTRATION channel. Logging is done with the orcutil.PrintControlFlow calls, and when these lines are reached in an Orchestration test, the output shown in (3.6) will be logged showing the lines executed and the name of the goroutine which executed them (in this case 'func_5_main' – the main goroutine):

```
2021/04/08 22:58:22.719877 func5_main_0  :  chA := make(chan string)
2021/04/08 22:58:22.719877 func5_main_0  :  fmt.Printf("created new channel of type %T", chA)
```

(3.6)

Orchestration test

As previously mentioned, an Orchestration test is a Go file that declares the Orchestrate function, where this function instructs Orc how it should proceed so that a trace can be reproduced. In the deep pass stage, Orchestration tests are generated for each trace produced by Uppaal. The Orchestrate function declared in these tests is for the most part a sequence of sections of code, with the purpose of each these sections of code falling under one of the following categories:

1. Goroutine registration:
   At certain points in the trace, a new goroutines is spawned, or the program flow of an existing goroutines will enter into a function. In both of these cases, goroutine registration needs to occur. For newly spawned goroutines, a new channel used for OrcGuards needs to be registered and associated with that new goroutine. For the case where a goroutine enters a function, references to this goroutine from the trace will use the name of this function. To ensure that Orchestration test can still refer to the correct goroutine at any point, goroutines must re-register the channel they use for OrcGuards upon function entry.

2. OrcGuard and Guard unblocking:
   This is the most common section of code in an Orchestration test, and generally follows a simple unblock-unblock-wait structure. Its purpose is to reconstruct a Guard unblock that occurs in a trace from the Proto model. From the trace, the name of the goroutine or function in which the Guard unblock was performed is known, as is the

name of the Guard which was unblocked. From this, a section of code in the Orchestration test is constructed where the following actions occur:

   a. The OrcGuard for the specified goroutine is unblocked, allowing that particular goroutine to reach the Guard.
   b. The specified Guard is unblocked, allowing the goroutine to progress and execute the code along the transition.
   c. The Orchestrate function blocks itself using the same channel as was used by the OrcGuard before (in part a.), allowing it to wait until the program has completed its transition and reached the next state.

   When this unblock-unblock-wait structure is employed, all transitions can occur completely deterministically, replicating the trace perfectly.

3. Debug logging:
   In addition to the logs generated as the program runs, Vtrace's Orchestration test runner also writes debug information from each Orchestration test to the standard output as it automatically runs Orchestration tests. In the event that an Orchestration test corresponds to a real error, this debug information will include the standard error stream describing the error from the program's perspective. In the event that the Orchestration test corresponds to a false error however, the test runner will report how far the Orchestration test was able to progress (i.e. given an Orchestration test with N sections, the test runner will report how many of these sections were able to be executed, and what was the last section that executed to completion) .

Trace disallowing

As previously mentioned, trace disallowing involves modifying the queries given to Uppaal such that these queries will return new traces to any errors found by these queries. Before understanding how this works (see implementation), it is important to emphasise why this is required as a core part Vtrace's algorithm.

Trace disallowing is needed to deal with some of the abstractions that Toph uses when producing models. These abstractions can mean that even if a real error is present, the first trace produced for this error may take some path to the error which is not possible in Source, and so this error would be mistakenly classified as a false error. Vtrace therefore needs some way to generate new traces for the same error, so that it could find all possible traces to this error and verify that all of these traces correspond to false errors, or find the real error that lies somewhere among them.

One common example of a case where trace disallowing is used is in loops where the loop condition involves a variable, as shown below in (3.7):

```
for i := 0; i < x; i++ {
```
(3.7)

In this example, x is some variable that controls how many times the loop is repeated.

As Toph cannot reason about the value that the variable will have upon reaching the loop, this value is not included in the model meaning that while the model contains a loop, Uppaal has no idea how many iterations of this loop are expected in the Source program. When checking a query, Uppaal will therefore initially assume that only one iteration of the loop is needed and produce a trace where this occurs. In Source, a real error may exist beyond this loop, but since the trace specifies that only one iteration is needed while Source expects the loops to run for multiple iterations, then Vtrace will report that the trace corresponds to a false error as it cannot be reproduced. With trace disallowing, the query corresponding to this error is modified, so that the next time it is checked on the model (i.e. on the next iteration of the deep pass), Uppaal must generate a new trace. In this new trace Uppaal may report that the error is reachable after two iterations of the loop.

Uppaal queries and state space exploration

When checking a query on a model, Uppaal offers a selection of strategies for use when exploring the state space. In terms of deterministic strategies, Uppaal offers breadth-first and depth-first exploration. Since Uppaal will always cease exploring immediately upon encountering a state for which the query is false, each of these strategies will return only one trace for per query and running the query on the model multiple times will always produce this same trace. Using these two strategies, it is therefore possible to generate at most two distinct traces to an error. Uppaal also offers 'random-depth first' as an alternative non-deterministic exploration strategy. Random-depth first operates as its name suggests, exploring the state space to random depths until the query fails. Using this exploration strategy, all possible traces to an error can be found. While random-depth exploration can be used for generating new traces, it would still need trace disallowing to prevent Uppaal from being allowed to generate the same trace twice.

Breadth-first search is preferred due to the following reasons:

- Random-depth exploration is inconsistent: the unpredictable nature of random-depth exploration could mislead users of Vtrace. A user may uncover a real error with their program, make some slight modification, and then rerun Vtrace to determine if they have successfully fixed the error. If after running the deep pass stage for the same number of iterations, a real error has not been found, the user may incorrectly assume that they have fixed the error, unaware that Uppaal had randomly not produced the correct trace yet.

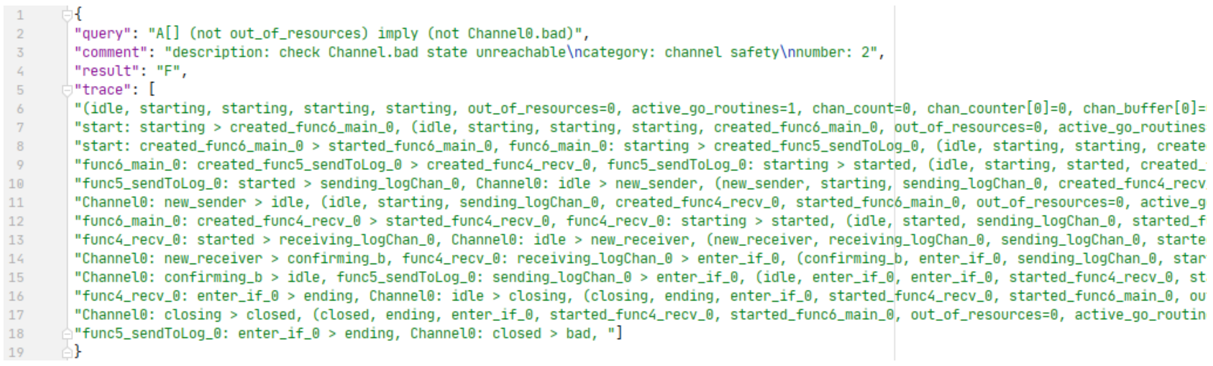
- Depth-first exploration traces prioritises unrealistic execution patterns: Depth-first exploration assumes that goroutines run to completion sequentially if lacking interaction (synchronisation/communication). This assumption is not representative of most programs, where goroutines are interleaved from the moment they are spawned, and errors arise from these interleavings. By comparison, breadth-first exploration prioritises interleaving goroutines together, even without interaction, which is a closer approximation of the way that the Go runtime actually interleaves goroutines. However, even with breadth-first exploration trace disallowing will eventually mean that the case where goroutines run sequentially is considered; but using breadth-first exploration means that this most common case where errors are due to interleaving is considered first.

4. Implementation

4.1 Trace extraction

One of the first parts of the implementation for this project involved finding some way to extract traces from Uppaal in a usable form. The executable used by Uppaal for model checking can be passed a flag that will instruct it to produce traces when verifying. However, the traces produced in this manner as not readable (states are not obviously labelled, and neither are transitions). To recover a trace where the states are readable, Uppaal's Java API can be used. This API allows for queries to return objects representing the trace, from which it is possible to find the transitions between states as well as the current variable values after every transition.

Therefore, the first stage of this project involved creating a Java program which, when provided with a model and queries for that model, could save these readable traces in a useable format. As Vtrace was to be written in Go which has built-in support for JSON encoding/decoding, information about traces from Uppaal is extracted and then exported in the form of JSON files. Below in figure 4.1 is an example of a JSON file that was created when a query failed on a Source model (full trace lines not shown).



```
{
"query": "A[] (not out_of_resources) imply (not Channel0.bad)",
"comment": "description: check Channel.bad state unreachable\ncategory: channel safety\nnumber: 2",
"result": "F",
"trace": [
"(idle, starting, starting, starting, starting, out_of_resources=0, active_go_routines=1, chan_count=0, chan_counter[0]=0, chan_buffer[0]=
"start: starting > created_func6_main_0, (idle, starting, starting, starting, created_func6_main_0, out_of_resources=0, active_go_routines
"start: created_func6_main_0 > started_func6_main_0, func6_main_0: starting > created_func5_sendToLog_0, (idle, starting, starting, create
"func6_main_0: created_func5_sendToLog_0 > created_func4_recv_0, func5_sendToLog_0: starting > started, (idle, starting, started, created_
"func5_sendToLog_0: started > sending_logChan_0, Channel0: idle > new_sender, (new_sender, starting, sending_logChan_0, created_func4_recv
"Channel0: new_sender > idle, (idle, starting, sending_logChan_0, created_func4_recv_0, started_func6_main_0, out_of_resources=0, active_g
"func6_main_0: created_func4_recv_0 > started_func4_recv_0, func4_recv_0: starting > started, (idle, started, sending_logChan_0, started_f
"func4_recv_0: started > receiving_logChan_0, Channel0: idle > new_receiver, (new_receiver, receiving_logChan_0, sending_logChan_0, starte
"Channel0: new_receiver > confirming_b, func4_recv_0: receiving_logChan_0 > enter_if_0, (confirming_b, enter_if_0, sending_logChan_0, star
"Channel0: confirming_b > idle, func5_sendToLog_0: sending_logChan_0 > enter_if_0, (idle, enter_if_0, enter_if_0, started_func4_recv_0, st
"func4_recv_0: enter_if_0 > ending, Channel0: idle > closing, (closing, ending, enter_if_0, started_func4_recv_0, started_func6_main_0, ou
"Channel0: closing > closed, (closed, ending, enter_if_0, started_func4_recv_0, started_func6_main_0, out_of_resources=0, active_go_routin
"func5_sendToLog_0: enter_if_0 > ending, Channel0: closed > bad, "]
}
```

Figure 4.1: Example of a JSON file containing a query formula, comment, result, and trace.

This JSON file contains the query formula responsible for producing this trace, the comment associated with this query, the trace itself, and the result obtained from Uppaal for this query - which can either be 'F' (query not satisfied), 'M' (query maybe satisfied), 'E' (error occurred for this query), or 'T' (query satisfied). In the case that a query's result is 'T', no trace will be produced. Within each line of the trace array, values contained within the round braces are the current variable values of variables in the model at each state, and the transitions between states take the form shown in (4.1).

$$X: A > B \tag{4.1}$$

Where A is an initial state, B is the next state that was reached after the transition. Note that while Uppaal does make a distinction between goroutines and functions as it runs, the syntax for traces does not make such a distinction, and so under this syntax X is the name of a function or goroutine in which this transition occurred.

4.2 Creating Orchestration tests

The three components of an Orchestrate function are implemented as follows:

Goroutine registration

Goroutine registration is done by passing the channel used by the OrcGuards for a currently-running function through a registration channel and to the Orchestrate function, where it can be stored in a map entry keyed by the name of that currently-running function. This function name is found from the trace which generates the Orchestration test. For example, if the trace specifies that a function or goroutine with the name "innerFunc" has been called or spawned from a function "outerFunc" after a Guard ChPROTO_main_12 was unblocked, it will represent this with the line in (4.2).

```
func7_outerFunc_0: received_ChPROTO_main_12_0 > created_func6_innerFunc_0
```
(4.2)

On this line, the name of each function is unique throughout the whole program, meaning that if innerFunc is called again somewhere else, it will be given another number as a suffix (e.g. the next call would be named func6_innerFunc_1). Regardless of whether this function is a new goroutine or not, all lines in the trace that are run within this function will have the form:

```
func6_innerFunc_0: A > B
```
(4.3)

Where A is the start state and B is the state reached following the transition. To register a channel as being used for OrcGuards for this function, the following line (4.4) in the Orchestrate function stores the channel in a map.

```
orcutil.LookupOrcChMap["func6_innerFunc_0"] = <-orcutil.RegisteringCh
```
(4.4)

OrcGuard and Guard unblocking

Lines of the trace will specify which goroutine is unblocking which Guard. For example, line below in (4.5) shows the line at which the Guard ChPROTO_main _10 is unblocked by the func7_outerFunc_0 goroutine/function:

```
func7_outerFunc_0: receiving_ChPROTO_main_10_0 > received_ChPROTO_main_10_0
```
(4.5)

When the Orchestrate function is created, the above line is converted into an unblock-unblock-wait sequence. First the registered channel for the func7_outerFunc_0 function is unblocked, then the Guard is unblocked, and then Orchestrate blocks itself using the same OrcGuard channel as before until the transition completes and the next state is reached.

```
orcutil.LookupOrcChMap["func7_outerFunc_0"] <- struct{}{}
ChPROTO_main_10 <- struct{}{}
orcutil.LookupOrcChMap["func7_outerFunc_0"] <- struct{}{}
```
(4.6)

The 'wait' part of this is excluded in the case where the Guard unblock is the last Guard unblock which occurs in that specific goroutine, since in this case, the function will exit after the last transition has completed (having a 'wait' for the last transition would also interfere with the operation of a utility function called OpenNextCh which handles goroutine/function exiting. This OpenNextCh function is explained in implementation section).

Debug logging

After every section in the Orchestrate function which performs either registration or unblocking, a function is called which writes a string version of that section to a temporary debug file which is read by Vtrace's test runner after the Orchestration test has completed. Accompanying this string version of the section is a number indicating how many sections are in the Orchestrate function in total, and the index of the last section that ran to completion. For example, after running the section with index 3 out of a total of 11 sections, which consisted of the lines in (4.7):

```
orcutil.LookupOrcChMap["func6_innerFunc_0"] <- struct{}{}
ChPROTO_main_10 <- struct{}{}
orcutil.LookupOrcChMap["func6_innerFunc_0"] <- struct{}{}
```
(4.7)

The debug logger would run the function:

```
orcutil.UpdateDebugLine(sendDebug, orcutil.Status{Line: 4, Max: 11, LastGoodLine:
`orcutil.LookupOrcChMap["func6_innerFunc _0"] <- struct{}{}\nChPROTO_main_10 <-
struct{}{}\norcutil.LookupOrcChMap["func6_innerFunc_0"] <- struct{}{}\n\n`})
```
(4.8)

The sendDebug boolean is set automatically, based on whether or not the Orchestration test is being run by the user or by the Vtrace test runner. UpdateDebugLine will only use logging if sendDebug is true.

4.3 Creating Proto and Orc

Static analysis and gofmt

Proto and Orc are created simultaneously at the start of the second pass through static analysis of Source (with state information from the Source model). The choice to generate these programs through static analysis (rather than working with an Abstract Syntax Tree) was made partially with the goal of readability and ease of use in mind – while Vtrace does aim to produce output which is sufficiently comprehensive that the user would not have to delve into the Proto or Orc versions and seek to understand or them in great detail, having the option to explore Proto and Orc so is a clear benefit. This is true in the case of both real and false errors, where it may be of interest for the user to inspect some parts of the Orc or Proto program to understand why and how an error was shown to be true or false.

However, the primary reason behind generating Orc and Proto as Go programs (i.e. directories and Go files) is that Toph needs to take an uncompiled Go program as an input, and therefore Proto is required to be of this form. As Proto and Orc share the same states, it makes sense to generate both at the same time.

In general, static analysis of a program however is not trivial, as people's unique coding styles can be troublesome to parse. This difficulty is lessened greatly (though not entirely) in the case of Go by the existence and ubiquitous use of the gofmt tool. With use of this tool being so standardised, Vtrace therefore makes the reasonable assumption that Source has been formatted according to gofmt and takes advantage of this fact when parsing Go files to generate Proto and Orc. As a simple example of some of the utility that gofmt brings, consider the following code (4.9) which has been formatted with gofmt.

```
1. x := someType{
2.   data: "test",
3. }
4.
5. if x.data != "test" {
6.   panic("something bad")
7. }
```

(4.9)

In this code, note that gofmt has left no space between the struct name ‘someType’ and the opening brace next to it, while in the ‘if’ statement a space has been added between the condition and the brace next to it. Exploiting this, Vtrace can distinguish the meaning of these two lines immediately, and treat the lines 1 – 3 as a struct (and for example, avoiding placing logging statements in the lines of this struct declaration), while treating the lines 5 – 7 as a block of code, where each new line would require a logging statement. Overall, gofmt is relied on by Vtrace as a form of filtering or pre-processing which is applied to the code, encoding information about types and content which would otherwise be challenging and inconvenient for Vtrace’s parser to acquire during Proto/Orc creation.

Extracting state information

Before Proto or Orc are created, the Source model must be inspected to determine which lines should be guarded. To do this, Vtrace uses the fact that Toph inserts comments in the model marking every element in the model with the filename and line of code that corresponds to it. To keep this line information in a useful form for later, Vtrace creates a map called ‘reducedLines’ mapping a string to a slice of integers (slices are dynamic arrays in Go). The XML file containing the Source model is then read line by line. For every filename found, an entry in the map is created, and for every entry in the map, a sorted slice of line numbers is maintained. Once fully populated, reducedLines is then returned to Vtrace for use in parsing.

File structures

All directories created by Vtrace are created under the parent directory of Source. In order for Vtrace to create Proto and Orc successfully, Source must be some directory which exists under the “src” directory under the directory specified by the users GOPATH environment variable. For example, Source is “D:\Users\Brian\go\src\mywork\prog\”, where GOPATH is “D:\Users\Brian\go\”. Vtrace creates Proto and Orc using the suffixes “_proto” and “_orc” and also adds a timestamp marking the Unix time that the Vtrace created them. Note that this timestamp is added by default but can be overridden and set to any positive integer value desired. Similarly, Vtrace creates a directory for the first pass with the “_first_pass” suffix (containing the Source model and any JSON files from traces extracted from it), and another directory for the second pass with the “_second_pass” suffix (which contains the Proto model and any traces extracted from it). This would result in the directory structure in (4.10):

```
%GOPATH%\src\mywork\
         ├─prog\
         ├─prog_first_pass_1624748400\
         ├─prog_orc_1624748400\
         ├─prog_proto_1624748400\
         ├─prog_second_pass_1624748400\
```

(4.10)

File writing

To create Proto and Orc, a separate goroutine is spawned for every Go file under the root directory of Source. Each of these goroutines runs a function called buildProtoFile that creates a Proto and Orc version of the Go file provided. Once all these goroutines have been spawned, another goroutine is spawned that runs a function called ManageMapCreation. The interaction between the buildProtoFile goroutines and the ManageMapCreation goroutine is described in figure 4.2 below. From within buildProtoFile, a struct called fCallsStore is responsible for collecting information that is gets sent to ManageMapCreation.

fCallsStore is an instance of a struct with the following CallsStore type:

```
type CallsStore struct {
  pkg         string
  store       []string
  importedPkgs []string
}
```
(4.11)

Here pkg is set to the package that the Source file is from, store is a slice of function identifiers from functions declared in Source file, and importedPkgs is a slice of packages imported in the Source file. Each of these fields is updated at various locations in the buildProtoFile function.

Described here (in order) is the sequence of operations performed by the buildProtoFile function.

1. **Import rewriting**
   Files in Source which import packages from Source have these imports replaced in Proto and Orc. For example, the import line
   ```
   import "mywork/prog/subdir”
   ```
   becomes
   ```
   import "mywork/prog_proto_1624748400/subdir”
   import "mywork/prog_orc_1624748400/subdir”
   ```
   in Proto and Orc respectively. This replacement is done only for packages that exist under Source, and any external packages are imported as before. All files in Proto and Orc are also modified to import the ‘PROTO_scheduler_autogen’ package, which exists under the root of Source. In Proto, this package is responsible for exporting the Scheduler function and Guard declarations, while in Orc, the Orchestration tests are placed under in this package and from here export the Orchestrate function and Guard declarations.
   In Orc, all files also import the package "github.com/brianneville/vtrace/project/orcutil" from Vtrace. This package provides utility functions used by the test runner when running Orchestration tests (see utility functions), and types used when modifying function definitions. This import rewriting is done as the Source file is read line by line. As this is done, the names of imported package are saved to fCallsStore as importedPkgs.

2. **Write Proto, and partially write Orc**
   Once the imports have been handled, the Source file is continued to be read until a line is found which begins a function definition. This function is checked to see if it is a one-line function (in the case it is, this one-line function is split into declaration and contents lines). Next, the name of this function is saved to fCallsStore. After this, all lines within this function definition are parsed sequentially, and Guards, OrcGuards and logging information are added at appropriate locations. In cases where line breaks are only used for visual purposes (such as long boolean expressions or struct declarations) lines are concatenated.
   If no lines are found that contain function definitions, then the Source file is copied to the locations of the Proto and Orc file, and buildProtoFile can exit.

Additionally, in Orc, some calls to utility functions are added accordingly, and function definitions are modified so that they receive an argument which is of the type of channel used by OrcGuards (these channels have type orcutil.OrcChan). By modifying function definitions, the channels used by OrcGuards are passed into functions, and so can continue to be used to block or unblock the goroutine which is running in that function.
For example:
A function:

```
func (v *view) handleMessage(message string) {
```

gets modified in Orc to have the definition:

```
func (v *view) handleMessage(ChORCHESTRATION orcutil.OrcChan,message string){
```

and will result in the following function identifier being added to fCallsStore:

```
".handleMessage,"
```

This identifier means that handleMessage is a function which is a receiver (due to the "." prefix), and accepts at least one argument due to the "," suffix).

Recall that an OrcGuard is a receive operation from the ChORCHESTRATION channel, and so by having a calling function pass the same ChORCHESTRATION into any called functions, Orchestration tests can continue to block and unblock the same goroutine regardless of what function it is in.

3. **Apply CallsMap to Orc**
After the entire Source file has been read, buildProtoFile will wait until all other Source files in the program have been read. At this point, fCallsStore is used to receive a map of all function names that could be present in the Source file, known as CallsMap. This map is specific to each package that the Source files are in. In this map, distinctions are made between functions that accept arguments and those which do not, as well as functions that are receivers on structs.
Now that buildProtoFile has callsMap, this callsMap is applied to the Orc file, meaning that the Orc file is read from start to finish again, line by line, and function calls which are found that have identifiers in the callsMap are modified so that they pass the ChORCHESTRATION variable into the function as the first argument. As in the example function above, any calls made to handleMessage would be modified so that instead of:

```
v.handleMessage(message)
```

they read:

```
v.handleMessage(ChORCHESTRATION, message)
```

The figure 4.2 describes the interaction between the ManageMapCreation goroutine and the buildProtoFile goroutines. Note that the dashed lines indicate conditional branches, meaning that the loop in ManageMapCreation will continue until all Source files have been parsed fully.

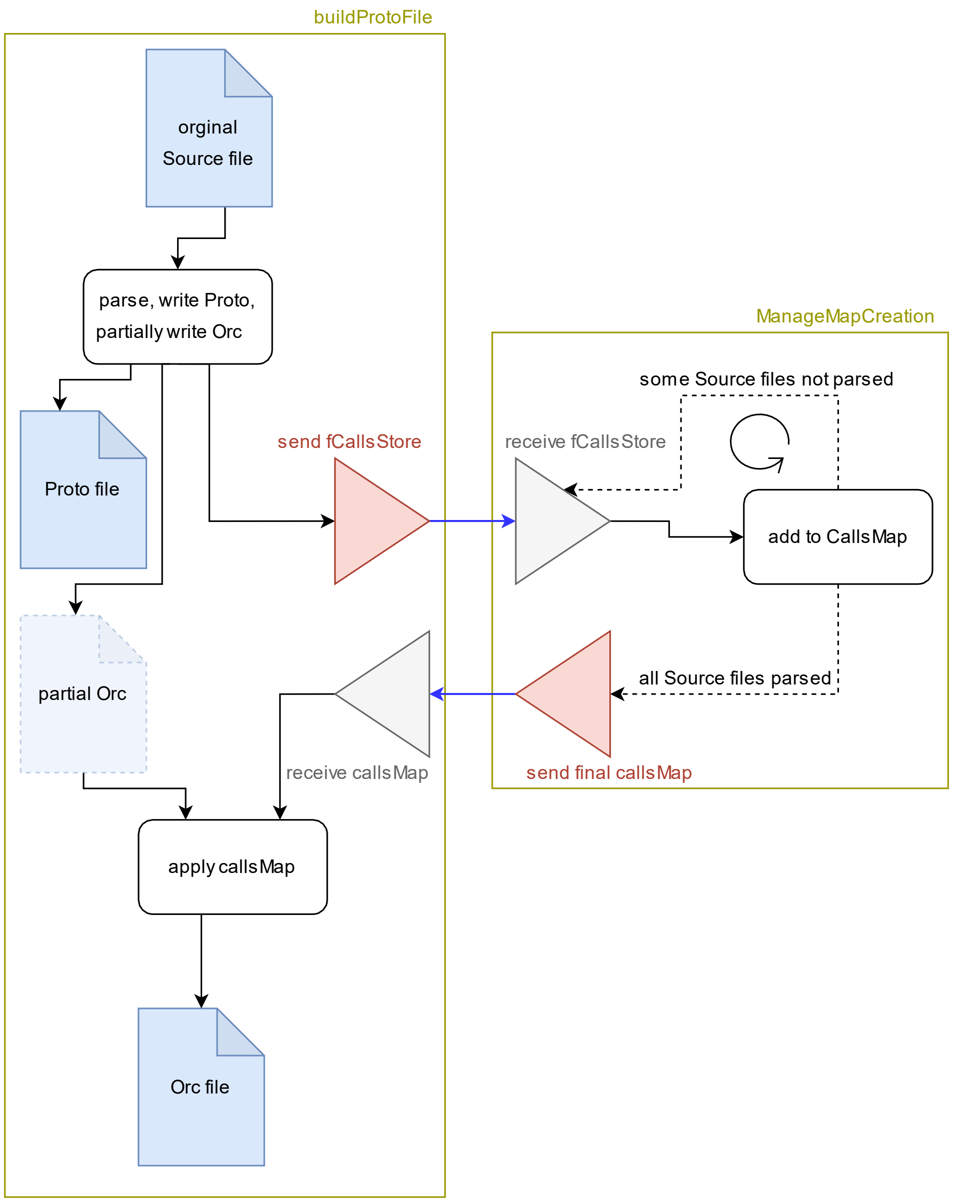


Figure 4.2: Interaction between the ManageMapCreation goroutine and buildProtoFile goroutines

Goroutine registration in Orc and Orchestration tests

In a trace extracted from the Uppaal, the transitions are labelled with the function they occur in. As previously mentioned, a transition has the format $X: A > B$, with X being either a function or a goroutine. As an example, in Proto the following lines (4.12) specify a guard and a call to spawn a new goroutine from the main goroutine:

```
<-PROTO_scheduler_autogen.ChPROTO_main_9
go checkVal(value)
```

(4.12)

When a trace includes spawning this goroutine, the trace will include lines specifying that a new goroutine is spawned, as in (4.13):

(4.13)

```
"func7_main_0: received_ChPROTO_main_9_0 > created_func6_checkVal_0,
"func7_main_0: created_func6_checkVal_0 > started_func6_checkVal_0, func6_checkVal_0: starting > started,
```

However, if somewhere else in the program the function checkVal is called again but not spawned in a goroutine, so that the program flow of the main goroutine continues into the function, then the program will contain lines shown in (4.14).

```
<-PROTO_scheduler_autogen.ChPROTO_main_14
checkVal(other_value)
```

(4.14)

Although the trace will also contain lines with the exact same format as (4.13), except for function instance number marking the function as different from func6_chechVal_0:

(4.15)

```
"func7_main_0: received_ChPROTO_main_14_0 > created_func6_checkVal_1,
"func7_main_0: created_func6_checkVal_1 > started_func6_checkVal_1, func6_checkVal_1: starting > started,
```

Therefore, in order to determine when a new goroutine has been spawned during Orchestration tests, Vtrace must make modifications from within the Orc file itself which support identification and reaction to new goroutines – the trace alone cannot be relied on to make this distinction.

To deal with these problem, Vtrace's parsing wraps any calls to create new goroutines in Orc within a new scope, and inside this scope redeclares the ChORCHESTRATION variable to a new channel. The call to spawn the goroutine is then made as normal, but now the ChORCHESTRATION variable that is being passed into the function has been assigned to a value unique to that newly spawned goroutine. Also, at the start of every single function, Vtrace adds a line of code that registers the ChORCHESTRATION variable with the Orchestration test, so that it can be used for the OrcGuards within this function.

By way of example using the checkVal function from above, the following are the lines that get executed in Orc. The call that does not spawn a new goroutine (4.14) is treated as a regular function call in Orc, meaning it gets treated as a normal function call.

```
<-ChORCHESTRATION
<-ChORCHESTRATION
<-PROTO_scheduler_autogen.ChPROTO_main_14
orcutil.PrintControlFlow(ChORCHESTRATION, "checkVal(other_value")
checkVal(ChORCHESTRATION, other_value)
```

(4.16)

The call that spawns a new goroutine (4.12) is wrapped in a new scope so that the ChORCHESTRATION variable can be redeclared, and in Orc it becomes:

```
<-ChORCHESTRATION
<-ChORCHESTRATION
<-PROTO_scheduler_autogen.ChPROTO_main_9
orcutil.PrintControlFlow(ChORCHESTRATION, "go checkVal(value)")
{
  ChORCHESTRATION := make(orcutil.OrcChan)

  go checkVal(ChORCHESTRATION, value)
}
```

(4.17)

Note that the ChORCHESTRATION value is only redeclared within this scope (i.e. the curly braces), and outside the scope, the value remains unchanged. Like all functions in Orc, the function checkVal also has a line added to the beginning of it, allowing this function to register whatever ChORCHESTRATION value it has received with an Orchestration test (note that the deferred call is explained later in the section on openNextCh).

```
func sendToLog(ChORCHESTRATION orcutil.OrcChan, val int) {
  defer orcutil.OpenNextCh(ChORCHESTRATION)
  orcutil.RegisteringCh <- ChORCHESTRATION
```

(4.18)

The result of these modifications is that Orchestration tests (which are reconstructed from traces and therefore have no awareness of whether a function is spawned in a goroutine or not), can simply maintain a map of the ChORCHESTRATION channels that are used by each function, and send to the correct ChORCHESTRATION values without knowing whether these are running in goroutines or not.

For example, in an Orchestration test the call to spawn checkVal in a new goroutine executes, and the new value of ChORCHESTRATION is sent into the orcutil.RegisteringCh channel. Within the Orchestrate function, the following lines register this value to the name of the newly created function (found from the trace lines).

```
orcutil.LookupOrcChMap["func6_checkVal_0"] = <-orcutil.RegisteringCh
```

Now, whenever the Orchestration test seeks to unblock the OrcGuards used by func6_checkVal_0, it sends to value stored in this map.

```
orcutil.LookupOrcChMap["func6_checkVal_0"] <- struct{}{}
```

Later, the call to run checkVal without a new goroutine, the same process occurs except this time, since ChORCHESTRATION was not redeclared in a new scope, the line

```
orcutil.LookupOrcChMap["func6_checkVal_1"] = <-orcutil.RegisteringCh
```

Will actually re-register the same ChORCHESTRATION that was used in the main goroutine to be used when unblocking the OrcGuards used by func6_checkVal_1.

Utility functions for Orc execution

While generating Orc, some additional lines of code are inserted in certain situations to allow Orchestration tests to manage situations. These lines make calls to functions that Vtrace exports under the orcutil package.

The first one of these functions is *BlockMainExit*. In BlockMainExit, a blocking receive operation is made on a certain channel which is never used anywhere else, and at no point in any Orchestration test is anything sent to this channel. The result is that BlockMainExit will block the calling function indefinitely. As its name suggests, BlockMainExit is used to block the main function from exiting. At the very start of the program’s main function, a call to BlockMainExit is deferred, meaning that if the main function is ever exiting normally (i.e. not exiting with a runtime panic or through a call to os.Exit), then the BlockMainExit function will called right before the function exits. By blocking the main function from exiting, Vtrace ensures that the Orchestration test gets a chance to run to completion if possible. In the case that the Orchestration test runs to completion, the Orchestrate function will make a call to os.Exit which will force the program to exit anyway. Without BlockMainExit, a real error which occurs as a result of code that executes after the last guard in the main goroutine has also been unblocked may not be caught. This last guard in the main goroutine may be unblocked either as part of a trace, or as part of a trace generated through trace disallowing.

```
var chLastBlock = make(chan struct{})

// prevent main goroutine from exiting early
func BlockMainExit() {
  <-chLastBlock
}
```

(4.19)

The orcutil.BlockMainExit function

For example, without BlockMainExit, an error in the program shown in (4.20) may go undetected if the main goroutine exits before A and B are finished communicating.

```
package main

func A(ch chan int) {
  v := <-ch
  if v == 0 {
    panic("something wrong")
  }
}

func B(ch chan int) {
  ch <- 0
}

func main() {
  ch := make(chan int)
  go A(ch)
  go B(ch)
}
```

program with a real error

(4.20)

(4.21)

```
func main() {
  defer orcutil.BlockMainExit()
```

The beginning of the main function in Orc

The second utility function exported by orcutil for use in Orc is the *UnblockOrchestrate* function. This function is used in locations around Orc to unblock the Orchestrate function, allowing it to progress past an OrcGuard which is blocking it and continue to run the rest of the trace. This may seem counterintuitive at first – one might think that if a trace (and thus the Orchestration test based on this trace) requires an OrcGuard to be unblocked, then it should only be unblocked from within the Orchestrate function, and failing to unblock this OrcGuard would indicate that the trace cannot be reproduced. Ordinarily it would be correct to think this, but there is one situation where this logic fails; that being the case where the line after a Guard is itself a blocking operation (i.e. a channel operation or a mutex lock/unlock). In this particular case, both the Orchestrate goroutine and the current running goroutine will be blocked; the current running goroutine will be blocked as it needs another goroutine to run to synchronise with, while the Orchestrate goroutine will be blocked as it is waiting for the current running goroutine to report that the code that lies along the transition between Guards has been executed. As a result, the program will enter an undesired deadlock state.

For example, if the program shown in (4.20) is run, one of either goroutine A or B will reach the line where they send or receive from the channel ch before the other goroutine, and will be blocked until the other goroutine reaches the line where it can send/receive and both goroutines can continue. As a result, Orc will add lines to A and B so that they make calls to UnblockOrchestrate.

UnblockOrchestrate has the following implementation, shown in (4.22) below.

```
// UnblockOrchestrate allows for the scheduler to progress so that the channel which 'syncs' with the
// current goroutine can be reached.
// Call this before any operation which blocks the current goroutine.
func UnblockOrchestrate(orcCh OrcChan) {
  select {
  case <-orcCh:
    orcCh <- struct{}{}
    break
  case <-time.After(500 * time.Millisecond):
    // we need to sleep for some time to ensure that the line which contains the send/recv
    // operation in all other goroutines is blocking
    break
  }
}
```

(4.22)

In the Orc version of the program in (4.20), the function B will have lines added so that it contains the following:

```
<-ChORCHESTRATION
<-PROTO_scheduler_autogen.ChPROTO_main_3
orcutil.PrintControlFlow(ChORCHESTRATION, "ch <- 0")
go orcutil.UnblockOrchestrate(ChORCHESTRATION)
ch <- 0
<-ChORCHESTRATION
```

(4.23)

While the function A will have lines added as such (shown only for the transition into the if statement):

```
<-ChORCHESTRATION
<-PROTO_scheduler_autogen.ChPROTO_main_0
orcutil.PrintControlFlow(ChORCHESTRATION, "v := <-ch")
go orcutil.UnblockOrchestrate(ChORCHESTRATION)
v := <-ch
if v == 0 {
  orcutil.PrintControlFlow(ChORCHESTRATION, "if v == 0 {")
  <-ChORCHESTRATION
```

(4.24)

In the case where an Orchestration test is based on a trace which needs to progress beyond the line v := <-ch and into the if statement, the following lines (4.25) will be in the Orchestrate function:

```
orcutil.LookupOrcChMap["func5_A_0"] <- struct{}{}
ChPROTO_main_0 <- struct{}{}
orcutil.LookupOrcChMap["func5_A_0"] <- struct{}{}
```

(4.25)

This unblocks the OrcGuard for the goroutine A, then unblocks the ChPROTO_main_0 Guard and waits for A to complete the transition to the next state. Once the Guard has been unblocked, the UnblockOrchestrate function is run in another goroutine. However, UnblockOrchestrate receives from the channel used by the OrcGuard, and the Orchestrate function can progress to eventually unblock the ChPROTO_main_3 Guard. Once A enters the if statement and finally completes the transition, the UnblockOrchestrate function performs a send operation to unblock the 'end of transition' OrcGuard.

Up to 500 milliseconds of delay in UnblockOrchestrate is needed to ensure that the entire program does not deadlock while waiting for the Orchestrate function to move from the line where it is unblocking the Guard, to the line where it is waiting on the last OrcGuard. Typically, moving from one line to the next like this will take an order of nanoseconds on any modern hardware (note that all other goroutines are blocked at this point, and not adding any delay), but a maximum delay of 500 milliseconds was added to absolutely guarantee that this program does not deadlock.

The last utility function exported for use in Orc is *OpenNextCh.* OpenNextCh is a simple function that takes an argument of a channel used by an OrcGuard and starts a goroutine which sends to that that channel, unblocking it the OrcGuard, as shown in (4.26):

```
// Used to unblock <-ChORCHESTRATION in caller after entering new goroutine
// When entering a new function, a call to OpenNextCh is deferred
// so it can unblock the <-ChORCHESTRATION line after returning back to the caller function.
// In the case that this function is a goroutine, this deferred call does nothing.
func OpenNextCh(orcCh OrcChan) {
  go func() { orcCh <- struct{}{} }()
}
```

(4.26)

OpenNextCh is used in two crucial situations throughout Orc. Firstly, OpenNextCh is used just outside scopes where new goroutines are spawned, for example:

```
<-ChORCHESTRATION
<-PROTO_scheduler_autogen.ChPROTO_main_6
orcutil.PrintControlFlow(ChORCHESTRATION, "go A(ch)")
{
  ChORCHESTRATION := make(orcutil.OrcChan)

  go A(ChORCHESTRATION, ch)
}
orcutil.OpenNextCh(ChORCHESTRATION)
```

(4.27)

Here, OpenNextCh is needed because the Orchestrate function can only verify that the checkVal goroutine has been spawned correctly by checking that the transition from the ChPROTO_main_6 Guard has completed inside a new goroutine, as such:

```
orcutil.LookupOrcChMap["func7_main_0"] <- struct{}{}
ChPROTO_main_6 <- struct{}{}

orcutil.LookupOrcChMap["func5_A_0"] = <-orcutil.RegisteringCh
orcutil.LookupOrcChMap["func5_A_0"] <- struct{}{}
```

(4.28)

The call to OpenNextCh in (4.27) means that the next OrcGuard in the main function will be unblocked, as though the transition from the ChPROTO_main_6 Guard to the next state has completed.

In addition to this, a call to OpenNextCh is deferred at the start of every function excluding the main function. In the case that the function is run in a goroutine this deferred call to OpenNextCh has no effect, but in the case where the function is called by some other function, this deferred call to OpenNextCh will result in the OrcGuard which is at the end of the transition in the calling function to be unblocked. For example, consider the following program (4.29) which has a real error introduced by the panic call.

```go
package main

func X(ch chan int) {
  <-ch
}

func Y(ch chan int) {
  ch <- 0
}

func main() {
  ch := make(chan int)
  go X(ch)
  Y(ch)
  panic("panic")
}
```

(4.29)

In Orc, both functions X and Y have lines which make deferred calls to OpenNextCh, as such:

```go
func Y(ChORCHESTRATION orcutil.OrcChan, ch chan int) {
  defer orcutil.OpenNextCh(ChORCHESTRATION)
```

(4.30)

```go
func X(ChORCHESTRATION orcutil.OrcChan, ch chan int) {
  defer orcutil.OpenNextCh(ChORCHESTRATION)
```

(4.31)

When the goroutine running X exits this deferred call has no effect, as this goroutine will never be started again. However, when the function call to Y exits, the deferred OpenNextCh call will unblock the OrcGuard which is at the end of the transition in the calling function, meaning that the program will be able to progress beyond OrcGuard at line 68 in (4.32).

```go
64. <-ChORCHESTRATION
65. <-PROTO_scheduler_autogen.ChPROTO_main_6
66. orcutil.PrintControlFlow(ChORCHESTRATION, "Y(ch)")
67. Y(ChORCHESTRATION, ch)
68. <-ChORCHESTRATION
69. <-ChORCHESTRATION
70. <-PROTO_scheduler_autogen.ChPROTO_main_7
71. orcutil.PrintControlFlow(ChORCHESTRATION, "panic(\"panic\")")
72. panic("panic")
```

(4.32)

Unlike UnblockOrchestrate, the OpenNextCh function cannot incorporate any kind of timeout. This is because Vtrace is not aware of how long it will take for the transition to occur (for example, if some code was written in between line 67 and 68, there is no way of knowing how long it will take to run. As a result, for every goroutine spawned in the original program, one additional goroutine will be spawned that is blocked for the remainder of the program.

4.4 Parsing

When explaining how Orc and Proto are created in the buildProtoFile function, it is mentioned that each function is parsed line by line, and that some effort is expended to ensure that lines are parsed appropriately with Guards and logging function calls are only inserted at appropriate locations. This sub-section will go into further detail about how this parsing is carried out.

One-line functions

In Go programs formatted with Gofmt, functions can be declared with both their definition and contents on the same line. For example, the following one-line functions which have been formatted with Gofmt:

```
func (c *calcType) String() string { return fmt.Sprintf("%d", c.value) }

func (c *calcType) CopyAdd(i int) *calcType { return &calcType{value: c.value + i} }
```

(4.33)

These one-line functions can be identified by the fact that every opening curly brace has an accompanying closing curly brace. When encountering any new function definition in the file, Vtrace's parsing checks the line that the function is declared on to see whether it is a one-line function. If it is, the function will be split into definition and contents sections. This splitting takes advantage of the way that Gofmt formats one-line functions. In particular one-line functions formatted with Gofmt will only have an opening curly brace followed by a space (i.e. "{ ") at the start of the contents line, and will only have a space followed by a closing curly brace (i.e. " }") at the end of the contents line. For example, the above functions (4.33) will be split into separate lines so that they are parsed as though they were declared:

```
func (c *calcType) String() string {
  return fmt.Sprintf("%d", c.value)
}

func (c *calcType) CopyAdd(i int) *calcType {
  return &calcType{value: c.value + i}
}
```

(4.34)

After one-line functions are split into lines these lines are passed to the same function which handles lines read within regular multi-line functions, known as writeLineWithinFunction.

writeLineWithinFunction

WriteLineWithinFunction is perhaps one of the most important functions in Vtrace. Its purpose is to parse a line provided to it, potentially concatenate this line onto the next line, and then write the line to Proto and Orc. When the line is being written, writeLineWithinFunction determines the placement for the Guards, OrcGuards and Logging for this line, making sure that these additional inclusions are not placed in any way which would cause errors during compilation (all inclusions must be added such that they do not disrupt the program code). WriteLineWithinFunction has the following definition (4.35):

(4.35)

```
func writeLineWithinFunction(line string, writer, writerOrc *bufio.Writer, scanner *bufio.Scanner,
  guardGen *guardGenerator, orcGuardGen *orcGuardGenerator, fCallsStore *calls.CallsStore,
  flags *buildFlags, reducedManager *reducedLinesManager) error {
```

With:

line – the line read from within a function from a file in Source.
writer, *writerOrc* – the writers for writing to the version of this file in Proto and Orc respectively.
scanner – a scanner which reads lines from the version of this file in Source.
guardGen, *orcGuardGen* – structs that manage the generation of Guards and OrcGuards.
fCallsStore – used to communicate function information to the ManageMapCreation goroutine.
flags – a struct of boolean values which store information discovered from previous lines of the function.
reducedManager – used to lookup whether the current line is mapped to a state in the Source model.

The first thing that writeLineWithinFunction does is iterate over the line character by character, collecting information about the line from characters which are not in comments. As it iterates, the index of the last valid character (the last non-whitespace character which is not part of a comment) is maintained, as well as the current difference between the number of opening and closing curly braces and round braces. Also, booleans track whether or not the line contains a blocking channel operation or an unblocking sync.Mutex.Unlock() call (which will be used later to insert a call to orcutil.UnblockOrchestrate) – as while sync.Mutex.Lock calls are represented with states in the Source model, sync.Mutex.Unlock calls only exist along transitions. Booleans also track whether or not this line is a comment, and whether or not this line will force the enclosing scope to return or not.

Once this line has been read in its entirety, the last valid character is inspected and used to determine whether or not this line is incomplete. For an incomplete line, the line ends with some operator or symbol, and the line break between the remainder of the line is for visual effect only. For example, lines ending in &&, || , &, |, + (but not ++), - (but not --), *, / (but not //), <<, >>, ^, =, or a comma character are all continued on the next line. Therefore, in the case of these incomplete lines, the next line is needed to properly parse the line given to writeLineWithinFunction. This is why the scanner is supplied to this function. Using the scanner, the next line is read and is concatenated onto the previous line at the index of the last valid character. The line is then parsed again, beginning from the index of the last valid character. The last valid character index is important for this concatenation purpose, as it allows for lines which have comments at their end to be concatenated without the comment overriding the concatenated line. For example, the following struct (4.36) which has comments in its definition:

```
testUser := User{
  name: "test", // this goes
  /*keep this*/ age:/*and this*/ 65,
  id: 6782341 ^
    /*this too*/ 101010101,
} // and the ending one
```

(4.36)

These lines will be rewritten by Vtrace during parsing so that it they are concatenated, and that any comments which interfere with the concatenation will be removed. The result of this concatenation and comment removal is that this line will be present in Orc and Proto as (4.37):

(4.37)

```
testUser := User{name: "test",/*keep this*/ age:/*and this*/ 65,id: 6782341 ^/*this too*/ 101010101} // and the ending one
```

With the comment "// this goes" removed so that the next line can be concatenated and not become a comment.

This line concatenation is important as it is one of the methods used by Vtrace's parsing to ensure that logging calls can be placed after every line, and not in the middle of any statements of code (which would produce a compilation error).

Once every character in the completed line has been parsed, a few additional checks are performed on the line, checking for keywords at the start of the line of code, such as "return", "goto", "panic(", "close(" , "break", "continue", "type", "case ", "select", "switch", "default:" or "func". A set of booleans are assigned based on combinations of the results of these checks. Additionally, the current line number is looked up using the reducedManager and a boolean is set. This boolean indicates whether or not the current line should have any Guards (lookups in reducedManager will return true in the case that the Source file name and line number are present as a state in the Source model).

Next, writeLineWithinFunction has a chain of if-else statements, where each branch of the if-else statement is taken based on a combination of all boolean values for the line. Based on the conditions, one of these branches will be taken and the line, logging function, and Guard (with accompanying OrcGuards in Orc), will be written using *writer* and *writerOrc* in a particular order, summarised in (4.38).

(4.38)

```
if conditionA {
  /* write order:
   Line
   Guard (+OrcGuards)
   Logging
  */
} else if conditionB {
  /* write order:
   Guard (+OrcGuards)
   Logging
   Line
  */
} else if conditionC {
  /* write order:
   Guard (+OrcGuards)
   Line
   Logging
  */
} else {
  /* write order:
   Logging
   Line
  */
}
```

The need for different orders arises from the multitude of different possible lines and blocks of code which can be written. For example:

The conditionA is true for function definitions and 'case' statements. For lines which are function definitions, the order of writing is necessary to ensure that no lines are written above the line which defines the function, as channel operations (Guards and OrcGuards) and function calls (logging) are not allowed outside of functions. Similarly, for lines which are case statements, channel operations and function calls must be written inside the case statement in order to be valid Go code. For lines which define functions (apart from the program's main function), it is at this point where functions definitions are modified to accept the OrcGuard's channel as an argument. It is also at this point where the function is added to *fCallsStore*.

While managing function definitions, the branch in conditionA handles the program's main function slightly differently. For the main function, the call to start the Scheduler function in a separate goroutine is inserted in Proto, and the call to start the Orchestrate function in a separate goroutine is inserted in Orc. Also inserted at the start of Orc is a line of code which assigns the ChORCHESTRATION variable for the main goroutine and registers it for use in the Orchestration test.

The conditionB and conditionC have much less concrete usages, and the choice to use one or the other is a lot more complicated and nuanced, and may need to account for both the information of the line and the state of the *buildFlags* variables set when parsing previous lines. However, one simple example of a situation where conditionB would be true is above lines that return from functions. In this case, it makes sense that the line with the logging, Guard, and OrcGuards would be above the return line, as they would be unreachable if they were placed below the return line. An example of a situation where condition C would be true is for a line with a function call in it, where this function call is an import from some external package or the standard library. In this case, it only makes sense to log that the function call has been executed after the function call returns (i.e. logging should be placed underneath the line). If the function is defined in the program, then the logging will be before the line (using condition B), showing that the goroutine has been started, since it may not progress beyond the function definition until later in the test.

The default case is for when all conditionA, conditionB and conditionC are false. In this case no Guards and OrcGuards are written, and logging is added only over the line. An additional condition is added when writing the logging line, just to ensure that this logging line is never written below lines that return to a higher scope, or below 'select' or 'switch' statements (since this would create code that cannot be compiled).

4.5 Running orchestration tests

Detecting real errors

Vtrace's test runner will automatically run Orchestration tests, and so it needs to be able to distinguish between tests which correspond to real errors, and those which correspond to false errors. To do this, Vtrace's test runner operates under the assumption that any time a program running with an Orchestration test deadlocks, that Orchestration test corresponds to a false error. This follows from the logic that for false errors, the Orchestration test has an Orchestrate function which specifies some sequence of Guard unblocks that are impossible to recreate, and deadlock will occur as it will try to unblock a Guard which cannot be unblocked or reached.

In the case of real errors, the Orchestrate function will execute every section of code within it, and at the end will call os.Exit with a custom error code (error code 13) which Vtrace's test runner identifies as verification that the Orchestration test corresponds to a real error. In cases where the program exits with a runtime error which is not a deadlock, Vtrace's test runner treats it as a real error.

Deadlock detection relies on the go runtime's deadlock detector, which will report a deadlock if a goroutine blocks itself while all other goroutines are also blocked.

Note that relying on deadlocks to check whether or not an error is false does not mean that Vtrace cannot verify traces which result from queries relating to program deadlocks. In the case that a trace specifies a sequence of Guard unblocks that it expects would lead to a deadlock, an Orchestration test will be produced which verifies whether this sequence is possible. If it is possible, the program will now no longer deadlock, since the Orchestrate function will still be running. At this point however, the Orchestrate function will have completed all sections, and will call os.Exit with error code 13 to end the program and let the test runner know that a real error has been found.

Trace disallowing

Trace disallowing is used in the deep pass stage to continuously provide Vtrace with new traces for any particular queries which Uppaal finds errors for, but which Vtrace finds are false errors. When considering how to implement this, it became apparent that modifications to the Proto model were unavoidable, as Uppaal's API does not offer any controls for preventing certain traces, and so the only way to prevent traces from being returned is to modify the model itself such that it internally supports preventing traces. To achieve this, the model needs to somehow maintain information about the trace. Since traces can already be identified by the sequence of Guard unblocks that occur, this information can be found by tracking the Guard unblocks that occur in the select statement in the model of the Scheduler function.

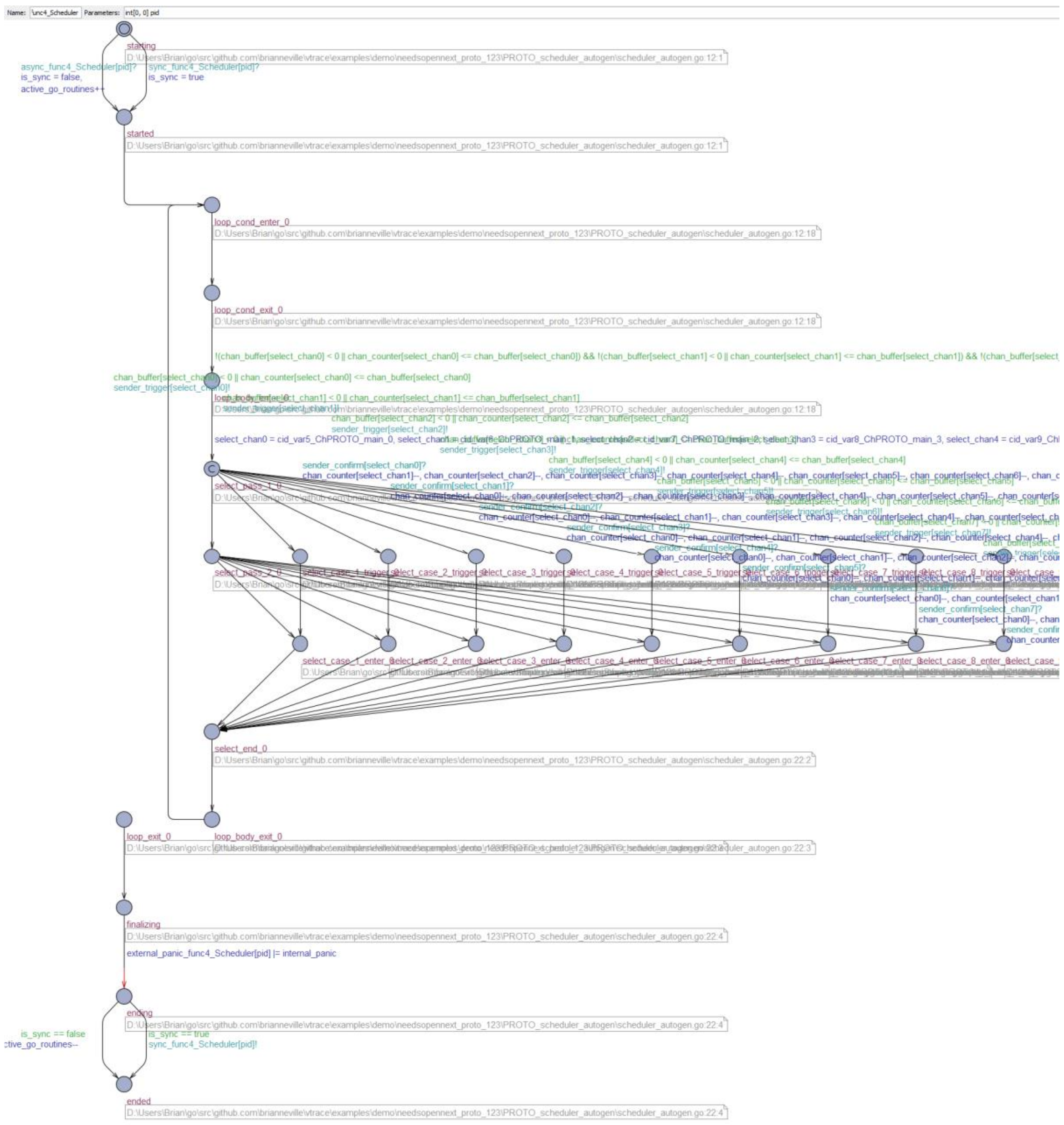

Name: func4_Scheduler Parameters: int[0, 0] pid
starting
async_func4_Scheduler[pid]?
is_sync = false,
active_go_routines++
sync_func4_Scheduler[pid]?
is_sync = true
started
loop_cond_enter_0
loop_cond_exit_0
loop_body_enter_0
select_pass_1_0
select_pass_2_0
select_end_0
loop_exit_0
loop_body_exit_0
finalizing
external_panic_func4_Scheduler[pid] != internal_panic
ending
is_sync == false
active_go_routines--
is_sync == true
sync_func4_Scheduler[pid]!
ended
D:\Users\Brian\go\src\github.com\brianneville\vtrace\examples\demo\needsopennext_proto_123\PROTO_scheduler_autogen\scheduler_autogen.go:12:1

Figure 4.3: Example Uppaal model of a Scheduler function.

Above in figure 4.3 is the Upppaal model as generated by Toph for a Scheduler function with 9 case statements. Each case statement is represented by two nodes, connected with four transitions. Information about the trace can therefore be collected by adding ‘update’ operations on these conditions, so that a function call is made or a variable is updated every time the transition is taken. Ideally, each of these update operations would simply append the index of the transition onto a list, and so at the end of the trace, this list would contain the full history of all Guards unblocked. Unfortunately, Uppaal does not allow for memory to be allocated dynamically during verification, and rather it only supports arrays with a fixed length. Therefore, to store the history of the trace, a hash is computed and stored in a variable which is updated on every transition. The FNV-1a hash[10] is used as a hashing function, in part due to its low collision rate, and largely due to its very fast compute time

[10] FNV-1a hash: http://www.isthe.com/chongo/tech/comp/fnv/ (last accessed 2021-05-09)

(computing the hash only involves one XOR and one multiply operation). To compute this hash, the following function (4.39) is added to the declarations for this Uppaal model:

```
typedef int[-2147483648, 2147483647] int32_t;
int32_t hash32 = -2128831035;
int32_t FNV_prime32 = 16777619;

void hash_path(int32_t id){
    hash32 = (hash32 ^ id) * FNV_prime32;
}
```

(4.39)

Toph is modified so that it can be passed a flag which instructs it to add this function in the declarations for the model, while also adding calls to this function along the transitions for the case statements in the model of the Scheduler function. On each transition, the index of the case statement is passed as an argument to hash_path (e.g. if the third case statement is entered, hash_path(2) is called). With this hashing in place, traces will report the value of the hash32 variable at every state, and the hash32 value can be read at the final state in the trace to get the hash which represents the entire trace.

The next part of the trace disallowing implementation is to prevent Uppaal from returning traces which have a hash that is known to be a false error. To achieve this, Vtrace takes the hash value from a trace which the test runner has identified as a false error and modifies the query which produces that trace so that it will hold true when the hash32 variable has that value. For example, take a query which has a formula that checks if a channel in a model can reach a bad state:

$$A[]\ (not\ out\_of\_resources)\ imply\ (not\ Channel14.bad)$$

If this query runs and produces a trace which concludes with the hash32 variable having a value of
-1657359148, then Vtrace will modify this query for the next iteration of the deep pass, such that its formula becomes:

$$A[]\ (not\ out\_of\_resources)\ imply\ (not\ (Channel14.bad\ and\ hash32\ !=\ -1657359148))$$

This new query formula will only become false if Channel14 reaches a bad state when hash32 is not the same as the value of hash32 which is known to produce a false error. This forces Uppaal to generate a trace which is different from the previous trace. In the case that Uppaal finds that this modified query formula holds true, then the original query formula is known to represent a false error. However, if this modified query formula does not hold, then the value of the hash32 variable is once again read from the last line in the trace and used to augment the query formula further. For example, if the hash32 value was 1372726421, then on the next iteration of the deep pass, the query formula would be modified to:

$$A[]\ (not\ out\_of\_resources)\ imply\ (not\ (Channel14.bad \\ and\ hash32\ !=\ -1657359148\ and\ hash32\ !=\ 1372726421))$$

Vtrace manages queries by maintaining a map mapping the number of the query (generated by Toph and placed found in the query's comment) to the query. For example, the above query formula could have an accompanying comment

"description: check Channel.bad state unreachable\ncategory: channel safety\nnumber: 2"

This number from the comment that is used when storing the query in a map. After each iteration of the deep pass, the queries which did not produce traces are removed from this map, while all remaining queries in the map have their formulae updated. The XML file containing the model is then modified by reading it until the part where the queries are declared, inserting every query declaration that still remains in the query map, and truncating the file after queries have been inserted.

4.6 Optimisations

Major optimisations

Three major optimisations are employed by Vtrace. The purpose of all of these optimisations is the same; to reduce the amount of time spent verifying queries which are known to not lead to errors. Details on the impact that each of these optimisations can be found in the evaluation section.

The first major optimisation is the second pass stage. Running queries with trace disallowing is a far more computationally expensive operation than running those queries without trace disallowing, and the second pass stage is used to ensure that only queries which actually produce errors are ever run with trace disallowing in the deep pass stage, which greatly reduces the amount of time per iteration in the deep pass stage, since the time for a query to fail and produce an error is generally vastly lower than the time taken for a query to be verified if this query will never produce an error.

The second major optimisation involves reducing the number of queries which are generated for the second pass stage (i.e. the queries that Toph generates for the Proto model). Specifically, this optimisation prevents Toph from generating queries for channels which have only been inserted into the program by Vtrace for use as Guards. Since these channels are not part of the original program, they will never produce errors in this program, and do not need to be checked. Additionally, since the behaviour of each of these channels is known and the way that they interact with the program (i.e. only receiving from the Scheduler function) is also known, these channels will never produce errors regardless. Since the number of channels used is always at least the same as the number of states in the Source model, allowing Toph to generate queries for these channels would mean that the a significant proportion of time in the second pass stage would be spent pointlessly verifying queries (scaling with the number of states in the Source model). To mitigate this, Toph is modified so that while the program is being translated, all channels which are found that have declarations in the same file as the Scheduler function (i.e. under some path ending in scheduler_autogen.go under the PROTO_scheduler_autogen directory) are ignored at the point where they would have queries generated. Additionally, all queries which would check that any ChPROTO channel can safely receive are ignored. Queries are also not generated for anything within the Scheduler function, by again skipping the processing which would otherwise be done for the scheduler_autogen.go file under the PROTO_scheduler_autogen directory.

The third major optimisation involves even further reduction of the number of queries which are checked in the second pass stage. However contrary to the previous optimisation which involves directly reducing the set of queries produced by Toph, this optimisation involves filtering the set of queries after they have been produced. The key idea behind this optimisation is to use the categories that Toph adds into the comments of every query to determine in advance which queries could possibly fail for the Proto model, given the categories of queries which have failed for the Source model. Any categories of queries which had no failing queries for the Source model will also have no failing queries for the Proto model, and so do not have to be checked at the second pass stage. For example, if the Source model has queries which have the category “channel safety”, and none of these queries produce errors, then any queries which are generated for the Proto model that are of the “channel safety” category also do not need to be checked during the second pass stage.

Minor optimisations

Vtrace also adds one minor optimisation when parsing to reduce the amount of Guards added where possible. This optimisation exploits the fact that every function definition corresponds to a state in the Source model. These function definition lines therefore need to have Guards, and the Guards for these lines will be placed directly underneath the function definition. In the case that the next line in the function which needs a Guard and is also not in a new scope (e.g. this next line is not inside a the branch of an ‘if’ statement or a loop body), then Vtrace can combine the two Guards into one, leading to fewer states in the Proto model and a faster verification time.
For example, consider the function (4.40).

```
func X(v int) {
  Y(v)
}
```

(4.40)

For this function, the line for the function definition will be a state in the Source model, as will the line for the call “Y(v)”. Rather than have these two Guards placed after each other in the same scope with no other state in between, a Guard is placed after the function definition and no Guard is placed before the call to the function.

```
func X(v int) {
<-PROTO_scheduler_autogen.ChPROTO_main_2
<-PROTO_scheduler_autogen.ChPROTO_main_3
Y(v)
<-PROTO_scheduler_autogen.ChPROTO_main_4
}
```

Unoptimised version

(4.41)

```
func X(v int) {
<-PROTO_scheduler_autogen.ChPROTO_main_2
Y(v)
<-PROTO_scheduler_autogen.ChPROTO_main_3
}
```

Optimised version

(4.42)

**this optimisation is also applied to the version of this function in Orc*

4.7 Flags for Vtrace

The following flags are used by Vtrace:

-pkg: (string)This flag specifies the location of Source, the main package for a program.

-proto-pkg: (string) This flag specifies the location of a Proto directory, if skipping the first pass stage

One of either pkg or proto-pkg must be set. The proto-pkg flag can be used to specify the directory of a Proto version of a program (using either a relative path or absolute path). This should be used only in the case that all other directories and files generated by Vtrace still exist (see the directory structure at (4.10)). Using the proto-pkg flag means that Vtrace will skip the first pass and begin at the start of the second pass. By default, the pkg flag should be used, but the proto-pkg flag exists for the cases where either the first pass takes a very long time, or where the user wishes to make some modification to the Proto and Orc file before re-running the second pass (for example, adding in custom logging calls or configuring some variables used in the program.

-reduced: (bool) This flag has a default value of true. Setting it to false will mean that every single line in the program will be given a Guard and OrcGuards. Leaving it as true will mean that only the lines which have states in the Source model are given a Guard and OrcGuards.

-timestamp: (int) This flag has a default value of -1, which indicates that the current Unix time should be used when creating directories (e.g. myprog_proto_1609536398, myprog_orc_1609536398). Specifying any positive value for this flag will result in that value being used to create directories (e.g. timestamp=456 results in myprog_proto_456, myprog_orc_456), or if a directory with that timestamp already exists, it will be overwritten.

-toph-args: (string) This flag specifies a list of space-separated flags to be set when Vtrace is using Toph to translate models. For example, to make Toph generate queries for only channel safety and exiting goroutines with panic, set -toph-args="-query-channel-safety=true -query-goroutine-exit-with-panic=true". For information about which can be passed into Toph, run toph.go and the flags will be printed to the console. The default value for this flag is an empty string "", which means that Toph will generate as many queries as possible given the components of the model.

-optimize-first: (bool) This flag causes Vtrace to instruct Toph to optimize the Source model, leading to faster times for the first pass stage, reduced states in the Source model, and therefore reduced Guards in the Proto model, leading to faster verification times in the second pass and deep pass stages. The default value is true.

-run-depth: (int) This flag should be set to the maximum number of iterations that Vtrace should run for the deep pass stage. The user is free to set this as high as they wish (for example, if they wish to exhaustively verify that false errors are indeed false errors). The default value is 15.

-ignore-toph-errors: (bool) This flag should be set to true if Vtrace is allowed to ignore any errors that Toph produces while translating. Toph may produce errors when encountering a line of code in a program it cannot translate fully, but in spite of this, it can still produce a model which contains real errors. By default, this flag is set to false and Vtrace will exit if Toph produces any translation errors. If this flag is set to true Vtrace will still print out any errors that Toph encounters, but it will continue rather than exiting.

-timing: (bool) This flag should be set to true if Vtrace is to report the timing that each stage (first pass, second pass, and each iteration of the deep pass) of the algorithm takes. The execution times for these stages will be printed to the console after each stage is complete. The default value of this flag is true.

5. Evaluation

5.1 Uppaal memory limitation

One common issue that was consistent across all attempts to evaluate Vtrace's performance was the memory limitation of Uppaal. Uppaal is a 32-bit process, and as such it cannot allocate more than 4GB of memory on a 64-bit system, or more than 2GB of memory on a 32-bit system[11]. For Source models this is not an issue, and models for large programs (such as boltdb[12]) can be verified. However, this memory limit becomes much more of an obstacle for the performance of the Proto model in the second pass (without trace disallowing), and absolutely devastating for the performance of the Proto model when using trace disallowing in the deep pass.

For the Proto model without trace disallowing, Toph has to add one additional instance of a channel process for every Guard in Proto. This means that the Proto model becomes much larger than the Source model, with a massively increased state space. This state space becomes even larger however, once trace disallowing is added to the Proto model. It is believed that this is due to the fact that with trace disallowing, every state in the Proto model becomes distinct (since the hash32 variable is constantly updated). This drastically increases the memory required for verification, since parts of the model which contain loops (such as the Scheduler function) are essentially unrolled, as each iteration of the loop would contain a global variable with a different value for each state in the loop. In general, for every state in a model without trace disallowing, there are multiple copies of that state in the model with trace disallowing. Each one of these copies has a unique hash32 value encoding the entire path to that state, and these copies pollute the overall state space.

5.2 Stage scaling

Since the time for the first pass, second pass and deep pass stages are all highly dependent on the original program supplied to Vtrace, it is difficult to quantify exactly how the performance for these stages scales as the program grows more complex. To perform some meaningful evaluation, a collection of test programs were written for Vtrace with each program designed to investigate the performance scaling of each stage in isolation.

For this evaluation, the following specification was used:

CPU	Intel i5-9600KF CPU @ 3.70GHz
Memory	2x8GB DDR4 @ 3000 MHz
Operating System	Windows 10.0.19041
Uppaal version[13]	4.1.24
Go version[14]	go1.14.6 windows/amd64
Java version	11.0.9 (major version 55)

Table 5.1: Hardware and software used for evaluation

No other programs were running in the background while testing, and every test of program runtime was repeated 5 times, with the final results being taken as an average of all of these runs. Each of the test programs was written to have a only one real error, and the time

[11] Uppaal's memory limitation https://bugsy.grid.aau.dk/bugzilla/show_bug.cgi?id=63 (last accessed 2021-05-10)
[12] https://github.com/boltdb/bolt
[13] Uppaal (version 4.1.24), Available at: https://uppaal.org/ (last accessed 2021-02-08)
[14] Go, Available at: https://golang.org/ (last accessed 2021-02-08)

measured for the deep pass stage is the sum of the time spent across all iterations that were needed until the real error was reported. Testing for any particular program was run until a clear trend in the growth of the program run time was observed and/or until Uppaal failed to verify queries for the Proto model of the test program.

Testing max states - sequences

The first program written is called sequences. (see Appendix A). In this program, a number of sending and receiving operations are performed between two goroutines, after which the program panics, producing a real error. The purpose of sequences is to investigate the relationship between the number of states in the Source model and the time taken for each of the stages. The number of states is controlled by increasing the number of channel operations in the program (since each channel operation will be a state in the Source model). Sequences has a real error, and this real error will always be found after one iteration of the deep pass.

For this test, the following graphs were produced showing how the run time scales as more states are added. In these graphs, N is the number of channel operations in each of the two goroutines, and T(N) is the time taken to run the stage to completion having N channel operations in each goroutine. In these graphs, the Log (base 2) of each of both N and T(N) are used, so that the graph is showing the growth of the run time, rather than the run time itself.

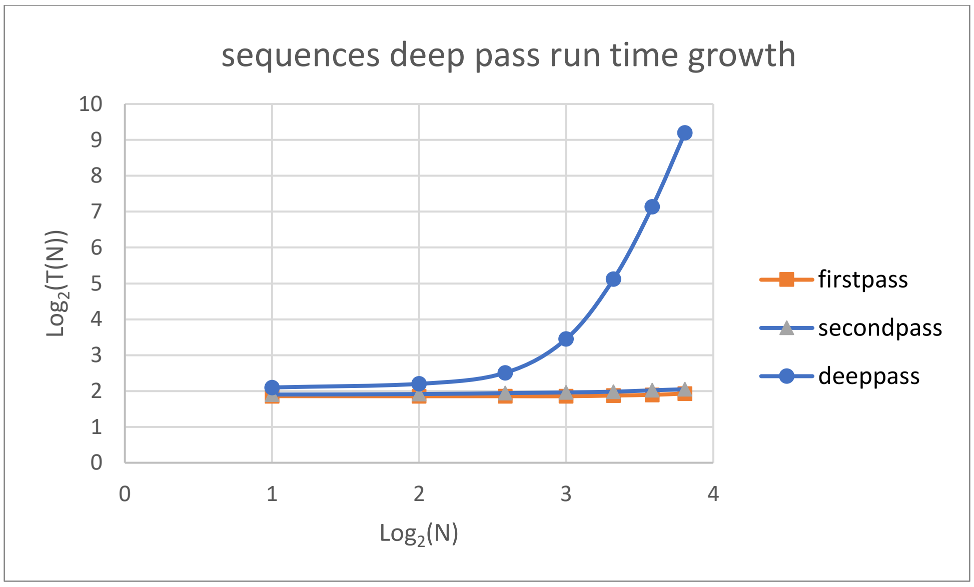


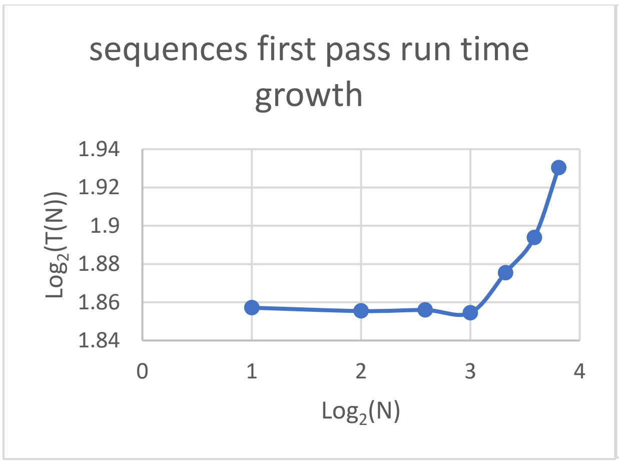
Figure 5.1: Run time growth of stages for sequences program

From this graph, it is clear to see the run time for the deep pass grows drastically as more states are added to the model, with Uppaal going out of memory in the deep pass when attempting to verify a query for a Proto model which has 35 states in the Source model.

This is clearly a problem, as 35 states is a very low number to have in a Go program, and certainly this indicates that Vtrace in its current form is not viable for large codebases. When fitting a curve to this data, it is found that the following equation approximates the curve of the run time growth for the deep pass for the ‘sequences’ program:

$$Log_2(T(N)) = 0.001349(Log_2(N))^{6.427} + 2.025$$

Most notably, the run time for the deep pass appears to grow at an exponential rate and grows much faster than the run time for the first or second pass stages. It is worth noting however that both the first pass and second pass stages are also growing exponentially, as seen in the below graphs (figures 5.2 and 5.3). Since the deep pass will fail before either of these stages become so time consuming as to matter, their growth rate is of less importance for this program.



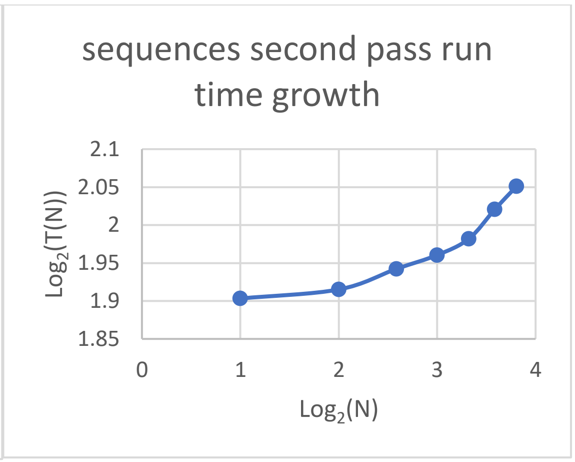


Figure 5.2: Run time growth of first pass stage for sequences program

Figure 5.3: Run time growth of second pass stage for sequences program

Testing loop verification - mutexloopconst

The next program used for testing was called mutexloopconst (see Appendix B). In this program a loop runs, locking a mutex on every even-numbered iteration and unlocking it on every odd-numbered iteration. After the loop exits, the mutex is locked again. If the loop has run for an even number of times the program will panic, as a mutex cannot be re-locked while a lock for it is already held. This test investigates the effect that loops have on a program, as loops are a way to increase the effective size of a program without adding more states to it.

Importantly in this program, the number of times that the loop runs is known at compile time, and therefore is known to Toph when translating and can be built into the model. The resulting graph shows the run time growth for all first pass, second pass and deep pass stages, where N is the number of iterations performed by the loop:

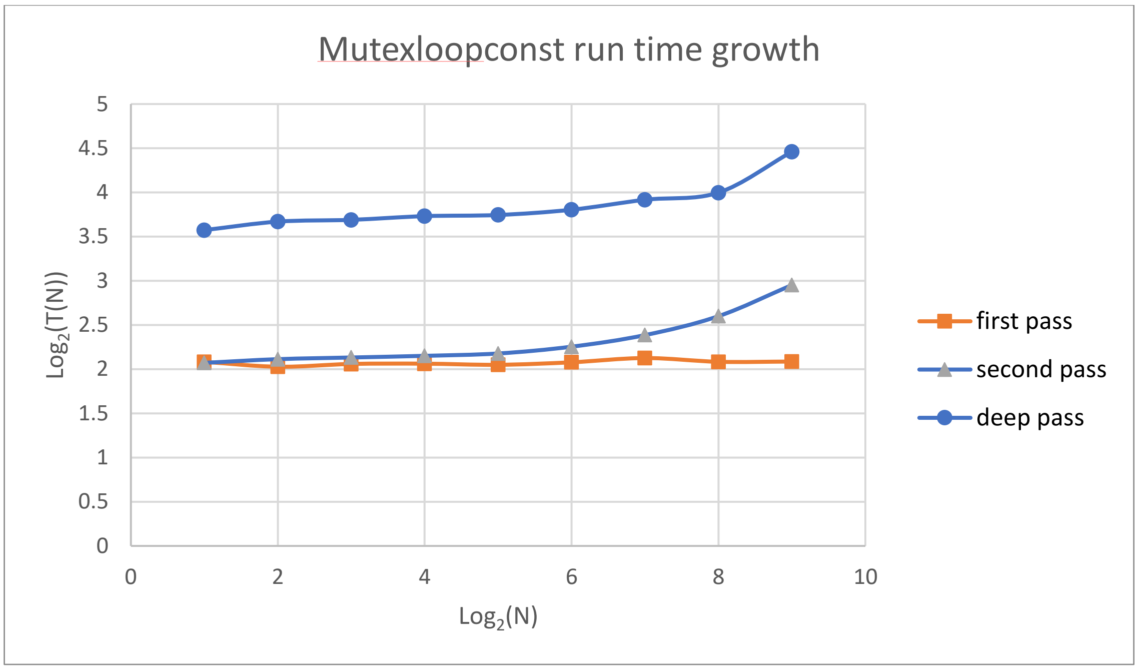


Figure 5.4: Run time growth of stages for mutexloopconst program

The run time for the first pass appears to be constant, with the second pass and deep pass showing very slight exponential growth, with the second pass run time growth modelled by:

$$Log_2\big(T(N)\big) = \ 5.757x10^{-5}\ (Log_2(N))^{4.363} + \ 2.109$$

And the deep pass run time growth also modelled exponentially as:

$$Log_2\big(T(N)\big) = \ 1.041x10^{-5}\ (Log_2(N))^{5.093} + \ 3.673$$

Overall, mutexloopconst is a good demonstration of a program which scales well for Vtrace, performing well up to and including the case where N=512 loop iterations, before Uppaal's server crashes when attempting to verify N=1024 iterations.

Testing loop verification - mutexloopvar

Often in Go programs, a loop's iterations are not known at compile time, but rather are determined by some variable which is set as the code executes. The mutexloopvar program (see Appendix C) is a variation of mutexloopconst in which the limit of loop iterations is set as a variable, not set constant at compile time. In this case, Toph cannot determine how many times a loop must run, and neither can Uppaal. Multiple iterations of the deep pass are needed before Uppaal produces a trace which contains the correct number of iterations to exit the loop and verify that a real error exists. The run time growth for each stage is shown below, with N being the value of the variable that is set to the maximum number of iterations for the loop:

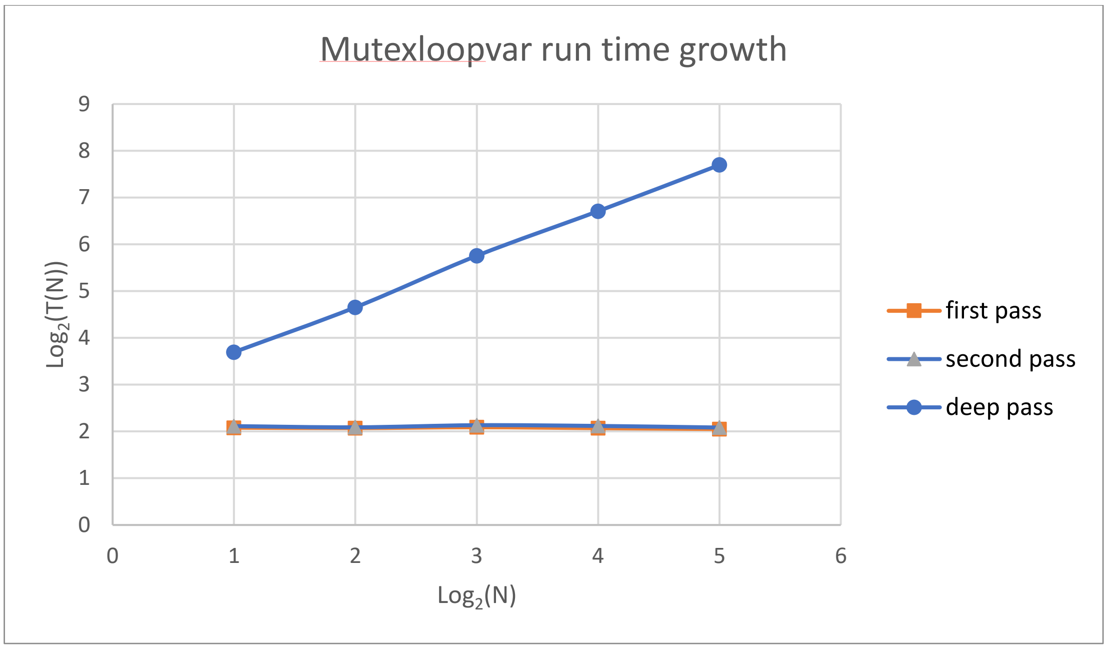


Figure 5.5: run time growth of stages for mutexloopvar program

Here, the run time for the first pass and second pass appear to not grow (although from the mutexloopconst test, it is likely that they exhibit a very slight exponential growth unnoticeable for the lower value of N here). Meanwhile, the deep pass grows linearly, and its growth can be modelled with the equation:

$$Log_2\left(T(N)\right) = 1.006Log_2(N) + \ 2.684$$

Looking at the program, the cause of this linearity can be understood clearly. For any value of N, Uppaal will first produce a trace which contains only one iteration of the loop. If N = 1, then the real error is found from this trace. However, if N > 1, then the Vtrace test runner will report a false error, since it has run an Orchestration test based on a trace which expects the loop to run once before continuing to the next state after the loop – something that is not possible to reproduce. This trace will therefore be disallowed, and the next trace produced by Uppaal will contain two iterations of the loop. This will continue for N cycles, until Uppaal produces a trace with N iterations of the loop.

Testing goroutine spawning - Loopspawn

The last performance test was to investigate the growth of the run time as more goroutines were spawned in a program. This test uses the loopspawn program (see Appendix D). In this program, a loop runs for a constant number of iterations (the number of iterations is fixed at compile time). Each iteration of this loop spawns a new goroutine to run a very simple function called checkPanic which will cause a panic if the goroutine is the last goroutine that is spawned by the loop. This panic is a real error and is detected on the first iteration of the deep pass.

The following graph in figure 5.6 shows the run time growth of the loopspawn program as the number of spawned goroutines (N) increases.

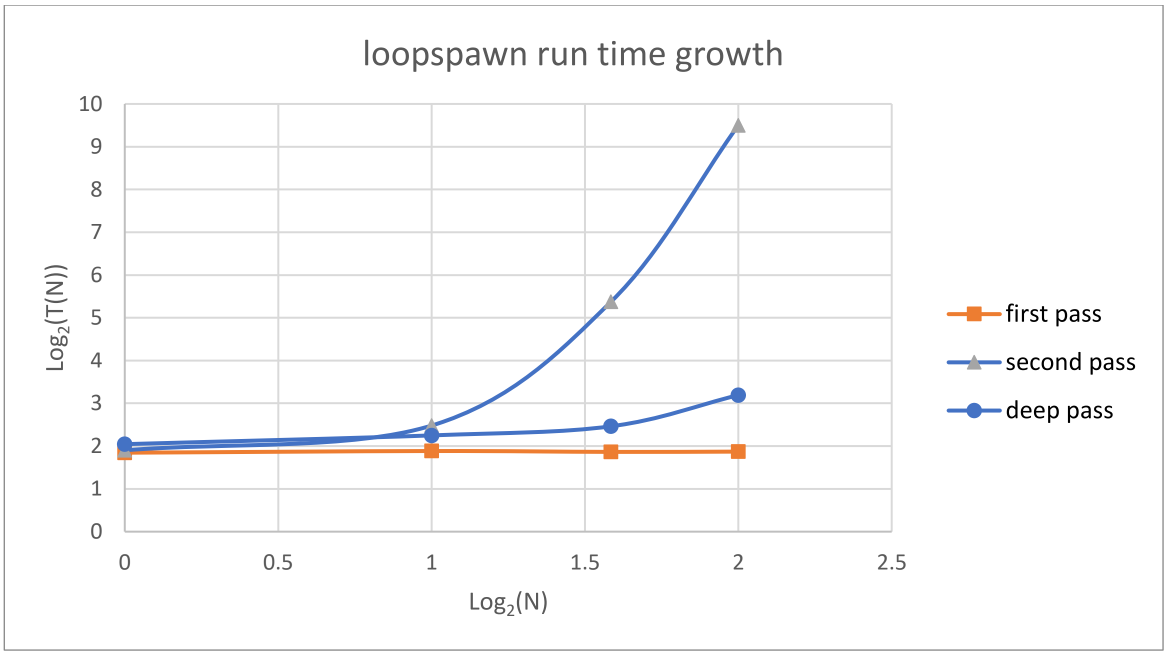


Figure 5.6: run time growth of stages for loopspawn program

Unlike other graphs, the second pass stage dominates here, growing at a rate which causes Uppaal to run out of memory when trying to complete the second pass stage in a program with only 5 goroutines spawned. The second pass can be modelled by the exponential equation:

$$Log_2\big(T(N)\big) = 0.721Log_2(N)^{3.411} + \ 1.85$$

And the deep pass can also be modelled exponentially by:

$$Log_2\big(T(N)\big) = 0.05323Log_2(N)^{4.338} + \ 2.107$$

It may seem surprising that the second pass here has such bad performance relative to the deep pass, given all other tests have the deep pass as the performance bottleneck stage. However, when looking at the queries run in the second pass stage, the run time growth of the second pass stage becomes a lot more obvious. In the second pass, lot of queries for this Proto model are generated which fall under the category of checking that a goroutine does not exit with a panic, and are not removed since this category is of the same type as the queries that fail. Verifying these queries takes a long time, even in this scenario where all goroutines have so few Guards.

The poor performance and abysmal scaling of loopspawn has made it a prime target for measuring the improvement of optimisations made to Vtrace's algorithm and implementation, specifically those aimed at query reduction.

5.3 Query reduction optimisation

For testing the performance of query optimisation, the loopspawn program with 4 concurrent goroutines as a metric (for reference, this version of loopspawn will be known as loopspawn-4). Before the introduction of the first optimisation (the addition of the second pass stage to find only failing queries), Vtrace would proceed from the first pass into the deep pass, and all queries that Toph generated for the Proto model would be checked with trace disallowing.

This was highly inefficient, and it was not possible for the deep pass stage to complete without Uppaal running out of memory.

Second pass

With the introduction of the second pass stage to reduce the queries that run in the deep pass, loopspawn-4 was able to progress past the second pass stage, taking 5783 seconds (1 hour 36 minutes and 23 seconds) to complete the second pass stage, checking 36 queries. After this second pass had completed however, Uppaal still ran out of memory in the deep pass stage when checking the following query which verifies that the Scheduler function cannot deadlock:

$$A[]\ (not\ out\_of\_resources)\ imply\ (not\ (deadlock\ and\ func4\_Scheduler\_0.select\_pass\_2\_0))$$

Note that this query is related to the Scheduler function and can be safely removed with the next optimisation as being a result of code added by Vtrace, it will never cause a real error.

Removing Scheduler/Guard queries

Next, Toph was modified to prevent queries being generated for Guard channels or the Scheduler function, and as a result loopspawn-4 was able to complete the second pass stage in 1408 seconds (23 minutes and 28 seconds), a reduction of 75.6% compared to the original time. In this second pass stage, only 14 queries were checked, representing a reduction of 61.1% compared to the original amount.

After the second pass stage had completed, the deep pass was now able to complete without going out of memory, taking only 10 seconds to find the error (only one iteration of the deep pass).

Category-based removal

After queries were removed based on their category and the category of queries that failed in the first pass, loopspawn-4 was able to complete the second pass stage in 706 seconds (11 minutes and 46 seconds), a reduction of 87.79% compared to the original time. In the second pass stage now, only 9 queries were checked, a reduction of 75% compared to the original amount. Again, the deep pass stage was able to complete without going out of memory and took only 9 seconds (only one iteration of the deep pass).

4.7 Self-parsing test

To test the capabilities of the parser (i.e. how well the parser is able to concatenate lines and place Guards correctly), a short test was written in which Vtrace was instructed to parse its own code (3365 lines in 19 files, not including Go tests) with the flag reduced=false, meaning that every single line was to be Guarded. If no Guards were placed incorrectly (i.e. they were not in the middle of lines or within struct definitions etc), the Proto version should compile without errors. This is achieved successfully, with Proto and Orc versions of Vtrace being generated in an average of 32.80 milliseconds, and with this Proto version compiling without issue.

6 Implementation limitations and future work

Vtrace's current implementation, while extensive, has a number of limitations that may be resolved in future work. This section will cover some limitations, and also detail future improvements that would benefit Vtrace.

6.1 Incompatibility with cgo

Currently, Vtrace relies on the deadlock detection system built into the Go runtime. This Go runtime uses a group of OS-level threads (known as Machines or 'M's), onto which a scheduler multiplexes all goroutines in the program. In the event that all Ms are blocked, then the runtime can report that the program is deadlock. This approach to deadlock detection becomes an issue when cgo is included in a program, as its presence results in an extra OS-thread being spawned[15]. The existence of this extra M means that the Go runtime will no longer be able to report a deadlock from purely goroutine-based interaction and thus, Vtrace can no longer use this deadlocking as a check for a false error.

One major case where this limitation is obvious is in programs which use 'net' package in the standard library to do networking operations. One some systems (unix systems for example), the net package uses a Go-based resolver for domain lookup, with the option to use a cgo resolver available but turned off by default. On other systems (such as Windows), the net package does not have this option, and can only use the cgo resolver. As a result, Go programs that use the net package for network operations on Windows systems will not run in Vtrace.

Currently, Vtrace does not detect programs which use cgo, or any packages that import cgo. In future, this detection could be added so that users get warned if their code is incompatible with Vtrace. In future work perhaps Vtrace could allow simple create/read/update/delete operations to be performed by replacing calls to the net package with custom functions that Vtrace would export. Each of these custom functions could create a new OS process which executes the net request, writes the result to a file and ends. The custom function would then read the file, deletes it and returns the result to the calling function. This would mean that connections would not be able to be maintained in the program and reused, but that basic network operations would be possible.

This would allow for logging to be written to the Vtrace test runner in real time over network connections. Perhaps the Vtrace test runner could host a server with a webpage GUI that displays the progress through the program by each goroutine in real time. If Vtrace supports large programs in future, or if Orchestration tests begin to take a long time to run, this could be useful and really elevate the user experience.

6.2 Reading/writing external memory

Currently, all Orchestration tests run under the assumption that the program does not interact with external memory or does not have any elements of randomness. If it is the case that some external memory is read and then written or overwritten at any stage during the program, then this reading and overwriting will occur when every Orchestration test is running ,and these Orchestration tests may not be able to find errors as they do not all start with the same initial conditions (i.e. modified memory is preserved during testing). To

[15] *Runtime file defining cgo-related variables,* https://golang.org/src/runtime/cgo.go (last accessed 2021-05-10)

prevent against this, Source programs written for Vtrace should not overwrite any memory that they ever have read from or will read from.

To avoid modifying or intercepting any system calls (which would require customising a Go runtime, thus tying Vtrace to a particular Go version), future work could be done into checking which files are modified by the Source program, and making backups of these files so that they can be restored before any Orchestration test is run.

6.3 Shared memory

The use of shared memory in a Source program can be challenging for Vtrace to deal with and can in some situations make Orchestration tests nondeterministic. Due to the abstraction performed by Toph certain types of variables cannot be modelled, and so are effectively invisible to Uppaal when verifying. While this is fine in the case of locally-scoped variables, global variables or variables that are shared between goroutines create complications for running Orchestration tests.

Firstly, In the case that some memory is shared between multiple goroutines and an error exists in one of these goroutines which depends on the value of that shared memory, Uppaal cannot ever produce a trace where it knows exactly how to produce that error. As a result, programs with errors of this form will only be found through iterations of the deep pass, as one of these iterations will eventually produce a trace in which Uppaal randomly finds the correct sequence of Guard unblocks that lead to the error. Depending on the number of states and goroutines, this kind of random search may take a long time and a lot of iterations of the deep pass.

As an example of this, consider the following program (6.1):

```
package main

import "sync"

var x int

func panicMaybe() {
  if x == 0 {
    panic("can catch")
  }
}

func main() {
  x = 1
  go panicMaybe()
  x++
  var m sync.Mutex // create a new state here
  m.Lock()
  if x == 2 {
    x = 0
  }
}
```

(6.1)

In this program, a real error exists since it is possible that the main function gets to reset the value of the global variable x to zero before the panicMaybe goroutine evaluates the if statement. Note that in this program, a mutex is added after the goroutine is spawned to create a state in the Source model (and a Guard in Proto). This Guard would never be unblocked on the first iteration of the deep pass (since Uppaal sees nothing beyond this state that contributes to the error) and will only be unblocked during some later iteration of the deep pass. As a result, this program takes multiple deep pass iterations before the error is found. The following logs are collected from these passes, showing the various Orchestration tests that get run in an attempt to find the error.

```
2021/05/08 02:44:32.802242 func6_main_0  :  func main() {
2021/05/08 02:44:32.803274 func6_main_0  :  x = 1
2021/05/08 02:44:32.803274 func6_main_0  :  go panicMaybe()
2021/05/08 02:44:32.803274 func6_main_0  :  x++
2021/05/08 02:44:32.803274 func6_main_0  :  var m sync.Mutex // create a new state here
2021/05/08 02:44:32.804260 func5_panicMaybe_0  :  func panicMaybe() {
2021/05/08 02:44:32.804260 func5_panicMaybe_0  :  }
```

Deep pass: first iteration - trace attempts to go into if statement immediately

```
2021/05/08 02:44:36.256209 func6_main_0  :  func main() {
2021/05/08 02:44:36.257225 func6_main_0  :  x = 1
2021/05/08 02:44:36.257225 func6_main_0  :  go panicMaybe()
2021/05/08 02:44:36.257225 func6_main_0  :  x++
2021/05/08 02:44:36.257225 func6_main_0  :  var m sync.Mutex // create a new state here
2021/05/08 02:44:36.259206 func5_panicMaybe_0  :  func panicMaybe() {
2021/05/08 02:44:36.259206 func5_panicMaybe_0  :  }
```

Deep pass: second iteration - different trace, producing same output as first iteration

```
2021/05/08 02:44:39.817906 func6_main_0  :  func main() {
2021/05/08 02:44:39.818878 func6_main_0  :  x = 1
2021/05/08 02:44:39.819926 func6_main_0  :  go panicMaybe()
2021/05/08 02:44:39.819926 func6_main_0  :  x++
2021/05/08 02:44:39.819926 func6_main_0  :  var m sync.Mutex // create a new state here
2021/05/08 02:44:39.820878 func5_panicMaybe_0  :  func panicMaybe() {
2021/05/08 02:44:39.820878 func5_panicMaybe_0  :  }
2021/05/08 02:44:39.821878 func6_main_0  :  m.Lock()
2021/05/08 02:44:39.821878 func6_main_0  :  if x == 2 {
2021/05/08 02:44:39.821878 func6_main_0  :  x = 0
2021/05/08 02:44:39.821878 func6_main_0  :  }
```

Deep pass: third iteration - progressing past the mutex state, wrong order

```
2021/05/08 02:44:43.753878 func6_main_0  :  func main() {
2021/05/08 02:44:43.753878 func6_main_0  :  x = 1
2021/05/08 02:44:43.754878 func6_main_0  :  go panicMaybe()
2021/05/08 02:44:43.754878 func6_main_0  :  x++
2021/05/08 02:44:43.754878 func6_main_0  :  var m sync.Mutex // create a new state here
2021/05/08 02:44:43.755877 func6_main_0  :  m.Lock()
2021/05/08 02:44:43.755877 func6_main_0  :  if x == 2 {
2021/05/08 02:44:43.755877 func6_main_0  :  x = 0
2021/05/08 02:44:43.755877 func6_main_0  :  }
2021/05/08 02:44:43.755877 func5_panicMaybe_0  :  func panicMaybe() {
2021/05/08 02:44:43.756877 func5_panicMaybe_0  :  if x == 0 {
2021/05/08 02:44:43.756877 func5_panicMaybe_0  :  panic("can catch")
```

Deep pass: fourth iteration - error found

The second case where shared memory poses an issue for Vtrace is in the case where the shared memory is read or written along a transition (which is executed deterministically according to an Orchestration test), in such a way that an interleaving which would be possible is no longer possible.

For example, consider the above program (6.1), but placing the ‘x++’ line along the transition before the entrance into the if statement in panicMaybe, as shown in (6.2).

```
package main

import "sync"

var x int

func panicMaybe() {
  x++ // code along transition, error is uncatchable
  if x == 0 {
    panic("cannot catch")
  }
}

func main() {
  x = 1
  go panicMaybe()
  var m sync.Mutex // create a new state here
  m.Lock()
  if x == 2 {
    x = 0
  }
}
```

(6.2)

In this case, no amount of iterations of the deep pass will result in the error being caught, as the x++ lies along a transition, and its execution will always be followed by the if statement. This is more obvious when seeing this function in Proto. In the below snippet of code (6.3), an Orchestration test will deterministically take execution directly from ChPROTO_main_0 to ChPROTO_main_1, or from ChPROTO_main_0 to ChPROTO_main_2, and x will never receive any interleaving with the main goroutine that would set x=0 after the x++ line was executed.

```
func panicMaybe() {
  <-PROTO_scheduler_autogen.ChPROTO_main_0
  x++ // code along transition, error is uncatchable
  if x == 0 {
    <-PROTO_scheduler_autogen.ChPROTO_main_1
    panic("cannot catch")
  }
  <-PROTO_scheduler_autogen.ChPROTO_main_2
}
```

(6.3)

The last case where shared memory becomes an issue for Vtrace to deal with is when reads or writes to shared memory that could result in an error are directly underneath an operation which requires synchronisation with another goroutine. In this case, the operation will require a call to orcutil.UnblockOrchestrate to progress the Orchestrate function, and so the lines below this synchronised operation can no longer be executed deterministically (i.e. the Orchestrate function cannot wait for the next state to be reached here). Therefore, in the case that the lines beneath the synchronisation operation contain some read/writes to shared memory, some interleaving which would otherwise cause an error may not be occur, and an Orchestration test could incorrectly be reported as corresponding to a false error.

For example, consider the following version (6.4) of the above programs ((6.1) and (6.1)):

```
package main

var x int

func panicMaybe(ch chan int) {
  <-ch
  if x == 0 {
    panic("maybe catch")
  }
}

func main() {
  x = 1
  ch := make(chan int)
  go panicMaybe(ch)
  ch <- 0
  /* to demonstrate nondeterminism, import time and uncomment this: */
  // time.Sleep(1*time.Second) // sleep to ensure goroutine progresses first
  x++
  if x == 2 {
    x = 0
  }
}
```

(6.4)

In here the same error exists as before (although the additional state added by the mutex is no longer present). However, as the below code snippets (6.6) and (6.7) show, the Orchestration test will contain calls to unblock Orchestrate function, releasing the calling goroutine to complete its transition to the next state. Here consider the situation where the maybePanic() goroutine evaluates its if statement and exits before the x++ line is executed, and a real error is not found.

```
<-PROTO_scheduler_autogen.ChPROTO_main_5
orcutil.PrintControlFlow(ChORCHESTRATION, "ch <- 0")
go orcutil.UnblockOrchestrate(ChORCHESTRATION)
ch <- 0
/* to demonstrate nondeterminism, import time and uncomment this: */
// time.Sleep(1*time.Second) // sleep to ensure goroutine progresses first
x++
orcutil.PrintControlFlow(ChORCHESTRATION, "x++")
if x == 2 {
  orcutil.PrintControlFlow(ChORCHESTRATION, "if x == 2 {")
  x = 0
  orcutil.PrintControlFlow(ChORCHESTRATION, "x = 0")
}
orcutil.PrintControlFlow(ChORCHESTRATION, "}")
<-ChORCHESTRATION
```

(6.6)

```
<-PROTO_scheduler_autogen.ChPROTO_main_0
orcutil.PrintControlFlow(ChORCHESTRATION, "func panicMaybe(ch chan int) {")
orcutil.PrintControlFlow(ChORCHESTRATION, "<-ch")
go orcutil.UnblockOrchestrate(ChORCHESTRATION)
<-ch
if x == 0 {
```

(6.7)

The one solution to these last two issues is the use of the ‘reduced’ flag for Vtrace. By passing the argument reduced=false, Vtrace will place Guards over every single line when building Proto. This will make the state space for the Proto model a lot larger, increase memory usage in the deep pass and increase the time taken for Uppaal to check queries, but it will ensure that the problems described above are managed. With Guards over every line, enough deep pass iterations will eventually cause the real error to be found. Future work could be done into improving Vtrace's handling for shared memory. Global variables could possibly be identified while parsing (as they will always be referenced by the same name), and could have Guards placed over lines where they are read or modified, meaning that the deep pass would work better with these variables without the need for reduce being set false. Additionally, some better way of controlling the Orchestrate function without needing to call unblockOrchestrate would prevent the nondeterminism problem described above.

6.4 Hashing in Uppaal

Since the FNV-1a function can have collisions, it cannot be guaranteed that two different traces would have the same hash and cause a real error to be never found. Specifically, if the hash of a trace which would produce a real error in an Orchestration test happened to be the same as the hash of a trace from a previous iteration of the deep pass, then the trace for the real error would never be reported.

For example, if a trace has a hash of 736285766 is built into an Orchestration test that is run and found to correspond to a false error, then trace disallowing will ensure that no other trace with that hash is generated. However, if a different path to the error had a hash which collided with that earlier trace, then this path to the error would also never be returned as a trace. If that error path could have been viable in Orc, then the trace which would prove that a real error exists has been accidentally ignored.

Since every hash function has collisions, any future work which wished to solve this issue would have to be based off of a fundamentally new approach to trace disallowing – one which did not involve hashing. While FNV-1a hash does have a low collision rate, a hash function with an even lower collision rate would help reduce the chance of collisions (see table I in [12], p.687). However, any other hash function would be more expensive to compute, and the decreased likelihood of collisions may not be worth the increase in time spent checking queries in the deep pass.

6.5 Function objects and arguments

Since Vtrace adds the ChORCHESTRATION argument to all function definitions, and passes this variable into all function calls, it cannot handle cases where the function itself is passed into another function (at which point its name may become modified within that called function). Likewise, any function objects that are passed into other functions will now no longer match the type specified by the argument.

For example, consider the following program (6.8), in which a function is passed into another function:

```
package main

func A(toCall func(i int), value int) {
  toCall(value)
}

func B(v int) {
  panic(v)
}

func main() {
  A(B, 2)
}
```

(6.8)

In Orc, the function definitions here for A and B will become (6.9) and (6.10).

```
func A(ChORCHESTRATION orcutil.OrcChan, toCall func(i int), value int) {
```
(6.9)

```
func B(ChORCHESTRATION orcutil.OrcChan, v int) {
```
(6.10)

This however poses an issue in two lines in this program. Firstly, the line where A is called in main has still passes in the function B, but now the function B is no longer of the type `func(i int)` and cannot be passed into this function anymore. Similarly, in the line where toCall is called will still be written as (6.11)

```
toCall(value)
```
(6.11)

Since no function named toCall was defined in this program, and the parser was not aware that the ChORCHESTRATION argument should have been added here.

The result of these issues is that no Orchestration test will run, as Orc contains Go code which causes the compiler to produce an error. To help notify the user if this issue is present in their program, Vtrace will warn the user as Orc is being generated in the event that a function is found passed into another function. For example, when Proto and Orc were being generated for this example program above, the following warning line was printed to the standard output:

```
WARNING: in file
D:\Users\Brian\go\src\github.com\brianneville\vtrace\examples\demo\funcobj_orc_123\main.go, the
variable "B" may be a function in line: A(ChORCHESTRATION,B, 2).
                If it is a function,please modify this any calls that are made to the variable, and modify the
variable's type if necessary.
```

If the user wishes to keep this function-passed-into-function code in their program, they can amend the issue in Orc by modifying the function calls and definitions. After this, the user can then restart Vtrace from the second pass stage using their corrected version of Orc by running Vtrace with the proto-pkg flag set to the path to the Proto directory.

Future work on Vtrace could add a map of argument names that correspond to function objects into the *flags* *buildFlags variable that is used in writeLineWithinFunction. Then any time a function call is made to one of the functions that is saved in this map, that function call could pass the ChORCHESTRATION variable too. Also, the function definition line could have any parameters which are of some function type, have their type changed to include the ChORCHESTRATION argument.

6.6 Deep pass optimisation

If only one iteration of the deep pass is needed for a query to produce a trace that leads to an error when built into an Orchestration test, then the inclusion of trace-disallowing has only served to make performance worse when checking this query. Therefore, the deep pass could be optimised in future work by only adding trace-disallowing after the first iteration. This would mean that the first iteration and the second iteration of the deep pass would lead to the exact same traces, but given the additional time that checking queries takes with trace disallowing, it would be worthwhile performing this first iteration without trace disallowing.

6.7 Pattern-based query reduction

One future way to further reduce the queries in the second pass stage would be to do pattern matching on these queries against the queries that failed in the first pass. This could be done by constructing a regex for each of the queries that failed in the first pass and using this regex to further filter the queries in the second pass. For example in the case that a goroutine exits with a panic, the query that failed for this error in the first pass could be used to build a regex that only allows queries which have that structure to be used for the second pass. If the query formula that failed in the first pass was:

A[] (not out_of_resources) imply (not (func4_panickyFunc_0.ending and !func4_panickyFunc_0.is_sync and func4_panickyFunc_0.internal_panic))

Then this query formula could be turned into a regex which matches

*A[] (not out_of_resources) imply (not (func***X***_panickyFunc_***Y***.ending and !func***X***_panickyFunc_***Y***.is_sync and func***X***_panickyFunc_***Y***.internal_panic))*

Where **X** and **Y** are any sequence of digits in the range 0-9.

If this pattern-based matching was good enough, it could mean that query-filtering aspect of the second pass stage is no longer required, and the filtered set of queries for the Proto model could be run for the first time in the deep pass stage.

7 Conclusion

Revisiting the original research question of *"Given a trace through an Uppaal model of a Go program, is it possible to instrument the Go program such that the trace can be determined to be reproducible?"*, the work presented in this dissertation has shown that this is very much possible, and Vtrace is capable of performing this trace extraction, instrumentation, and determination automatically.

However, Vtrace is still not perfect and in its current implementation, it most notably faces challenges with applied to large-scale programs. In small part this is due to the memory limitations of Uppaal, but primarily this poor scaling can be seen as a consequence of the way that Vtrace manufactures determinism within programs (through channel operations which pollute the state space in Uppaal models).

This project has brought a lot of challenges, both in terms of algorithm design and implementation. Vtrace was built entirely from the ground up without any other real reference tool and took a lot of planning throughout. Figuring out how to write a tool which could handle any Go program in a general sense was also difficult, especially since Vtrace operates on the code itself (syntactically), and not on some partially compiled format of the program (such as an AST). As more challenges were encountered during the implementation, and more code was added by Vtrace into Proto and Orc for orchestration purposes, the necessity of a human readable output became clearer. Vtrace's line logging allows for a much more intuitive understanding of how an error manifests, and overall makes Vtrace a lot more user-friendly.

Vtrace was designed to first be able to handle the most basic Go programs, then iteratively built to be more performant and capable of handling more complex programs. However, if redesigning Vtrace from the beginning, it could be beneficial to make scalability an absolute priority and design the entire tool around handling large-scale programs. With this approach, it is likely that the resulting tool was would be much more useful, since errors become increasingly obscure in larger programs.

While there is still a lot of future work that could be done and improvements that could be added, Vtrace in its current form still offers great utility to Toph and presents a novel approach to trace reconstruction in Go programs.

References

[1] S. Prata (2004) *C primer plus, 5th edition*, Pearson Education, 113–114

[2] Clarke, E., Grumberg, O., Jha, S., Lu, Y., & Veith, H. (2001). *Progress on the State Explosion Problem in Model Checking.* Informatics, 176–194. doi:10.1007/3-540-44577-3_12

[3] Kai Stadtmüller, Martin Sulzmann, and Peter Thiemann (2016) *Static Trace-Based Deadlock Analysis for Synchronous Mini-Go*. Asian Symposium on Programming Languages and Systems (APLAS).

[4] Arne Philipiet (2020) *Verification of concurrent Go programs using Uppaal,* School of Computer Science and Statistics, Trinity College Dublin

[5] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. (2004) *Flashback: A lightweight extension for rollback and deterministic replay for software debugging*, Proceedings of the USENIX Annual Technical Conference (USENIX '04).

[6] Y. Saito. (2005) *Jockey: A User-space Library for Record-replay Debugging*. Proceedings of the International Symposium on Automated Analysis-driven Debugging (AADEBUG).

[7] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, I. Stoica. (2007) *Friday: Global Comprehension for Distributed Replay*, Proceedings of the fourth Symposium on Networked Systems Design and Implementation (NSDI'07).

[8] D. Geels, G. Altekar, S. Shenker, and I. Stoica. (2006) *Replay Debugging for Distributed Applications.* USENIX Annual Technical Conference.

[9] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. (2008) *D3s: Debugging deployed distributed systems*. Proceedings of the Fifth Symposium on Networked Systems Design Implementation (NSDI'08).

[10] Y. Zhang, S. Makarov, X. Ren, D. Lion, D.Yuan (2017) *Pensieve: Non-Intrusive Failure Reproduction for Distributed Systems Using the Event Chaining Approach.* Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17).

[11] Vaastav Anand (2020) *Dara The Explorer: Coverage Based Exploration for Model Checking of Distributed Systems in Go*, The Faculty Of Graduate And Postdoctoral Studies, The University of British Columbia.

[12] César Estébanez, Yago Saez, Gustavo Recio and Pedro Isasi. (2014). *Performance of the most common non-cryptographic hash functions*. Software: Practice and Experience 2014 44:681–698

Appendices

A. Sequences

Shown here with N (number of channel operations per goroutine) = 14

```go
package main

var ch = make(chan int)

func sender() {
  ch <- 1
  ch <- 1
  // 2 ^
  ch <- 1
  ch <- 1
  // 4 ^
  ch <- 1
  ch <- 1
  // 6^
  ch <- 1
  ch <- 1
  // 8^
  ch <- 1
  ch <- 1
  // 10 ^
  ch <- 1
  ch <- 1
  // 12 ^
  ch <- 1
  ch <- 1
  // 14 ^
  //ch <- 1
  //ch <- 1
  // 16 ^

  close(ch)
}

func main() {
  go sender()
  <-ch
  // 2^
  <-ch
  <-ch
  // 4 ^
  <-ch
  <-ch
  // 6^
  <-ch
  <-ch
  // 8 ^
  <-ch
  <-ch
  // 10 ^
  <-ch
  <-ch
  // 12 ^
  <-ch
  <-ch
  // 14 ^
  //<-ch
  //<-ch
  // 16 ^

  x := <-ch
  if x == 1 {
    close(ch)
  }
}
```

B. Mutexloopconst

Shown here with N (number of loops) = 8

```
package main

import (
  "fmt"
  "sync"
)

func main() {
  var m sync.Mutex

  m.Lock()
  for i := 0; i < 8; i++ {
    if i&0x1 == 0x1 {
      m.Lock()
      // do some work using lock here
      fmt.Println("working with lock")
      continue
    }
    m.Unlock()
  }

  // if above loop has run for an even number of
  // then locking will cause an error (cant double lock)
  m.Lock()
}
```

C. Mutexloopvar

Shown here with N (number of loops) = 8

```
package main

import (
  "fmt"
  "sync"
)

// could be set from some other package
var LoopIter = 8

func main() {
  var m sync.Mutex

  m.Lock()
  for i := 0; i < LoopIter; i++ {
    if i&0x1 == 0x1 {
      m.Lock()
      // do some work using lock here
      fmt.Println("working with lock")
      continue
    }
    m.Unlock()
  }

  // if above loop has run for an even number of
  // then locking will cause an error (cant double lock)
  m.Lock()
}
```

D. Loopspawn

Shown here with N (number of goroutines spawned) = 4

```
package main

func checkPanic(i int) {
  if i == 3 {
    panic("somethings wrong")
  }
}

func main() {
  for i := 0; i < 4; i++ {
    go checkPanic(i)
  }
}
```