TUHH
Hamburg University of Technology

Bachelor Thesis

Leonard Techel

Low-Connectivity State Space Exploration using Swarm Model Checking on the GPU

September 22, 2021

supervised by:
Prof. Dr. Sibylle Schupp
Sascha Lehmann

Hamburg University of Technology (TUHH)
Technische Universität Hamburg
Institute for Software Systems
21073 Hamburg

STS
Software
Technology
Systems

Abstract

Using small, independent verification tests, model checking large models with billions of states can be parallelized on GPUs. This approach works great on models with high connectivity. However, it fails when a model has only few edges between states or large portions of the state space are hidden behind bottleneck structures.

Past work on the *Grapple* model checker has tried different approaches including depth-limiting and alternating between breadth-first and depth-first search.

The main goal of this thesis is to: (1) create a systematic way of classifying a model as *low-connectivity* (2) minimize the amount of verification tests needed to maximize the state space coverage of said models. To do that, we provide an implementation of the *Grapple* model checker.

Contents

List of Algorithms

List of Figures

List of Tables

1 Introduction

In an explicit-state model checker, state space exploration of large models with billions of states is a time-consuming problem.

Using swarm verification, the problem can be split into small, independent verification tests. Past work has shown that by executing these verification tests in parallel on GPUs, a high-speed model checker can be implemented [3].

GPUs are interesting for model checking because they allow a significant performance increase on parallel algorithms, are widely available and relatively cheap in comparison to specialized hardware like FPGAs/ASICs.

To create the small, independent verification tests, diversification is used. Each verification test only covers a subset of the state space. Together, the verification tests nearly achieve full state space coverage. This approach works great on models with high connectivity. However, it fails when a model has only few edges between states or large portions of the state space are hidden behind bottleneck structures.

Contributions:

- Breakdown of the state space exploration loop using subroutines
- Implementation of a Grapple model checker
- A method of estimating unique states visited with a fixed error
- The *start over* search strategy that can reach deeper states
- A visualization of Grapple BFS frontiers

2 Related Work

Swarm Verification was first studied by Holzmann et al. as a verification method for models with large state spaces that do not allow exhaustive verification in any reasonable amount of time [8].

Continuing their work, they introduced multiple swarm verification implementation techniques, including a memory-efficient method of marking states as visited, multiple state space diversification techniques, and an evaluation of swarm verification performance on a number of real-world models [9].

This thesis is based on the *Grapple framework*, which implements Swarm Verification on the GPU using the CUDA parallel programming framework. To do so, they combined Swarm Verification and its techniques with a lock-free, parallel breadth-first search and an on-the-fly state generator specifically designed for GPUs [3].

The parallel BFS is adopted from a paper that parallelized the state space exploration loop of the SPIN model checker [7].

The state generation is adopted from a paper which studied parallelization of the SPIN model checker on the GPU [1].

3 Background

This chapter gives a brief summary on the main topics that this thesis is based on. First, we define the scope of model checking that we are looking at in Section 3.1. We continue with an overview of Swarm Verification on the GPU and the specific problem that we are going to investigate, low-connectivity state space exploration, in Section 3.2. Last but not least, we give an introduction to the CUDA GPU programming framework in Section 3.3.

3.1 Model Checking

Model checking is a formal verification method that checks whether a state machine satisfies a specification. For example, the state machine of an elevator may be verified to meet the safety property of not opening the doors between floors. To do so, a *model checker* searches the state space for counterexamples, also called violations. When a violation is found, the path of state transitions that leads to the violation is reported back.

Model checking can be divided into three main problems: Description of models through state machines, definition of the specification through temporal logic, and algorithms that verify whether a state machine models a specification.

There are two main branches of model checking: Explicit-State Model Checking and Symbolic Model Checking. Explicit-State Model Checking can only verify finite state machines. In particular, the model has to have finite states, each state needs to be representable by a finite-size tuple containing its atomic propositions, and the model changes state through execution of state transitions. To overcome these limitations and verify potentially infinite-size state machines or systems of unknown structure, Symbolic Model Checking uses the abstraction of *symbols*, each representing a set of states and transitions. Within this thesis, we are only considering explicit-state model checking.

Each state in a model is labelled with atomic propositions that hold true while the state is active. An example for such propositions are the current values of variables in a program at a given state. In a specification, different types of properties can then be expressed onto these propositions. Three common properties are reachability, safety and liveness: Reachability means that an atomic proposition holds true at some state in the future. Safety means that an atomic proposition holds true at all states in the future. Liveness means that an atomic proposition holds true infinitely often in the future, meaning that it does not happen that the atomic proposition never holds true. Within this thesis, we are only considering reachability and safety properties.

A challenge all model checking algorithms have to face is the *state explosion problem*. In an asynchronous model of n processes, each consisting of m states, the number of states grows exponentially by the number of processes, namely m^n. This means that even for small models, it is often not possible to fit all reachable states of the system into a computer's memory. Therefore, every model checking algorithm needs to reduce the state space in some sense. However, even then, a non-parallel algorithm may need a lot

of physical time for exhaustive verification of the state space. In exhaustive verification, all states are visited and checked for a violation. [2, 6]

3.2 Parallelized Model Checking

The basic operation in an explicit-state model checker is a state space exploration loop, as defined in Algorithm 1. Model checking can be speed up by parallelizing the state space exploration, for example using a parallel breadth-first search (BFS) [7].

A major challenge in parallelized BFS is the communication overhead between threads: Shared memory does not allow to easily split the work onto a cluster of heterogeneous processors or the massively parallel architecture of a GPU on which thousands of threads can exist simultaneously.

Algorithm 1 Fundamental State Space Exploration Loop

```
while there are unvisited states do
    mark state as visited
    if state violates spec then
        report path to state
```

3.2.1 Swarm Verification

Swarm Verification is a new approach on parallelized model checking [8]. It solves the challenge of parallelizing state-space search by splitting the state space exploration into many small, independent, memory-limited tasks called *Verification Tests* (VTs). Each VT only covers a small subset of the total state space and, using *diversification techniques*, uses a different search path. By executing all VTs and collecting their results, we still achieve nearly full state space coverage.

As VTs are independent of each other, we can easily execute them on heterogeneous computers. Even further, as VTs are also memory-limited, we can massively parallelize them on devices with very limited resources like GPUs.

Multiple diversification techniques can be applied, e.g. randomizing the order of nondeterministic choice on states with multiple outgoing transitions or reversing the search order. The fundamental diversification technique is state pruning using hash collisions: During state exploration, states get marked as visited in a limited-size hash table. By using a different hash function for each VT, the state space gets automatically pruned through hash collisions. The process is illustrated in Figure 3.1, that shows an example state graph on which BFS is performed by two VTs. On the left side, state B and C cause a hash collision, resulting in the subgraph originating in state C being removed. On the right side, state E and F cause a hash collision, resulting in the subgraph originating in state F being removed. Each state exploration on its own does not cover the whole state space, however, by combining the results from both searches, all states are covered again.

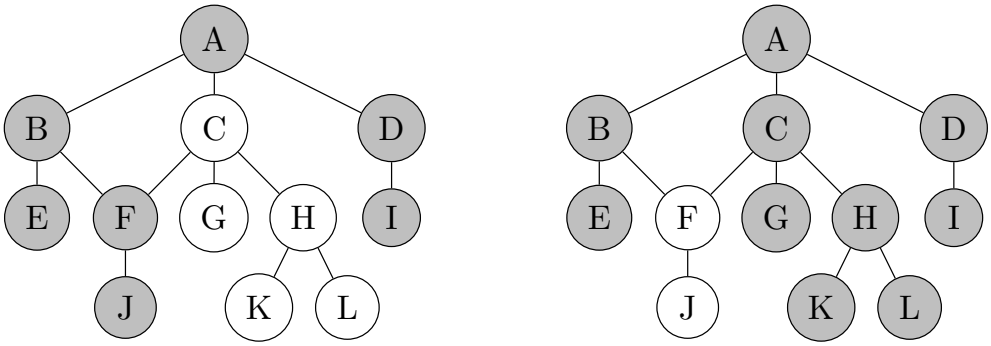


Figure 3.1: State pruning using hash collisions

3.2.2 Grapple Framework

The paper "Swarm model checking on the GPU" by DeFrancisco et al. [3] which this thesis is based on contributes the *Grapple* framework for parallel swarm verification on the GPU using CUDA. In their implementation of Swarm Verification, each VT's state space exploration runs in a parallel BFS, using CUDA's built-in synchronization primitives to coordinate threads.

3.2.3 Low-Connectivity Models

A key observation of [3] is that their algorithm's state space coverage in relation to the number of executed VTs slows down on models with *low connectivity*. A model has *low connectivity* if at least one of the following properties is satisfied:

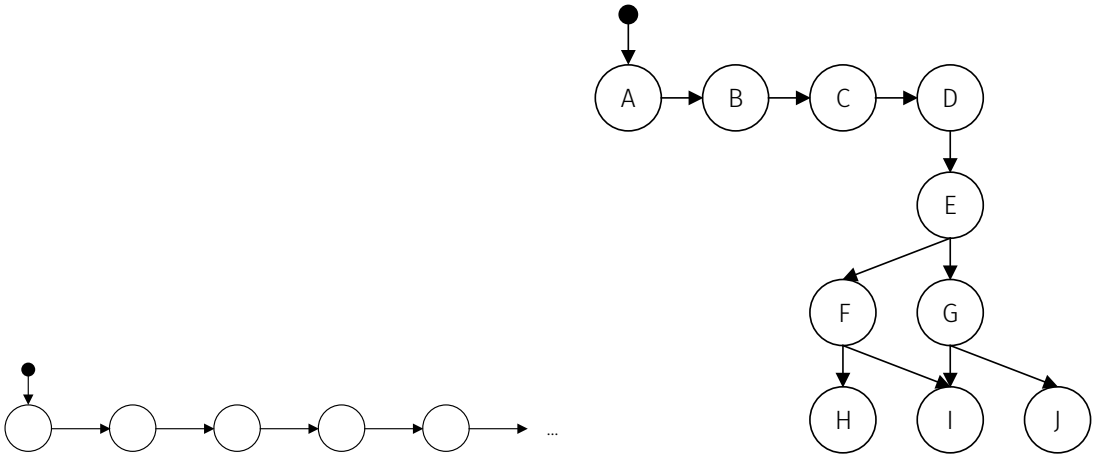
- **Generally Linear**: The average number of edges per state is close to two: One inbound, one outbound
- **Bottleneck Structures**: A single state or group of states other than the initial state that needs to be passed to reach most of the state space

See Figure 3.2 for examples of these properties.

3.3 CUDA

CUDA is NVIDIA's proprietary GPU programming framework that provides automatic massive scalability.

The CUDA model defines a three-level abstraction. At the lowest level, a program function called *kernel* is defined using C/C++. The kernel is executed by defining the amount of *threads* and *blocks* that run in a *grid*: Each grid contains multiple blocks. Each block contains multiple threads. Each thread in a block executes the kernel function using the single instruction, multiple thread (SIMT) execution model. Thus, each thread should



(a) Example of a generally linear graph (b) Example of a bottleneck structure

Figure 3.2: Examples of low-connectivity properties

work on an independent piece of memory. Only threads within a block can cooperate, for example using shared memory or synchronization barriers. Internally, the GPU, also called *device*, maps the blocks of threads onto its *streaming multiprocessors*. By doing so, tens of thousands of threads can be executed in parallel. The architecture allows scaling even further onto multiple GPUs.

Furthermore, CUDA defines multiple memory levels, each with different size and access speed.

- *Registers* are used for thread-local variables. There can be at max 255 registers à 32 bit per thread.
- *Shared memory* can only be accessed by the threads of the block in which it got allocated. As it is on-chip, it is faster than global memory. By default, there is 48 KB of shared memory.
- *Constant memory* is part of the device memory, with the constant cache in front of it. Cache hits are served at higher speed than just using global memory. There is 64 KB of constant memory.
- *Global memory* is the largest memory with usually multiple gigabytes of size. It is placed in device memory and thus also the slowest.

CUDA requires the problem to be solved by a program to be split into many small sub-problems. This is a good fit with Swarm Verification, which turns the model checking problem into many small, independent VTs.

4 Theory

This chapter gives a detailed explanation of the algorithms used by our model checker. It starts with a formal definition of transition systems in Section 4.1. We continue with an in-depth description of the Grapple framework's state space exploration loop and its components in Section 4.2 Following that, we present a list of known state space diversification techniques and our *start over* extension to the Grapple algorithm in Section 4.3. Then, an explanation of the waypoints model commonly used to benchmark swarm model checkers follows in Section 4.4. We finish the chapter with a solution to the problem of counting the number of unique states visited in Swarm Verification in Section 4.5.

4.1 Model Definition

As explained in Section 3.1, a model consists of a transition system and a specification that the transition system is verified for. To formally define a model's transition system, we are using a *Kripke Structure*

$$M = (S; S_0; R; L)$$

where S is a finite set of states, $S_0 \in S$ is a finite set of initial states $S_0 \subseteq S$, $R \subseteq S \times S$ is a left-total transition relation and $L : S \mapsto 2^{AP}$ is a labelling function that assigns each state its atomic propositions.

To define verification properties, Kripke structures are traditionally used in conjunction with temporal logic. In this thesis, we focus on the state space exploration part of model checking. Verification of states is thus abstracted away into the `state_violates` subroutine and not further covered. By calling the subroutine on every visited state, we can still be sure that all violations in the covered state space are found, as long as the subroutine is correct.

4.2 Grapple Model Checker

In Grapple, each VT executes an internally parallel state space exploration. We call it *internally parallel* to differentiate between the parallelism *within* a VT and the parallelism *between* multiple VTs. Each thread in a VT executes Algorithm 2, which is a modified version of the Grapple state space exploration loop [3, Algorithm 1] using subroutines. We use subroutines to separately reason about queue operations, successor generation and marking states as visited. We are going to call this algorithm the *Grapple algorithm*. The Grapple algorithm is based on Holzmann's parallel BFS [7].

The following explanation of the algorithm is split into four parts: We start with a classification of Grapple's state generation strategy and its constraints imposed onto models in Section 4.2.1. Based on that, we explain Grapple's parallel breadth-first search and define the corresponding queue operations in Section 4.2.2. Having the

successor generation and parallel breadth-first search ready, we can finally define the single operation of the hash table in Section 4.2.3. Last but not least, we take a look at properties of the algorithm, including termination, in Section 4.2.4.

4.2.1 Model Definition and State Generation

State generation strategies of model checkers can be divided into *a priori* and *on-the-fly* generation. In a priori generation, the state space graph is completely known before running the verification algorithm, for example as an adjacency matrix. In on-the-fly generation, the successors of each state are created on the fly during verification. Grapple uses on-the-fly generation on the GPU.

As defined in Section 4.1, the transition relation of a model is left-total, meaning that for each state, there is at least one successor. We can define a function returning the successor set of a single state by

$$\begin{aligned} \text{succ} : S &\to S' \subseteq S \\ s &\mapsto \left\{ s' : (s, s') \in R \right\} \end{aligned}$$

This function allows each state to have a successor set with different cardinality. In Grapple, the successor set of a state is called its *nondeterministic choices* (NDCs), highlighting the fact that any of them is a valid continuation of a path through the model's transition system. To leverage the performance benefit of the GPU's SIMT execution model, Grapple's successor generation has to be branch-free, so each thread can follow the same execution path. To achieve that, the Grapple algorithm constraints the models it can verify to those with an equal number of NDCs for all states, i.e.

$$\forall s_1, s_2 \in S : |\operatorname{succ}(s_1)| = |\operatorname{succ}(s_2)|$$

Furthermore, Grapple can verify multi-process models. Following that, we can extend the definition of our transition system by the number of processes and NDCs:

$$M_{\text{Grapple}} = (S; S_0; R; L; P; NDC)$$

The algorithm then uses two subroutines for successor generation and violation checking:

$$\begin{aligned} \texttt{state\_successor} &: S \times NDC \times P \to S \\ \texttt{state\_violates} &: S \to \{0, 1\} \end{aligned}$$

The subroutine `state_successor` on-the-fly returns a single successor to a state and `state_violates` returns whether a state violates.

4.2.2 Parallel BFS and Queues

We are now going to explain the parallel state space exploration loop defined in Algorithm 2, which is the main on-device routine of a Grapple model checker. A VT consists of multiple threads, each executing the state space exploration loop. Together, they perform

a parallel BFS, searching the state space for violating states. To cooperate, they are using a shared, multidimensional queue array that allows them to communicate lock-free. Being lock-free removes the overhead of waiting for locks to be released and allows the algorithm to run on a larger amount of threads, limited only by the quadratic size of the queues. To be lock-free, there is only a single piece of memory to communicate over between each pair of threads. Writing and reading to this piece of memory is coordinated by splitting the algorithm into two alternating phases:

In the first phase $t = 0$, each thread i processes all states from its input queues $Q\,[t]\,[0 \ldots N]\,[i]$. Here, N denotes the amount of threads executing a VT, with $N = 32$ being the default. For each input state, successors are generated on-the-fly, as explained in Section 4.1. For each successor, we check whether it is already visited in a hash table. If a successor is priorly unvisited, we check whether it violates. Violating successors are reported to the host using a buffer. Non-violating successors are written to a random worker j's input queue $Q\,[1-t]\,[i]\,[j]$. When all input states are processed and threads are synchronized, the phases get swapped, i.e. $t = 1 - t$. This process repeats until no more unvisited successors are discovered.

Each queue Q is a first in, first out (FIFO) queue supporting three operations: `queue_push`$(Q, state)$ adds a state to the back, `queue_pop`(Q) removes and returns the first state from the front and `queue_empty`(Q) tells whether a queue is empty.

4.2.3 Hash Table

The hash table H is primarily used to mark states as visited, so the parallel BFS at the latest terminates when all states have been visited once, resulting in an exhaustive search. In Grapple, the hash table does not resolve collisions. By deliberately accepting collisions, it serves two additional purposes: First, using a limited-size hash table, the BFS also terminates when the hash table is full, causing a collision of every new successor with an already visited state. Furthermore, by using a different hash function in each VT, the search gets diversified between VTs. By combining those two purposes, each VT searches a different portion of the state space, as explained in Section 3.2.1. The hash table supports one operation that marks a state as visited and returns whether it was already visited before:

$$\texttt{mark\_visited} : H \times S \to \{0, 1\}$$

4.2.4 Properties of the algorithm

Validity The algorithm can only verify models with an equal number of NDCs for all states, as explained in Section 4.2.1. Furthermore, due to the `state_violates` subroutine being called without path, only safety and reachability properties on the covered state space can be verified.

Completeness A big challenge in Swarm Verification is the question on how much of the state space is actually covered, i.e. whether all violations are found. Due to overlap

in state space between VTs and the sheer amount of states being checked by each VT, computing the exact number of unique states visited and thus the exact state space coverage can hardly be done. Instead, we can estimate the state space coverage, as described in Section 4.5.

Termination The algorithm terminates when there are no more unvisited state successors according to the hash table, as explained in Section 4.2.3. In this case, all input queues of the next phase remain empty, i.e. $Q\,[1-t]\,[0\ldots N]\,[0\ldots N] = \emptyset$. Resulting from this, the algorithm at worst terminates after having visited the number of states being the minimum of queue size and state space size.

4.3 State Space Diversification

The goal of swarm verification is to search for violations in state spaces that are too large for traditional, exhaustive verification. To reach a broad coverage throughout the state space, multiple diversification techniques can be applied, each altering the search path of each VT.

This section starts with a list of known diversification techniques in Section 4.3.1. We then present a new extension to swarm verification called *start overs* that allows the exploration to reach deeper states and at the same time saves memory by using smaller hash tables in Section 4.3.2.

4.3.1 Diversification Techniques

The following diversification techniques are known: [9, 3]

- State Pruning using Hash Collisions
  Each VT marks its visited states in a limited-size hash table, each using a different hash function. Hash collisions cause different partitions of the state space. Hash table exhaustion causes termination of the exploration when all new states are supposedly already visited.
- Reverse order of nondeterministic choice
  Iterate through nondeterministic choices in reverse order, i.e. $NDC, \ldots, 0$.
- Reverse processes
  Iterate through processes in reverse order, i.e. $P, \ldots, 0$.
- Random order of nondeterministic choice
  Instead of iterating through nondeterministic choices from $0, \ldots, NDC$, choose one random order for each VT. The random order remains the same through all searches in a VT.
- Random process order
  Instead of iterating through processes from $0, \ldots, P$, choose one random order for each VT. The random order remains the same through all searches in a VT.

Algorithm 2 Grapple state space exploration loop of a single worker

```
t ∈ {0,1} ← 0                                   ▷ Current algorithm phase
H ← ∅                                           ▷ Hash table of visited states
Q[2][N][N][I] ← ∅                               ▷ Queues

▷ Initial state                                 ◁
queue_push(Q[t][0][0], S_0[0])
mark_visited(H, S_0[0])

__syncthreads()

done ← false
while not done do
    for i ← 0, ..., N do
        while not queue_empty(Q[t][threadIdx.x][i]) do
            state ← queue_pop(Q[t][threadIdx.x][i])
            for p ← 0, ..., P do
                for ndc ← 0, ..., NDC do
                    succ ← state_successor(state, p, ndc)
                    visited ← mark_visited(H, succ)
                    if not visited then
                        if state_violates(succ) then
                            report path to state
                        else
                            next ← random output queue n ∈ N
                            queue_push(Q[1 − t][next][threadIdx.x], succ)

    __syncthreads()

    done ← queue_empty(Q[1 − t][0 ... N][threadIdx.x])
    t ← 1 − t

    __syncthreads()
```

- Parallel Deep Search (PDS)
  Select one random nondeterministic choice per VT, discard others. The selected nondeterministic choice remains the same through all searches in a VT.

- process-PDS
  Select one random process per VT, discard others. The selected process remains the same through all searches in a VT.

4.3.2 Start Overs

A key observation on the hash table is that the rate of hash collisions and by that the diversification between VTs can only be raised by lowering the hash table size. This is a result of the nature of hash functions, mapping from an infinite domain of unknown distribution to a finite co-domain of an approximately uniform distribution.

A key observation on the state space exploration is that its maximum depth is mainly determined by the hash table capacity, as explained in Section 4.2.3. As each VT again starts at the initial state, even the union of the results of all VTs may potentially cover only the tip of a model.

Following these two observations, we invented the *Start Over* strategy, which is an extension to the Grapple algorithm that is self-contained within each VT. Start overs work as follows: Each VT keeps a ring buffer of the last unvisited successors it has found. When the parallel BFS of a VT terminates, queues and hash table are cleared, the last unvisited successors are pushed into the queues as new, initial states and the BFS is started again. This process loops a predefined amount of times.

By continuing the BFS within a VT using the set of last unvisited successors, we can reach deeper states of the model that are reachable from the original initial state. By exploring deeper states, we expect a faster growing state space coverage and thus to discover more violations. Start overs allow us to lower the hash table size to cause more collisions, as we can mitigate the decrease of visited states by adding additional start overs.

4.4 Waypoints Model

The waypoints model (WP) is a benchmark for swarm verification model checkers. It is first introduced in [9] and used as primary benchmark in [3]. Our implementation is defined in Algorithm 3.

The idea of the waypoints model is to go through all 32-bit integers, each representing a single state of the model. This results in a large state space of $2^{32} = 4\,294\,967\,296$ states. To do so, it uses eight processes, each in control of four bits. At successor generation, each process will nondeterministically set one of its bits.

There is a single reachability property: We predefine a set of 100 uniform randomly chosen 32-bit integers called *waypoints*. A state violates if it is part of the set. As the waypoints are uniformly distributed, the number of unique discovered waypoints is equal

to the percentage of state space covered. For example, 24 discovered waypoints means that approximately 24 % of the state space is covered.

Algorithm 3 Waypoints model

```
violations ← {100 random 32-bit integers}

function STATE_SUCCESSOR(state, p, ndc)
    return state | 1 << ((4 · process) + ndc)

function STATE_VIOLATES(state)
    return state ∈ violations
```

4.5 Counting Unique States Visited

In this section, we introduce a method for estimating the amount of unique states visited with a fixed error across all VTs in a swarm verification. As swarm verification achieves a much faster verification by only covering *nearly* 100 % of the state space, unique states visited are an important quantity to give guarantees on what *nearly* actually means. It is also important to evaluate the exploration of low-connectivity models, as they lower the growth of state space coverage, which can be observed and quantified using the unique states visited.

Executed VTs may overlap in explored state space, meaning that across all VTs, a state may be visited multiple times. In order to calculate the exact number of unique states visited, we have to identify distinct states in the stream of all visited states. This is called the *count-distinct problem*.

Intuitively, one could collect all visited states in a set and then take its cardinality. However, this approach requires an amount of memory proportional to the amount of visited states which, as of the state space explosion problem, results in exponential memory usage. A solution to this problem are *probabilistic cardinality estimators* that approximate the number of unique states within a fixed error using significantly less memory.

We choose the HyperLogLog++ (HLL) algorithm as it is commonly used, supports estimating large cardinalities beyond a billion, and is relatively easy to implement [5]. For each VT, we create a HLL. Inside each VTs state space exploration, we call the `add` operation for each newly discovered state. When a VT has finished, we `merge` its HLL with those of all other already finished VTs. We then can execute the `estimate` operation on the global HLL to get the estimation of unique visited states across all finished VTs. By only estimating the cardinality, the HLL can operate on a fraction of memory. Estimating cardinalities of $> 10^9$ with an error of 2 % can be done in 1.5 kB of memory [4].

The relative error introduced by a HLL is constant and calculated using $\sigma = 1.04/\sqrt{m}$ where m is the amount of registers used by the HLL. The 65 % error bound for each value

x estimated by the HLL is then $[x \cdot (1 - \sigma), x \cdot (1 + \sigma)]$. 95 % and 99 % error bounds can be calculated using 2σ and 3σ.

To calculate the state space coverage in percent, we furthermore have to know the total state space size. As estimating the state space size of unknown models is out of scope of this thesis, we can only calculate coverage of models with known state space size.

5 Implementation

Our implementation serves two purposes: To reproduce the results from the Grapple paper [3]. And to run our experiment series on the state space exploration of low-connectivity models.

This chapter documents our implementation of a Grapple model checker. The general architecture of the model checker is described in Section 5.1. Usage instructions are described in Section 5.2.

5.1 Source code

We chose C++17 as programming language, so we can make full use of the CUDA Toolkit, which is provided as C header. The CUDA Toolkit is used in version 11.4. As build system, CMake 3.16 is used. Code is written using the Object-Oriented Programming paradigm. Our implementation consists of six components:

- The main program, running the CUDA kernels and collecting their results.
- The CUDA kernel, implementing a parallel state space exploration loop.
- The queues, providing lock-free communication between threads in a VT.
- The hash tables in which states are marked as visited.
- The model, providing successor generation and violation checking.
- The HyperLogLog, counting unique states visited across all VTs.

In the following, we are going to describe the implementation-specific details and challenges of each component.

5.1.1 Main Program

The main program runs in a single thread on the host. On startup, it seeds a global pseudorandom number generator (PRNG) and creates a random value for each CUDA thread. Then, for each run, it executes the CUDA kernel in a grid of $K = 250$ blocks, each consisting of $N = 32$ threads. Each block represents a VT, meaning that VTs are executed in batches of 250. Each thread represents a worker of a VT's parallel state space exploration. After each run, the main program collects the discovered violations and number of unique states visited, accumulates them and prints them to the standard output line-by-line as CSV.

5.1.2 CUDA Kernel

The CUDA kernel executes the state space exploration loop, as described in Section 4.2.

5.1.3 Queues

The queues provide a data structure for lock-free communication between threads in a VT, as described in Section 4.2.2. By default, each queue has a capacity of $I = 4$ states. They are only used on-device within the CUDA kernel and stored in global memory due to the limited size of shared memory. Resulting from this, we have to map $K \times 2 \times N \times N \times I$ queues into memory. This proposes two challenges:

1. We have to implement fixed-capacity queues.
2. We have to map multidimensional queues into a one-dimensional, linear memory allocation.

Usually, fixed-capacity queues are implemented as ring buffer. However, due to the design of the two-phase parallel state space exploration, it is sufficient to implement the queue as a singly linked list. In addition to that, we have to keep a pointer to the `head` and `tail` of the linked list. Then, in the first phase, each queue is filled through the `queue_push` operation by adding an element to its linked list and updating the `tail`, until the `tail` points to the last element in memory. In the second phase, each queue is completely emptied through the `queue_pop` operation by retrieving the `head`, then incrementing it until `head` equals `tail`. A queue is empty if both `head` and `tail` are null pointers.

The memory address of thread i's input queue, which is an output queue of thread j, in algorithm phase t, of VT v, is calculated using the formula:

$$v \cdot (2 \cdot N \cdot N) + t \cdot (N \cdot N) + j \cdot N + i$$

Figure 5.1 illustrates the mapping of a single example VT with $K = 1$ and $N = 4$, meaning that we allocate memory for $1 \times 2 \times 4 \times 4 = 32$ queues. On the top row, each slot represents the memory allocation of a single queue. The remaining rows illustrate the constants enclosed in parentheses in our formula: Each set of output queues has a width of N, which we have to multiply with the index of the current thread to get at its first memory slot. Each phase has a width of $(N \cdot N)$, which we have to multiply with the index of the current phase. Each VT has a width of $(2 \cdot N \cdot N)$, which we again have to multiply with the index of the current VT to get to its first memory position. By adding up all products, and adding the index of the current input queue, we get the address of the exact slot.

5.1.4 Model and Successor Generation

Models are implemented using a data structure that represents a current state. Each state instance then provides the two operations `state_successor` and `state_violates`. The state instances are stored in the queues.

State generation is usually heavily based on branching through `switch`- and `if`-conditions. However, CUDA's SIMT execution model, in which warps of 32 threads operate on same instruction, causes threads to pause when their execution paths diverge.

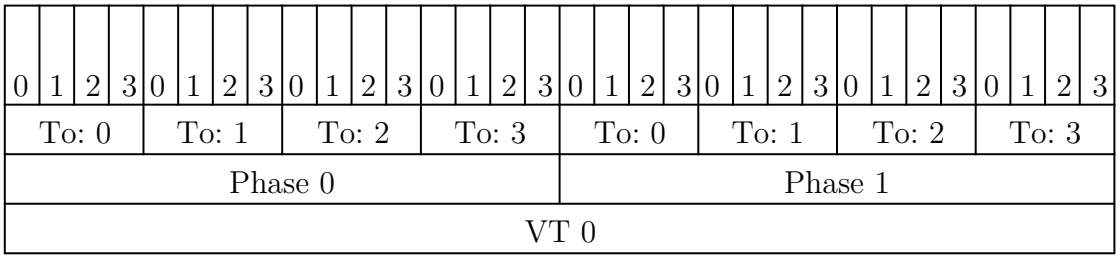


Figure 5.1: A single VT of the 4D array, mapped to 1D memory

This means that state generation through branching results in a slower CUDA execution. As a countermeasure, Bartocci et al. propose to calculate all possible state transitions every time, preserving the SIMT execution and at the same time creating different states in each thread [1, Algorithm 3]. To do so, it makes use of the custom ternary operator described in Algorithm 4 that exploits the fact that boolean values evaluate to either zero or one.

Algorithm 4 Branching-Free Ternary Operator

function BranchFreeTernary(bool c, int t, int f)
  return $c \cdot t + (1 - c) \cdot f$

A key difference of our implementation is that we do not fit a state's variables into a single integer using bit shifts. Instead, we use regular C++ class data members. The increased memory demand by regular data members can be neglected, as the queues in which we store state instances are in global memory which is usually gigabytes in size. Also, the same behavior as in the paper can be achieved using C++ bit field members.

Using our custom ternary, we can re-use DVE models from the discontinued BEEM model database[1]. DVE is a file format used by the DIVINE 3 model checker. We translate DVE models into CUDA compatible C++ with the template described in Algorithm 5.

5.1.5 Hash Tables

The hash tables provide a data structure in which states are marked as visited. They are only used on-device within the CUDA kernel and stored in shared memory, resulting in a highly constrained size. Each VT operates on a single hash table, meaning that all threads in a block share one hash table. For memory efficiency, we use bitstate hashing [3]. Our hash tables can store $2^{18} = 262\,144$ states, taking up 32 768 bytes of shared memory.

In bitstate hashing, a single bit represents whether a state is visited. Per clock cycle, CUDA can transmit 32 bit of shared memory, which is stored in 32-bit words. Resulting from this, we use a bucket size of 32 bit. The bit representing whether a state is visited is then computed using Algorithm 6.

[1] `https://paradise.fi.muni.cz/beem/`

Algorithm 5 On-The-Fly State Generation on the GPU

```
▷ Global variables ◁
glob ← 0
▷ Process-Local variables use arrays. Here, N is the number of processes ◁
state[N] ← {0, ..., 0}

function STATE_SUCCESSOR(p, ndc, state)
    ▷ Evaluate all guards before calculating transitions ◁
    guard_1 ← state[p] = 0
    guard_2 ← state[p] = 1 ∧ glob ≤ 2
    guard_3 ← state[p] = 1 ∧ glob > 2
    ▷ Transition from state 0 to 1 ◁
    next.state ← BranchFreeTernary(guard_1, 1, state.state)
    ▷ Transition from state 1 to 1 ◁
    next.glob ← BranchFreeTernary(guard_2, state.glob + 1, state.glob)
    next.state ← BranchFreeTernary(guard_2, 1, next.state)
    ▷ Transition from state 1 to 0 ◁
    next.glob ← BranchFreeTernary(guard_3, 0, next.glob)
    next.state ← BranchFreeTernary(guard_3, 0, next.state)

    return next
```

The hash function's goal is to map random data with unknown distribution onto a nearly uniform distribution, so each bucket of the hash table is used equally likely. In our implementation, hashing is done using the Jenkins Hash function[2]. The MurMurHash3[3], which is used by our HyperLogLog, yields similar results.

Hash collisions resulting in false-positives are intended, as explained in Section 4.2.3. To achieve that in every VT different states cause a collision, each is including a different seed into the hash. The rate of collisions can then be controlled by the hash table's size.

5.1.6 HyperLogLog++

The HyperLogLog (HLL) provides an algorithm for distributed counting of distinct elements, as described in Section 4.5. In our implementation, the grid of VTs of each run is operating on a single HLL that is stored in global memory. Visited states are added to the HLL only on-device. The host collects the HLLs after the kernels have finished, merge them into one big HLL and does the estimation of unique states visited. Merging and estimating the amount of unique states visited is done only on the host.

[2] `https://www.burtleburtle.net/bob/hash/doobs.html`
[3] `https://github.com/aappleby/smhasher`

Algorithm 6 Bitstate hashing

```
▷ Initialize hash table buckets. Here, N is the hash table size ◁
table[(1 << N − 5)] ← {0, . . . , 0}

function MARK_VISITED(table, state)
    hash ← make_hash(state, seed) & (1 << N) − 1
    ▷ First 5 bits (2^5 = 32) are the index within the bucket ◁
    elem ← hash >> (N − 5)
    ▷ Last N − 5 bits are the bucket index ◁
    bucket ← hash & (1 << N − 5) − 1
    ▷ Retrieve current state ◁
    isVisited ← table[bucket] & (1 << elem) ≠ 0
    ▷ Update hash bucket ◁
    table[bucket] ← table[bucket] | (1 << elem)

    return isVisited
```

5.2 Usage

To verify a model using our model checker, a user has to complete the following steps:

1. Describe their model in a C++ class using Algorithm 5
2. Compile our model checker including their model
3. Execute the single output binary. On execution, the user may set the PRNG seed and number of runs using command-line options. The number of runs needed to achieve a certain state space coverage needs to be evaluated experimentally.
4. Interpret the CSV output

6 Evaluation

This chapter presents our series of empirical experiments conducted on our implementation. It starts with experiments on the correctness of our implementation and a comparison with the Grapple paper's results in Section 6.1. We then continue with a series of experiments that compare the Waypoints model, which is known to be high-connectivity, with three low-connectivity models in Section 6.2.

All experiments are performed on an NVIDIA GeForce RTX 2080 Ti GPU, AMD Ryzen Threadripper 2990WX CPU and 128 GB main memory using Ubuntu 20.04.2 LTS. All values related to state space coverage are estimated using a HLL with 2^{14} registers and thus prone to an error, as explained in Section 4.5.

6.1 Correctness and Comparison with the Paper

We start the evaluation with whether our implementation reaches the same number of discovered waypoints in relation to the amount of executed VTs as presented in the Grapple paper. We would expect both implementation to behave similarly, discovering at least 80 waypoints in 20 000 VTs and all 100 waypoints in about 100 000 VTs, as shown in Figure 6.1a. However, that is clearly not the case, as presented in Figure 6.1b. Our implementation found significantly fewer waypoints, even though the amount of VTs was 2.5x larger.

To further investigate differences between the implementations, we conducted additional experiments.

6.1.1 Comparison of Waypoints and HLL as Estimators of State Space Coverage

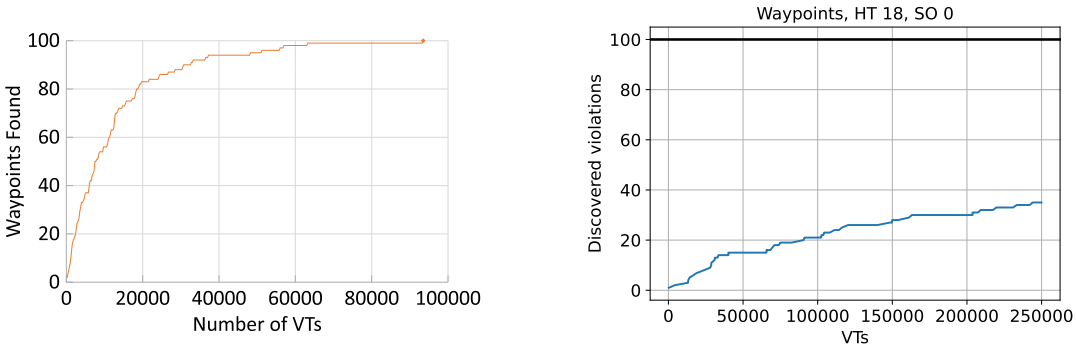
As explained in Section 4.4, counting waypoints is a method for estimating the achieved state space coverage. To make sure that the discovered waypoints of our implementation correctly indicate the state space coverage, we compared them to the state space coverage estimation of the HyperLogLog introduced in Section 4.5. We would expect both estimators to show similar results. For example, when 20 waypoints are discovered, the HyperLogLog estimation should show about 20 % state space coverage.

Figure 6.2 presents results for both estimators. Clearly, both resemble each other closely. Thus, we assume them to work correct. This also confirmed the results from our first experiment, underlining our implementation's deficient state space exploration.

6.1.2 Impact of Hash Table Size and Sequential VT Execution

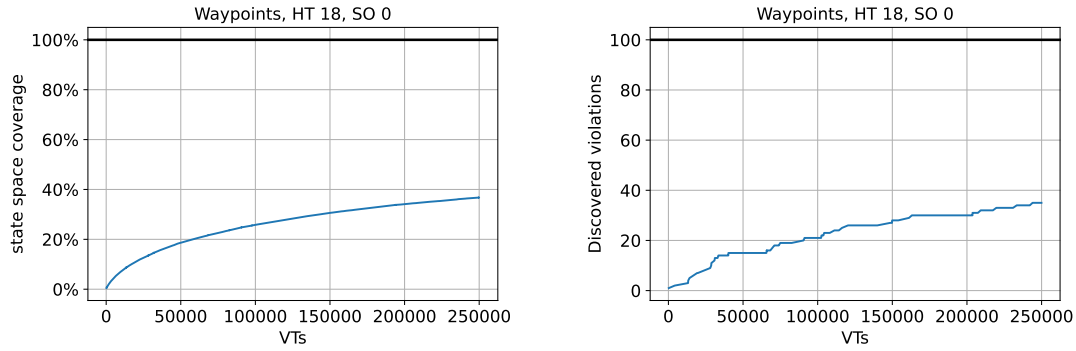
Both implementations share a queue size of $32 \times 32 \times 4$. Beside of that, we identified two key differences of our implementation:

- Our hash tables are 33 % smaller than in the paper. This is due to our bit shift implementation presented in Section 5.1.5 which cannot address the full 48 KiB of



(a) Results from the paper (b) Results of our implementation

Figure 6.1: Waypoints model, comparison between reference and our implementation



(a) HLL state space coverage estimation (b) Waypoints state space coverage estimation

Figure 6.2: Comparison between Waypoints and HyperLogLog state space coverage estimation

shared memory.

- We batch VTs in a CUDA kernel, each executing 250 at once. In the paper, there seems to be only a single VT per CUDA kernel.

To find out whether increasing the hash table size to full 48 KiB increases waypoint discovery such that it is similar to the results from the Grapple paper, we experimentally changed our implementation's addressing. As the hash table capacity should significantly affect the state space coverage of each VT, we would expect results close to those from the paper. For each experiment, we executed 80 runs à 250 VTs (= 20 000 VTs in total). Results are shown in Table 6.1.

The larger hash table yielded 2.1x as much state space coverage at only 1.45x as many total states visited. Thus, a larger hash table seems beneficial on this particular model.

However, in the paper, a Grapple swarm verification with 392 800 hash table slots per VT discovered 85 waypoints in 20 000 VTs, equalling 85 % state space coverage, which is

Table 6.1: Exploration of the waypoints model using different hash table capacities in 80 runs à 250 VTs

HT Capacity	State Space Coverage	Total States Visited
262 144	11.04 %	63 568 607
393 056	23.14 %	91 937 680

Table 6.2: Parallel and sequential exploration of the waypoints model

VTs per Kernel	Runs	State Space Coverage	Total States Visited	Execution Time
1	250	0.456 %	63 568 343	26.691 s
250	1	0.450 %	63 568 498	0.754 s

about 3.7x as much as our large-size hash table implementation. Resulting from this, the large-size hash table is still far from reproducing the paper's results.

In the paper, supposedly one VT is executed per *grid*, resulting in one CUDA kernel per VT. To compare our parallelization of VTs in a single CUDA kernel with the sequential execution of multiple kernels, we switched the compiled number of VTs and the executed runs. We would expect both variants to create similar results. Furthermore, we would expect a shorter execution time of the parallel execution as it only needs a single kernel setup. Results for both variants are shown in Table 6.2. The state space coverage and total visited states of both experiments are highly similar. The execution time of parallel execution was more than 35x faster. Concluding this, the experiment showed that batch execution of VTs highly speeds up execution time at equal results.

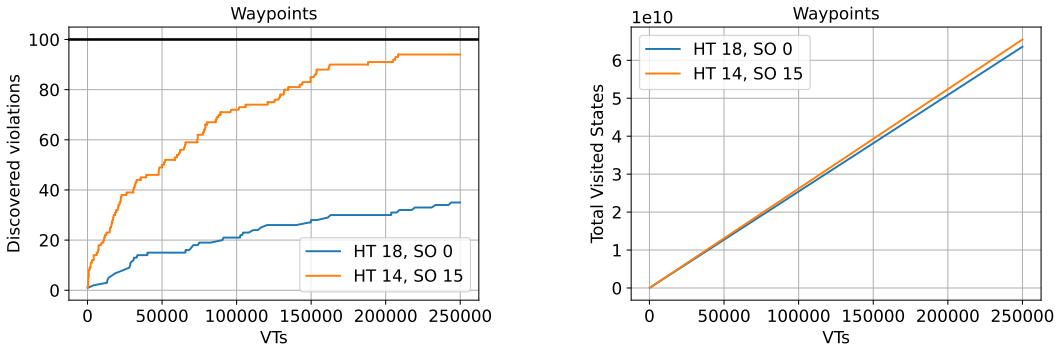
The last two experiments showed that there seems to be a lack in our implementation beside the hash table capacity, queue size and execution order.

6.1.3 Increasing State Space Coverage using Start Overs

We suspect that part of the bad exploration performance is that each VT can only reach as deep into the state space as its hash table can find unvisited states. As a countermeasure, we introduced the start over strategy in Section 4.3.2.

The next experiment evaluates whether start overs actually increase the growth of state space coverage and thus the model checking performance. By reaching deeper states, we would expect to discover more violations and thus a faster growing state space coverage. To do so, we compared 1000 exploration runs of the waypoints model with an arbitrary, much lower hash table capacity of $2^{14} = 16\,384$ slots and 15 start overs with the default configuration of $2^{18} = 262\,144$ hash table slots and 0 start overs. We chose 15 start overs to mitigate the smaller hash table, which is $\frac{1}{16}$ the size, resulting in 16 total searches, including the initial search. Results for both explorations are presented in Figure 6.3a. With start overs, the exploration took 243 s. Without, it took 442 s.

As the number of total visited states by a VT varies between the two explorations due



(a) Comparison of discovered waypoints

(b) Comparison of total visited states

Figure 6.3: State space exploration with start overs

to for example different hash collisions, we also compared the cumulated number of total visited states of both explorations in Figure 6.3b.

Clearly, the start over strategy resulted in a significant increase of discovered violations on a similar number of visited states and VTs. Also, the execution time of the start over strategy was 1.82x faster. We conclude that start overs create a significant performance increase on the waypoints model.

6.2 Low-Connectivity Model Evaluation

In this section, we try to make key differences of low-connectivity state space exploration tangible. The paper suspects three models of being low-connectivity:

- Dining philosophers problem
- Anderson queue lock mutual exclusion algorithm
- Peterson mutual exclusion protocol

6.2.1 Dining Philosophers Problem with Different Numbers of Processes

The paper suggests that a state space size surpassing the hash table capacity significantly slows down the state space exploration, resulting in a higher number of VTs needed to achieve $> 99\,\%$ state space coverage. To find out how much the state space size affects the exploration performance, we compared three variants with 11, 12 and 13 processes of the dining philosophers problem. Using our default hash table capacity of 2^{18} slots, we would expect a slow-down starting with 12 processes, as fewer processes have a state space size smaller than the hash table capacity. As presented in Table 6.3, exploring DP-13 took 213x the VTs as for DP-11, even though it is only 9x its state space size. This confirmed our hypothesis of a significant slow-down for this particular model.

Table 6.3: Exploration of the dining philosophers problem with 11, 12 and 13 processes

Model	State Space Size	Hash Table Utilization	# of VTs to explore
DP-11	177 146	68 %	100 % in 250 VTs
DP-12	531 440	203 %	100 % in 250 VTs
DP-13	1 594 322	608 %	100 % in 53 250 VTs

Table 6.4: Exploration of low-connectivity models, each after 125 000 VTs

Model	Unique States Visited	Total States Visited	Rate
Anderson 3	$1.227\,79 \times 10^6$	15 477 969 183	0.008 %
Peterson 5	$4.085\,78 \times 10^6$	24 670 432 336	0.017 %
DP 15	$1.382\,13 \times 10^7$	30 436 723 433	0.045 %
Waypoints	$1.217\,59 \times 10^9$	35 725 590 331	3.408 %

6.2.2 Comparison of Unique States Visited, Total States Visited and State Space Coverage

In the paper, the models yielded very different results concerning the unique and total states visited. Thus, the next experiment compares these performance indicators for our set of models on a fixed amount of 125 000 VTs. We would expect clear differentiation between the models and eventually a clear distinction between high- and low-connectivity. Our results are displayed in Table 6.4. Clearly, the waypoints model achieved the highest number of total states visited. We further observed an interesting property by taking the unique states visited in relation to the total states visited, presented in the *Rate* column: All three low-connectivity models achieved a rate $< 0.1\,\%$. In conclusion, the performance deficiency of the three presented low-connectivity models can be observed through unique states visited and total states visited.

6.2.3 BFS Frontiers of Low-Connectivity Models

The algorithm introduced in Section 4.2 is operating in alternating phases that produce so-called *frontiers*. Each frontier is an intermediate state of the BFS, representing the deepest states found until now. A new frontier is built by checking whether generated successors were visited before. For this experiment, we call already visited states *failed* and states added to the new frontier *visited*. To find out whether different models affect the BFS frontiers in terms of visited and failed states, we executed a single VT and counted visited and failed states for each frontier generated.

In Grapple, each BFS starts with a single initial state. As of this, we would expect the number of visited states to quickly build up until the number of failed states takes over due to the hash tables filling up.

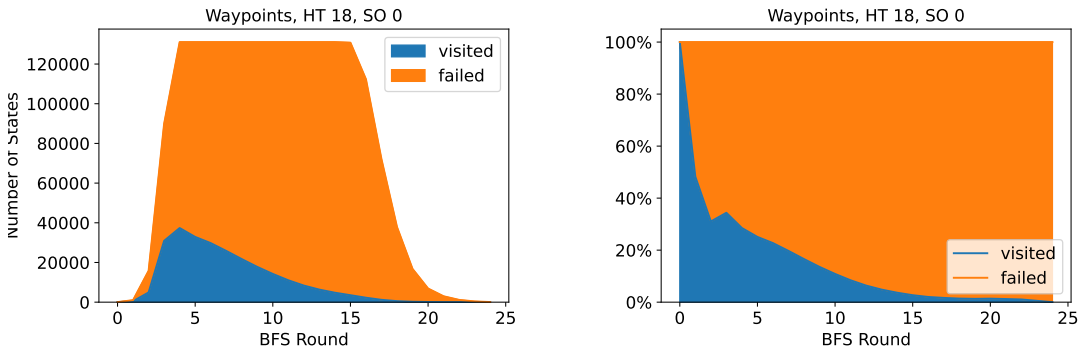


Figure 6.4: BFS frontier visualization of the waypoints model

For visualization of BFS frontiers, we provide two different methods:

- Plotting the absolute number of visited and failed states.
- Plotting visited and failed states in relation to each other, both adding up to 100 %.

Figure 6.4 presents both visualizations for the waypoints model. As expected and shown in the absolute representation on the left, the amount of visited states quickly built up and then decayed until the BFS terminated. We observed similar behavior in the relative representation shown on the right: The search started with only visited states, which decayed quickly as the failed states took over until the search terminated. The relative representation contained an interesting anomaly in the build up phase between BFS round 0 and 5, where in one round, the relative amount of visited states suddenly grew again.

Visualizations of low-connectivity models are presented in Figure 6.5. The dining philosophers and Peterson model both showed a behavior similar to the waypoints model. There is one characteristic only shared by the low-connectivity models: In their relative representations, they all had two spikes of visited states right before their search terminated. We suspect them to be a characteristic of low-connectivity models, but this assumption needs additional research.

The Anderson model in contrast showed completely unexpected behavior: In the absolute representation on the left, neither a constant maximum of visited and failed states, nor the typical build-up-then-decay-process could be observed. Instead, there is a notch between BFS round 50 and 100 where the amount of both visited and failed states over multiple BFS rounds went down, then built up again. We suspect that this notch could be a sign for a bottleneck, but this assumption also needs additional research. Furthermore, the relative representation on the right showed a near constant amount of around 40 % visited states per BFS round, with two salient spikes on the end before out of a sudden the search terminated.

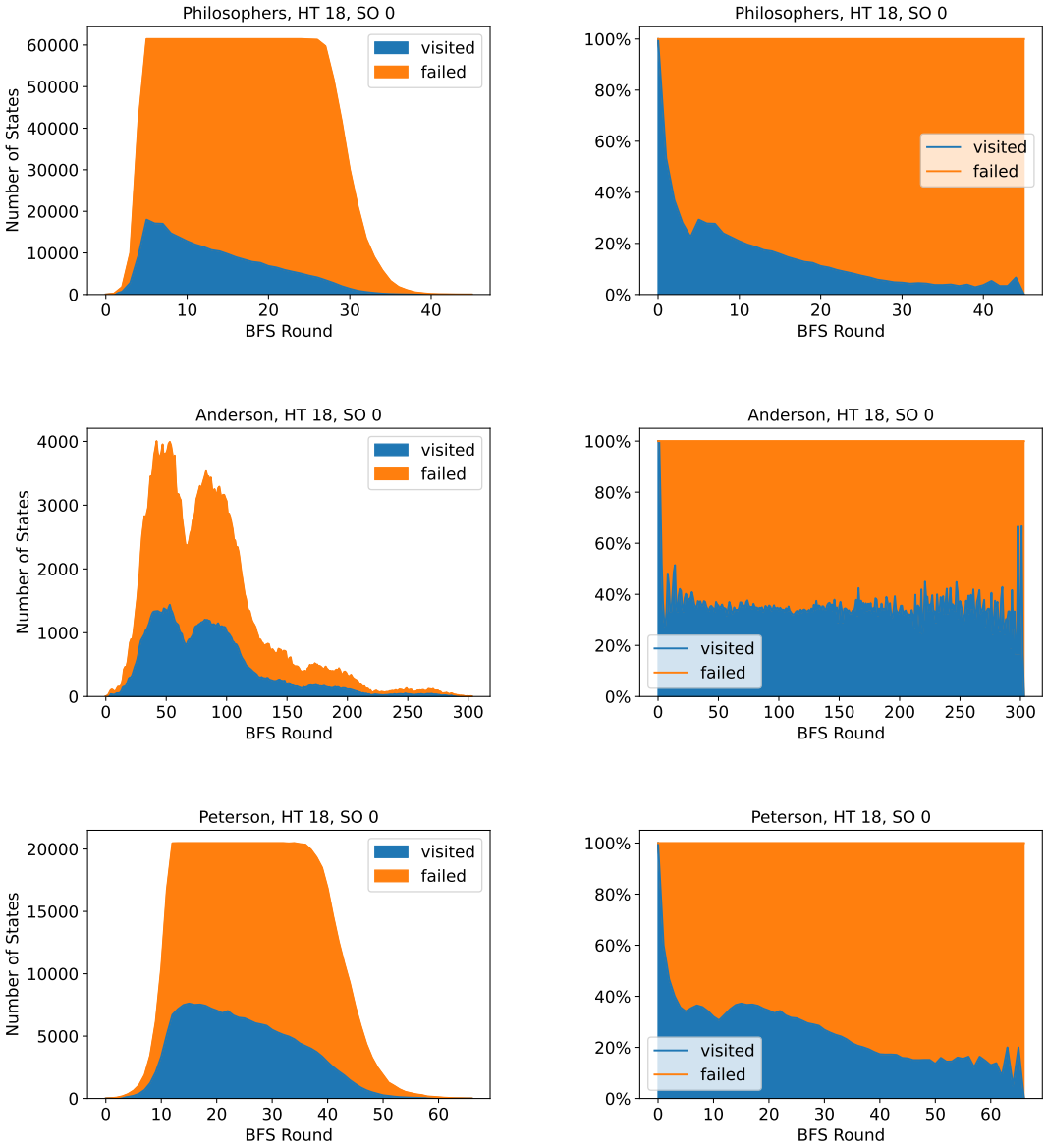


Figure 6.5: BFS frontier visualization of low-connectivity models

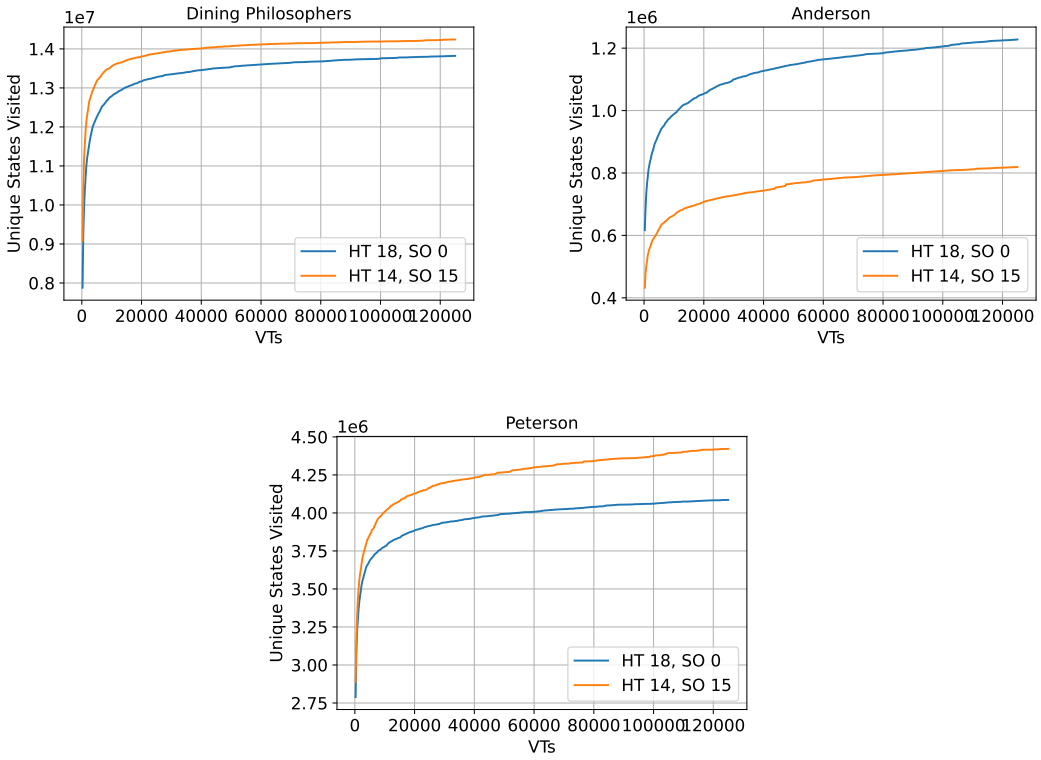


Figure 6.6: Start over strategy on low-connectivity models

6.2.4 Start Over Strategy on Low-Connectivity Models

Our last experiment evaluates whether the start over strategy yields the same performance benefit shown in Section 6.1.3 on low-connectivity models. For each of our three low-connectivity models, we conducted two explorations of 500 runs à 250 VTs. One with 2^{18} hash table slots and 0 start overs, the other one with 2^{14} hash table slots and 15 start overs (same configuration as in Section 6.1.3). As the paper has shown that increasing the search depth can increase coverage on low-connectivity models, we would expect our start over strategy to do so as well.

Plots of the results are presented in Figure 6.6. The most interesting observation is that on the Anderson model, start overs significantly worsened the state space coverage in relation to the executed VTs. Beyond that, the dining philosophers and Peterson model fulfilled our expectation with a slightly improved state space coverage. Looking at the execution time, start overs sped up the execution time on all three models: The dining philosophers model executed 1.15x faster, the Anderson model 1.82x faster and the Peterson model 1.2x faster.

In conclusion, the experiment has shown that start overs can actually improve exploration performance in terms of state space coverage and execution time on the presented models, with the Anderson model being an exception that needs further investigation.

7 Conclusion

7.1 Future Work

- Estimate state space size for models of unknown size
- Automatically determine the optimal number of start overs
- Implement usage of multiple GPU devices through CUDA streaming API
- Analyze State Space Diversification, i.e. ”how good is a partition?”

7.2 Discussion

- As in the evaluation: The Anderson model draws special interest: Its BFS frontiers show a unique behavior and using the start over strategy, it performs contrary to all other models.

Bibliography

[1] Ezio Bartocci, Richard DeFrancisco, and Scott A. Smolka. "Towards a GPGPU-Parallel SPIN Model Checker". In: *Proceedings of the 2014 International SPIN Symposium on Model Checking of Software*. SPIN 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 87–96. ISBN: 9781450324526. DOI: 10.1145/2632362.2632379.

[2] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. "Introduction to Model Checking". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 1–26. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_1.

[3] Richard DeFrancisco et al. "Swarm model checking on the GPU". In: *International Journal on Software Tools for Technology Transfer* 22.5 (Oct. 2020), pp. 583–599. ISSN: 1433-2787. DOI: 10.1007/s10009-020-00576-x.

[4] Philippe Flajolet et al. "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm". In: *Discrete Mathematics & Theoretical Computer Science* DMTCS Proceedings vol. AH, 2007 Conference on Analysis of Algorithms (AofA 07) (Jan. 2007). DOI: 10.46298/dmtcs.3545.

[5] Stefan Heule, Marc Nunkesser, and Alexander Hall. "HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm". In: *Proceedings of the 16th International Conference on Extending Database Technology*. EDBT '13. Genoa, Italy: Association for Computing Machinery, 2013, pp. 683–692. ISBN: 9781450315975. DOI: 10.1145/2452376.2452456.

[6] Gerard J. Holzmann. "Explicit-State Model Checking". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 153–171. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_5.

[7] Gerard J. Holzmann. "Parallelizing the Spin Model Checker". In: *Model Checking Software*. Ed. by Alastair Donaldson and David Parker. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 155–171. ISBN: 978-3-642-31759-0. DOI: 10.1007/978-3-642-31759-0_12.

[8] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. "Swarm Verification". In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. 2008, pp. 1–6. DOI: 10.1109/ASE.2008.9.

[9] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. "Swarm Verification Techniques". In: *IEEE Transactions on Software Engineering* 37.6 (2011), pp. 845–857. DOI: 10.1109/TSE.2010.110.