TUHH
Hamburg University of Technology

Bachelor Thesis

Leonard Techel

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

September 27, 2021

supervised by:
Prof. Dr. Sibylle Schupp
M.Sc. Sascha Lehmann

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

STS
Software
Technology
Systems

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig sowie ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Hamburg, den 27. September 2021

Abstract

Explicit verification of all states in a software or hardware system through model checking can be time-consuming due to the state explosion problem. Swarm verification is an approach to tackle the state explosion problem by splitting the verification into many small, independent verification tests that can be massively parallelized, for example on the GPU. Past work on the Grapple model checking framework has shown that low-connectivity models with only a single edge between each pair of states, or large portions of the state space hidden behind bottleneck structures, can cause a swarm model checker to significantly slow down in terms of unique states visited per verification test. In this thesis, we provide an implementation of a Grapple model checker, a method for estimating unique states visited, an extension to the default Grapple search strategy called *start overs* that can reach deeper states, and a visualization technique for breadth-first search frontiers. Our experimental results show that our start over strategy can achieve a significant increase in state space coverage. Our experiments on low-connectivity models shows promising characteristics that can potentially be used to identify low-connectivity models.

Contents

List of Algorithms

List of Figures

List of Tables

1 Introduction

Swarm verification is an explicit-state model checking approach that tackles the state explosion problem by splitting the state space exploration into many small, independent verification tests (VTs), each covering only a subset of the state space. As VTs are small and independent, they can be parallelized on the GPU, resulting in high-speed model checking that can outperform the execution time of similar CPU-based approaches by multiple degrees. Past work has shown that low-connectivity models with only few edges between states, or large portions of the state space hidden behind bottleneck structures, can cause a swarm model checker to achieve significantly less state space coverage.

In this thesis, we provide the following contributions to swarm model checking on the GPU and the exploration of low-connectivity models:

- A method for estimating unique states visited with a fixed error in a swarm verification. Having the unique states visited, we can observe a swarm's progress in state space coverage.
- A Grapple search strategy extension called *start overs* that can reach deeper states by continuing the search within a VT multiple times. Furthermore, the strategy allows using even smaller hash tables by compensating lost hash table capacity with additional start overs.
- A visualization of unvisited and visited states in the frontiers of Grapple's breadth-first search that shows promising characteristics on a set of low-connectivity models.
- An in-depth explanation of the Grapple state space exploration loop by breaking it down into subroutines.
- An experimental implementation of a Grapple model checker, including a selection of models specifically implemented for on-the-fly verification on the GPU.

Comparison with the swarm verification waypoints benchmark has shown that our unique states visited estimation provides reliable results. We compared the progress in state space coverage of our start over strategy on multiple models, including the waypoints benchmark and the well-known dining philosophers problem. On those models, our start over strategy has shown a significant increase in state space coverage on a similar number of visited states and VTs in comparison to the default Grapple algorithm.

A notable exception has been the Anderson queue lock mutual exclusion algorithm. Using our start over strategy, state space coverage has worsened significantly. Furthermore, the model has shown interesting behavior in the visualization of its BFS frontiers.

The rest of this thesis is structured as follows: We start with an overview of related work in Chapter 2. A brief summary on model checking, swarm verification and the CUDA GPU programming framework is given in Chapter 3. The Grapple algorithm's components, swarm diversification techniques, including our start over strategy, and the unique states visited estimation are explained in Chapter 4. Our concrete implementation is presented

in Chapter 5. An extensive series of experiments comparing our implementation with the Grapple paper, and experiments comparing low-connectivity models with the waypoints benchmark model, is conducted in Chapter 6. Finally, we conclude our results and give an outlook to additional interesting research problems arisen by this thesis in Chapter 7.

2 Related Work

In this thesis, we are implementing a parallelized swarm verification model checker on the GPU using CUDA to conduct a series of experiments on low-connectivity models.

Swarm Verification was first studied by Holzmann et al. as verification method for models with large state spaces on which an exhaustive verification cannot be completed in any reasonable amount of time due to the state explosion problem [16]. It was first used as an extension to the SPIN model checker. Techniques for implementing swarm verification were introduced in [17], including a memory-efficient method of marking states as visited called *bitstate hashing*, multiple state space diversification techniques, and an evaluation of swarm verification performance on real-world models.

This thesis is based on the Grapple model checking framework [10]. They implement an explicit-state [13], on-the-fly, parallel swarm verification model checker on the GPU using the CUDA framework [9]. Our implementation tries to stay as close as possible to theirs, with three noteworthy exceptions:

- Our hash tables are smaller, as our implementation of bitstate hashing cannot address the full 48 KiB of shared memory due to a simplified calculation of bucket and element using bit shifts instead of arithmetic operations.
- We use a probabilistic cardinality estimator, so we can estimate unique states visited on models with billions of states. The Grapple paper is using a global hash table for counting, which would quickly exceed memory on large models.
- To go deeper in a state space exploration, we extend the algorithm by a self-contained strategy named *start overs* that is applied directly on-device within the GPU kernel.

A key difference between Grapple and the SPIN swarm verification is the highly constrained hash table size in Grapple, which stores its hash tables in CUDA's 48 KiB of shared memory. In the SPIN implementation, the hash table size is decided per model, often multiple megabytes in size. An evaluation of using small hash tables is done on an implementation of swarm verification using FPGAs, from which Grapple also adapts its structure of internal components within VTs [7]. A key observation of said evaluation is that in the SPIN implementation, the waypoints benchmark model is explored fastest using 256 MB hash tables, which is considerably larger than in the FPGA and GPU swarms. Even though its smaller hash tables, both the FPGA and GPU swarm outperform the SPIN swarm in execution time on the waypoints model, underlining the performance benefit of massive parallelization of swarm verification.

Exhaustive multi-core parallel model checking is a research area with numerous approaches, including parallelization of the SPIN model checker [15] and the parallel DiVinE model checker [3, 4]. Both approaches share using a parallel breadth-first search (BFS) and support verification of liveness properties. However, the SPIN approach only supports a sub-class of liveness properties that can be reduced into safety properties called *bounded*

liveness [14]. The Grapple framework is re-using the parallel BFS from the SPIN-approach for parallelization within VTs, which we are explaining in-depth in Section 4.2.2 and Section 5.1.2.

The foundations of the Grapple framework originate from an extension to the SPIN model checker that runs its parallel BFS state space exploration [14] on the GPU using CUDA [6]. Most notably, the extension is already generating successors on-the-fly on the GPU. For that, it introduces an algorithm that allows successors to be generated without branching through `if`-/`else`-/`switch`-conditions, which we are explaining in-depth in Section 4.1 and Section 5.1.3. The leading differences to the Grapple framework are that they perform an exhaustive verification and that synchronization of threads is done using a custom *fast-barrier synchronization* instead of using native CUDA `__syncthreads()`.

Another approach to model checking on the GPU is performed by DiVinE-CUDA [2]. Their work is focussed on finding accepting cycles, and thus verification of liveness properties, by reformulating the maximal accepting predecessors (MAP) algorithm to be conducted as matrix-vector multiplication on the GPU [5]. Even though the approach supports on-the-fly verification using similar operations for successor generation and violation checking as the Grapple framework, it cannot simply be added to a swarm verification. In swarm verification, no explicit state graph is built up during verification, or in other words, no explicit history of already visited states is kept. Having no explicit history of states introduces a severe difficulty on verifying whole paths for liveness properties.

Probabilistic cardinality estimation is an extensively covered research area due to its wide range of applications in internet-based applications. A comparison of available algorithms can be found in [19]. We chose to use the HyperLogLog++ (HLL) algorithm, which improves the HyperLogLog algorithm [11] for very large and very small cardinalities, and allows counting beyond a billion, which is necessary to, for example, count unique states visited of the waypoints model. HLL is a distributed, highly memory-efficient probabilistic cardinality estimator with a fixed error [12]. It is especially well-suited for counting unique states visited in swarm verification for multiple reasons: Most importantly, each HLL takes up only a few kilobytes to count into the billions, meaning that it can be stored in CUDA shared memory. HLLs can be merged, so independence of VTs is preserved. Due to `add` and `estimate` operations being separate from each other, we can use only the low-complexity `add` operation on the GPU and defer the estimation to the host, so no significant computational overhead is added to the state space exploration. Having a fixed error, it creates the possibility of giving probabilistic guarantees on the state space coverage of a swarm verification.

Looking beyond model checking, *graph partitioning* is an intuitively related problem to swarm verification. In graph partitioning, a graph is cut into mutual exclusive groups, so, for example, distributed algorithms can be run on them. An interesting approach in terms of on-the-fly verification are the streaming graph partitioning algorithms [21], which receive edges or vertices of a graph in a random order as inputs and produce partitions on them using heuristics. A main difference to diversification in swarm verification is that states are distributed onto their partitions as they arrive, which, in a naive implementation, would require to search for violations as separate step after partitioning.

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. Finally, 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 its 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. Having an exponentially growing number of states, it is even for small models 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. [8, 13]

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) [14].

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 [16]. 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.

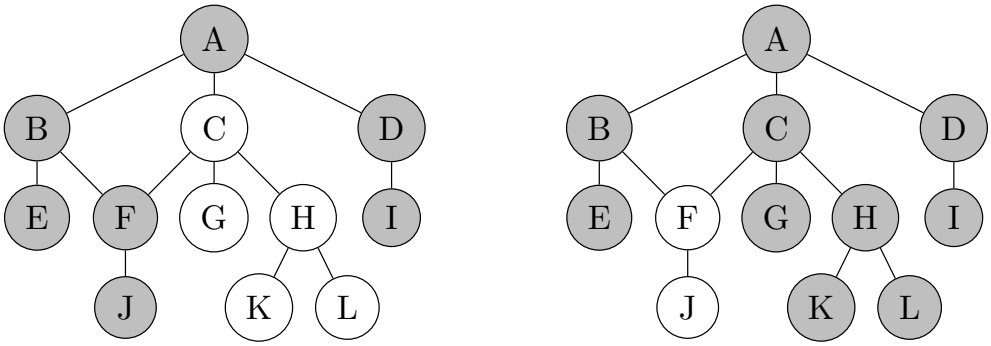


Figure 3.1: State pruning using hash collisions [10]

3.2.2 Grapple Framework

The paper "Swarm model checking on the GPU" by DeFrancisco et al. [10] 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

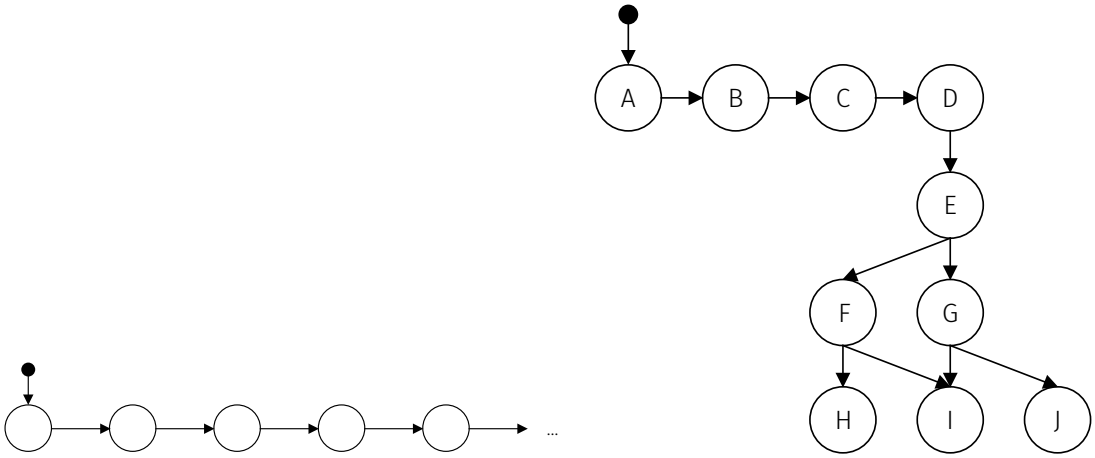
A key observation of [10] is that their algorithm's state space coverage in relation to the number of executed VTs slows down on models with *low connectivity*. A model has *low connectivity* if at least one of the following properties is satisfied:

- **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

Figure 3.2 shows examples for both properties. On the left, an example of a generally linear state space is presented. Clearly, every state only has one inbound and one outbound edge. The example on the right side shows a state space with a bottleneck introduced by state A–E that need to be passed in order to reach state F–J.

3.3 CUDA

CUDA is NVIDIA's proprietary GPU programming framework that provides automatic massive scalability [9]. 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



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

Figure 3.2: Examples of low-connectivity properties

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 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 KiB 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 KiB 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 requirement is a good fit with swarm verification, which turns the model checking problem into many small, independent VTs.

4 Concepts

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 of Algorithm 2 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 [10, Algorithm 1] using subroutines. We use subroutines to separately reason about queue operations, successor generation and marking states as visited. We abbreviate the Grapple state space exploration loop as *Grapple algorithm*. The Grapple algorithm is based on Holzmann's parallel BFS [14].

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 define the single operation of the hash table in Section 4.2.3. Finally, 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

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()
```

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

Next, we 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) fixed-capacity 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. In Grapple, the default capacity of a queue is $I = 4$.

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. Then, 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: [17, 10]

- 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.
- 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 introduce our *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 [17] and used as primary benchmark in [10]. 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.

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

Algorithm 3 Waypoints Model

```
P ← 8
NDC ← 4
violations ← {100 random 32-bit integers}

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

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

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 [12]. 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 [11].

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 [10], 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 that implement the parallel state space exploration loop, and collecting their results.
- 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.
- The output buffer, transferring discovered violations from the CUDA kernels back to the host.

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 random values for each CUDA thread. Using the random values, each VT can create different hash collisions, as explained in Section 4.2.3. Then, for each run, the main program executes the CUDA kernel in a grid of $K = 250$ blocks, each consisting of $N = 32$ threads. The CUDA kernel executes the state space exploration loop, as described in Section 4.2. 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 from all VTs, accumulates them and prints them to the standard output line-by-line as CSV.

5.1.2 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. To store the queues in global memory, we extend the amount of the queues by the amount K of VTs in a CUDA kernel, resulting in $K \times 2 \times N \times N$ queues in total. The multidimensional structure of the queues 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. In 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.

5.1.3 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.

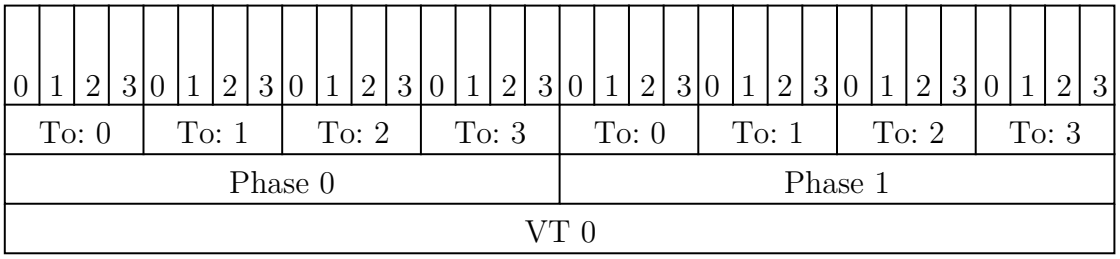


Figure 5.1: A single VT of the 4D array, mapped to 1D memory

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. 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 [6, Algorithm 3]. To do so, they make use of the custom ternary operator described in Algorithm 4 that exploits the fact that boolean values evaluate to either zero or one.

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 BEEM model database [20]. 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.

Algorithm 4 Branching-Free Ternary Operator

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

5.1.4 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 [10]. 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. To

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
```

achieve optimal utilization of memory transfers, we use a bucket size of 32 bit. The bit representing whether a state is visited is then computed using Algorithm 6.

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 [18]. The MurmurHash3 [1], 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.

5.1.5 HyperLogLog++

The HyperLogLog++ (HLL) provides an algorithm for distributed counting of distinct elements. Other than described in Section 4.5, in our implementation, each CUDA grid, consisting of multiple VTs, is sharing a single HLL that is stored in global memory, so we only have to copy a single data structure from device to host. Visited states are added to the HLL only on-device. The host collects the HLLs after the kernels have finished, merges 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.

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
```

5.1.6 Output Buffer

Each VT can discover an unpredictable amount of violations. In the worst case, each state visited by a VT violates. To transfer an unknown amount of violating states from device to host, so they can be reported to the user, we use a fixed-capacity FIFO buffer that drops new entries when full. The buffer is stored in global memory and shared between all VTs executed in a CUDA kernel.

The FIFO buffer is implemented using a fixed-capacity array that holds the elements, and two integers with indices to the buffer's `head` and `tail`. Other than the queues described in Section 5.1.2, we use array indices instead of pointers for `head` and `tail`, so the buffer can be copied from device to host without rewriting pointers from device to host memory. On `push`, the violating state is written into the elements array at the index of the `tail`, which is then incremented by one, until the buffer is full. The buffer can be considered full when `tail` equals the capacity. On `pop`, a pointer to the violating state at the position of the `head` is returned, and the `head` gets incremented by one, until the buffer is empty. The buffer can be considered empty when `head` equals `tail`.

This design of the buffer requires it to be used in two phases that cannot be mixed: In the first phase within the CUDA kernel, the buffer is filled with elements using the `push` operation. In the second phase after the CUDA kernel is finished, the buffer is copied to the host and emptied there using the `pop` operation. To reuse a buffer, we have to `memset` its memory to 0.

5.2 Usage

This section provides a user guide for our Grapple model checker. We start with a description of the workflow from model specification to output interpretation in

Section 5.2.1. We continue with the build configuration in Section 5.2.2 and a list of command line options that can modify the runtime behavior of the model checker in Section 5.2.3. Finally, we explain how to interpret the model checker's output in Section 5.2.4.

5.2.1 Model Checking Workflow

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

1. Describe their model in a C++ class using Algorithm 5,
2. Configure the build to include their model using the text macros explained in Section 5.2.2.
3. Compile the model checker using CMake.
4. Execute the compiled binary, with optional command line options specified in Section 5.2.3.
5. Interpret the CSV output, as explained in Section 5.2.4.

5.2.2 Build Configuration

In our implementation, the parameters of the Grapple algorithm and the verified model can be changed by overriding text macros. Text macros can be overridden by supplying command line options to the Nvidia CUDA Compiler `nvcc`, for example `-DGRAPPLE_MODEL=PhilosophersStateV2`. The following macros, each listed with purpose and default value, are available:

- `GRAPPLE_VTS`
  The number of VTs in a CUDA grid. Default: 250.
- `GRAPPLE_N`
  The number of threads in a VT. Default: 32.
- `GRAPPLE_I`
  The number of queue slots. Default: 4.
- `GRAPPLE_SO`
  The number of start overs. Default: 0.
- `GRAPPLE_HT`
  The hash table capacity, as a power of two. Default: 18.
- `GRAPPLE_HLL`
  The number of the HyperLogLog++ registers, as a power of two. Default: 14.
- `GRAPPLE_MODEL`
  The model that is going to be verified. Default: `WaypointsState`.

- `GRAPPLE_INSPECT_BFS`
  This macro enables the implementation of the experiment explained in Section 6.2.3. When set, the visited and failed states of a single VT's BFS frontiers are returned instead of model progress and violations. By activating this macro, `GRAPPLE_VTS` is automatically set to 1. Default: false.

5.2.3 Command Line Options

The compiled binary provides the following command line options to modify its runtime behavior:

- `-s <seed>`
  Specifies the seed of the PRNG that is used to generate all random numbers used in state space diversification. By specifying this option, reproducible results can be achieved. Default: Use a random seed.

- `-n <runs>`
  Specifies the amount of Grapple runs, where each run consists of `GRAPPLE_VTS` VTs. The amount of runs needed to achieve a certain state space coverage needs to be evaluated experimentally. Default: 200.

- `-q`
  When specified, no violations are printed out. This option can be used to only observe the state space coverage on a model, which is useful when otherwise too many violations are found. Default: Print all violations found.

5.2.4 Output Interpretation

During execution of the binary, the model checker continuously reports its progress line by line to the standard output. After each run, an output line containing the progress and an additional output line for every violation found in the run are created. The output is compatible with parsers for the comma-separated values (CSV) file format. Each output line contains the following values, separated by commas:

1. `run`
   The zero-based index of the run that created this output line.

2. `block`
   Only on violations: The CUDA block that discovered the violation.

3. `thread`
   Only on violations: The CUDA thread that discovered the violation.

4. `state`
   Only on violations: The text representation of the violating state.

5. `uniques`
   The amount of unique violations discovered across all VTs.

6. `visited`
   The amount of unique states visited across all VTs.

7. `visited_percent`
   The state space coverage, as floating point value in percent. Only available on models with known total state space size.

8. `vts`
   The amount of VTs executed until creation of the output line.

9. `total_visited`
   The amount of total states visited across all VTs.

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 results from the Grapple paper [10] 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. The seed of the PRNG in each experiment is `-s 1736331306`. 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 potential reasons for the weak performance of our implementation, we conducted additional experiments.

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

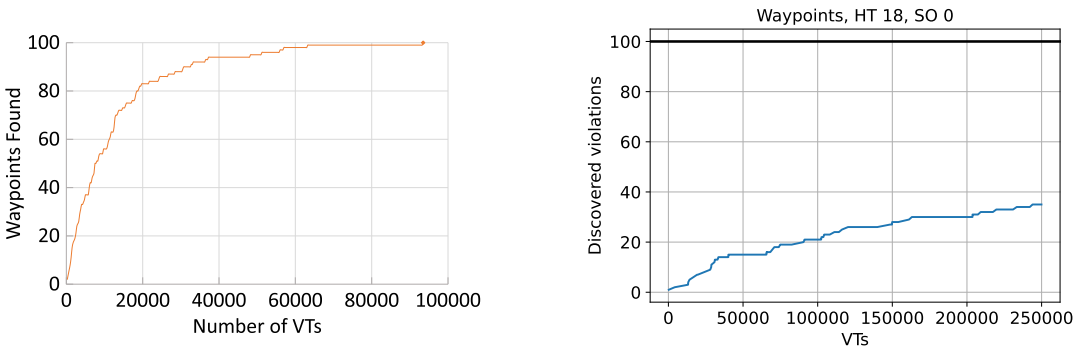
As explained in Section 4.4, counting waypoints is a method for estimating the achieved state space coverage. To make sure that the discovered waypoints of our implementation correctly indicate the state space coverage, we compared them to the state space coverage estimation of the HyperLogLog introduced 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. This is due to our bit shift implementation presented in Section 5.1.4 which cannot address the full 48 KiB of shared memory.



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

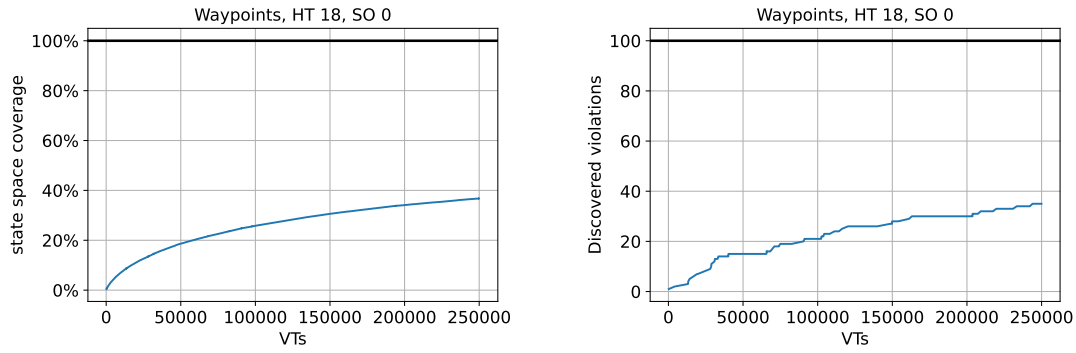
Figure 6.1: Waypoints model, comparison between reference and our implementation

- 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



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

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

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

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.

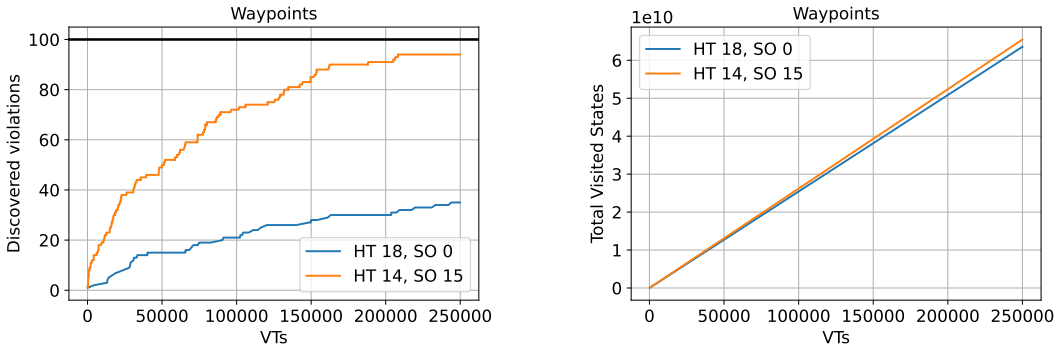
6.1.3 Increasing State Space Coverage using Start Overs

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.

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



(a) Comparison of discovered waypoints

(b) Comparison of total visited states

Figure 6.3: State space exploration with start overs

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

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.

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 %

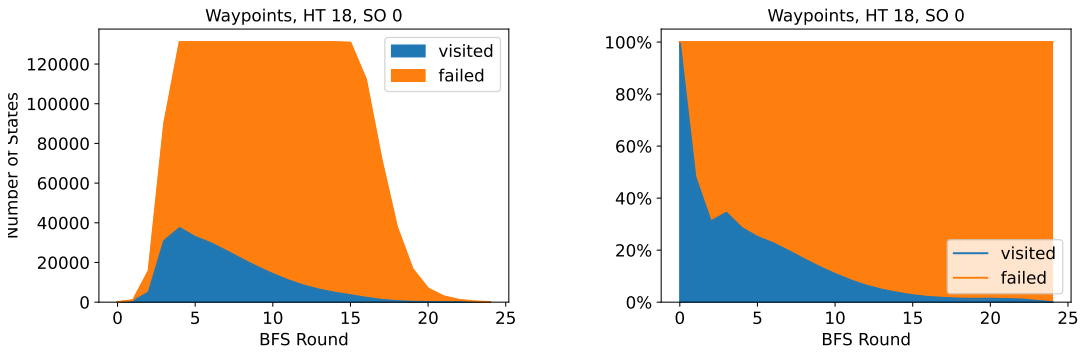


Figure 6.4: BFS frontier visualization of the waypoints model

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.

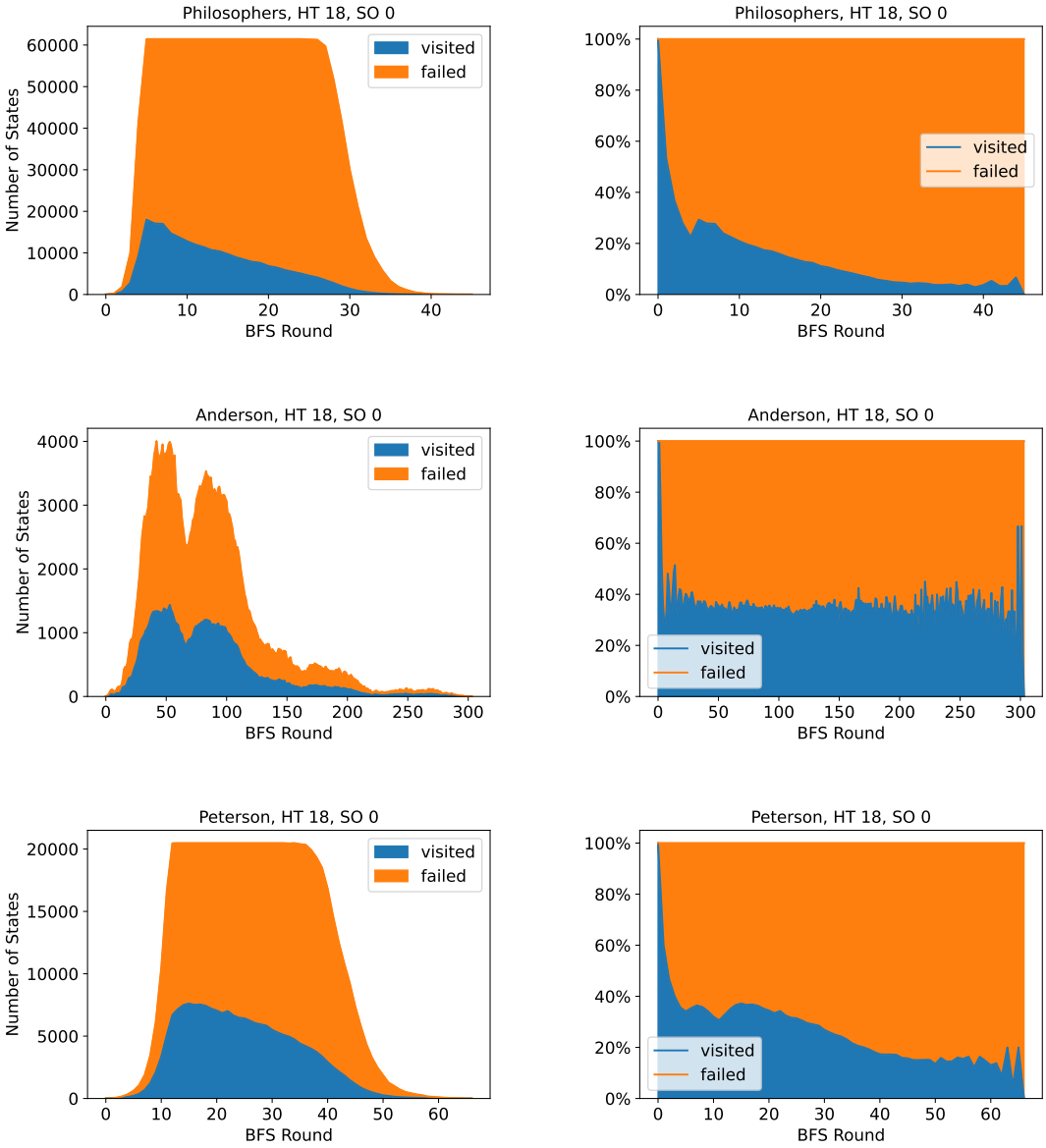


Figure 6.5: BFS frontier visualization of low-connectivity models

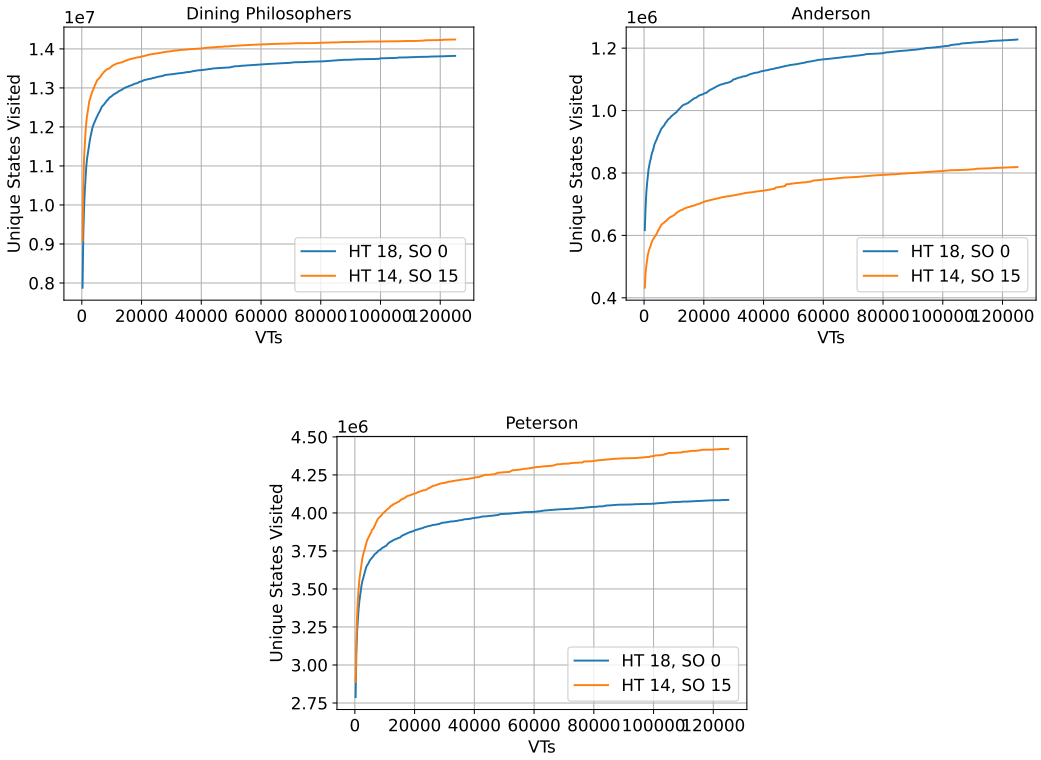


Figure 6.6: Start over strategy on low-connectivity models

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 Conclusions

The final chapter of this thesis starts with a discussion of our work in Section 7.1 and finishes with an outlook for future work on swarm verification in Section 7.2.

7.1 Discussion

We successfully implemented a Grapple swarm model checker, as presented in Chapter 5, and extended its state space exploration with a promising strategy called *start overs* to reach deeper states in Section 4.3.2. Our experiments in Section 6.1.3 and Section 6.2.4 have shown that a deeper state space exploration can result in an improved state space coverage on a similar number of total states visited.

We introduced and implemented a method for estimating unique states visited in a swarm verification with a fixed error in Section 4.5. Estimating unique states visited allows monitoring the progress in state space coverage of a swarm verification, even if due to the state explosion problem exact counting is impossible, as shown in Section 6.1.1.

Our experiments on low-connectivity models in Section 6.2 have shown promising characteristics that potentially could be used to identify low-connectivity models. The most promising experiment has been the visualization of BFS frontiers in Section 6.2.3, as it allowed comparing the internal progress in a VT between models.

The Anderson model has been a notable exception in our experiments on low-connectivity models: Its exploration resulted in the lowest total states visited among all analyzed models, as shown in Section 6.2.2. The visualization of its BFS frontiers in Section 6.2.3 contradicted our intuition by having a near constant amount of new states visited throughout its BFS frontiers. And finally, the start over strategy, in contrast to all other models in our experiments, performed worse on the Anderson model, as shown in Section 6.2.4.

Being in line with past work, this thesis has shown that swarm verification on the GPU is a promising approach for verification of very large state spaces in a short time using widely available, affordable consumer hardware.

The source code of our implementation of a Grapple swarm model checker and the corresponding experiment series are freely available on GitHub[1], allowing future work on swarm verification to continue on our efforts.

7.2 Future Work

While working on this thesis, multiple ideas for future work on swarm verification came up. We present two ideas continuing the work on estimation of state space coverage, an idea that could further increase the performance benefit of start overs, and a small idea for comparing state space diversification techniques.

[1] `https://github.com/barnslig/bachelor-thesis`

Estimate the state space size of models with unknown size To calculate the coverage of a swarm, both the total state space size and the unique states visited are required. The unique states visited can be estimated using our approach presented in Section 4.5. The total state space size however is often an unknown figure, unlike for some models used in our evaluation, for which the absolute size is known. Following that the total state space size is often unknown, finding a method of estimating it is a beneficial extension to swarm verification, as it would allow to estimate the swarm coverage on arbitrary models.

Detecting when exhaustive coverage is achieved Having an estimation of unique states visited and the total state space size of a model, we can estimate the progress in state space coverage of a swarm verification. Intuitively, knowing the progress in state space coverage would allow termination of the swarm when 100 % state space coverage is achieved. However, due to the inaccuracy of estimating unique states visited, we cannot simply terminate the search the first time the estimation achieves 100 %, without possibly leaving a small portion of the state space unvisited. An interesting future contribution could be thus a strategy of terminating the swarm when full state space coverage is achieved with, for example, 99 % probability.

Determining the optimal number of start overs To keep the state space exploration terminating, our start over strategy that we introduced in Section 4.3.2 requires providing the number of start overs. In Section 6.1.3, we only evaluated the strategy using a fixed amount of start overs to mitigate smaller hash tables. Looking further, finding an amount of start overs for a model that provides a good trade off between a faster state space coverage growth and enlarged VTs, which would worsen parallelization and execution time, is an interesting problem for future contributions.

Analyzing state space diversification In an ideal swarm verification, there would be no overlap between VTs, resulting in optimal parallelization of the verification. Due to the probabilistic approach of diversification techniques introduced in Section 4.3.1, finding a metric for state space overlap between VTs could be an interesting opportunity to compare diversification techniques.

Bibliography

[1] Austin Appleby. *MurmurHash3*. URL: https://github.com/aappleby/smhasher (visited on 08/31/2021).

[2] Jiří Barnat, Luboš Brim, and Milan Češka. "DiVinE-CUDA - A Tool for GPU Accelerated LTL Model Checking". In: *Proceedings 8th International Workshop on Parallel and Distributed Methods in verifiCation*. Vol. 14. EPTCS. Nov. 4, 2009, pp. 107–111. DOI: 10.4204/EPTCS.14.8.

[3] Jiří Barnat, Luboš Brim, and Petr Ročkai. "DiVinE Multi-Core — A Parallel LTL Model-Checker". In: *Automated Technology for Verification and Analysis*. Vol. 5311. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 234–239. DOI: 10.1007/978-3-540-88387-6_20.

[4] Jiří Barnat, Luboš Brim, and Petr Ročkai. "Scalable Multi-core LTL Model-Checking". In: *Model Checking Software*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 187–203. ISBN: 978-3-540-73370-6. DOI: 10.1007/978-3-540-73370-6_13.

[5] Jiří Barnat et al. "CUDA Accelerated LTL Model Checking". In: Shenzhen, China: IEEE, Dec. 8–11, 2009, pp. 34–41. ISBN: 978-0-7695-3900-3. DOI: 10.1109/ICPADS.2009.50.

[6] 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.

[7] Shenghsun Cho, Michael Ferdman, and Peter Milder. "FPGASwarm: High Throughput Model Checking on FPGAs". In: Dublin, Ireland: IEEE, Aug. 27–31, 2018, pp. 435–4357. ISBN: 978-1-5386-8518-1. DOI: 10.1109/FPL.2018.00080.

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

[9] *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 09/23/2021).

[10] 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.

[11] 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.

[12] 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.

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

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

[15] Gerard J. Holzmann. *The SPIN Model Checker — Primer and Reference Manual*. Addison-Wesley, 2004. ISBN: 978-0-321-22862-8.

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

[17] 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.

[18] Bob Jenkins. *A Hash Function for Hash Table Lookup*. URL: https://burtleburtle.net/bob/hash/doobs.html (visited on 07/26/2021).

[19] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. "Why Go Logarithmic If We Can Go Linear? Towards Effective Distinct Counting of Search Traffic". In: *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '08. Nantes, France: Association for Computing Machinery, 2008, pp. 618–629. ISBN: 9781595939265. DOI: 10.1145/1353343.1353418.

[20] ParaDiSe. *BEEM: BEnchmarks for Explicit Model Checkers*. Discontinued. URL: https://paradise.fi.muni.cz/beem/ (visited on 06/11/2021).

[21] Isabelle Stanton and Gabriel Kliot. "Streaming graph partitioning for large distributed graphs". In: *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. KDD '12. Beijing, China: Association for Computing Machinery, Aug. 12, 2012, pp. 1222–1230. ISBN: 9781450314626. DOI: 10.1145/2339530.2339722.