TUHH

Hamburg University of Technology

Leonard Techel

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

September 7, 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

TUHH

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

Bibliography   **21**

List of Figures

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.

2 Related Work

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 interested in. Then, we give an overview of Swarm Verification on the GPU and the specific problem that we are going to investigate: Low-Connectivity State Space Exploration. Last but not least, we give an introduction to the CUDA GPU programming framework.

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 lead 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]

3.1.1 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 the parallel breadth-first search (BFS) from the paper "Parallelizing the Spin Model Checker" by Holzmann [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

3.1.2 Swarm Verification

A new approach on parallelized model checking comes from the paper "Swarm Verification" by Holzmann et al. [8]. Swarm Verification 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 most powerful 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.
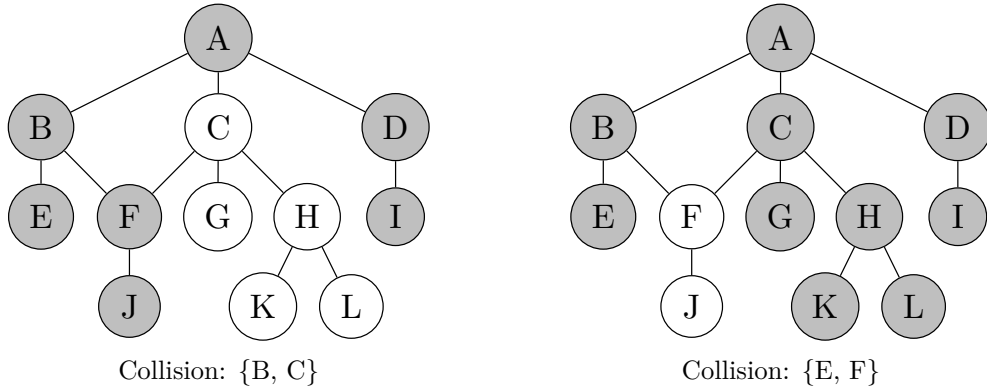
Collision: {B, C}   Collision: {E, F}

Figure 3.1: State pruning using hash collisions

3.1.3 Grapple Framework

The paper "Swarm model checking on the GPU" by DeFrancisco et al. [3], on which this thesis is based, 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.

3.1.4 Low-Connectivity Models

A key observation of [3] is that their algorithms'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.

3.2 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
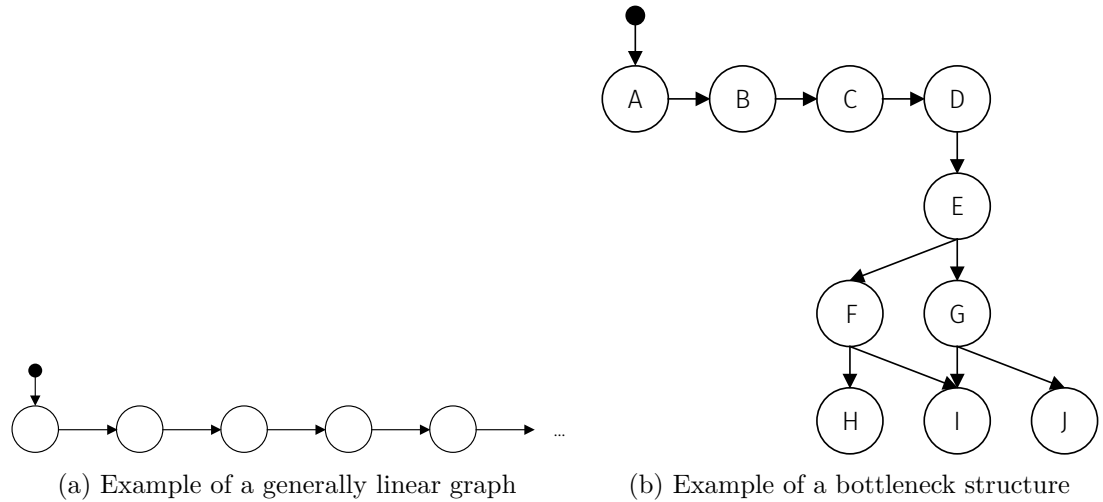
Figure 3.2: Examples of low-connectivity properties

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 ...

4.1 Grapple model checker

The state space exploration of each VT is described by Algorithm 1 in [3]. It is based on the parallel BFS from "Parallelizing the Spin Model Checker" by Holzmann [7]. In summary, this parallel BFS algorithm works by allowing lock-free communication between threads using a shared, multidimensional queue array where, for each pair of threads, there is only one piece of memory to communicate over. Writing and reading to this piece of memory is coordinated by splitting the algorithm into two alternating phases.

Each thread in a VT executes Algorithm 2. N is the amount of threads within a block, with $N = 32$ being the default. I is the amount of queue slots, with $I = 4$ being the default.

4.2 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.

The model is defined using a *Kripke Structure* $M = (S; S_0; R; L)$ where S is the finite set of states, $S_0 \in S$ is the finite set of initial states $S_0 \subseteq S$, $R \subseteq S \times S$ is the left-total transition relation and $L : S \mapsto 2^{AP}$ is the labelling function that assigns each state its atomic propositions.

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

The states of a model support two operations: state_successor($p, ndc, state$) on-the-fly returns a successor to a state and state_violates($state$) returns whether a state violates.

4.3 Queues

The queues support 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.

4.4 Hash table

The hash tables support one operation: mark_visited($V, state$) marks a state as visited and returns whether it was already visited before.

Algorithm 2 Grapple state space exploration loop of a single worker

---

$t \in \{0, 1\} \leftarrow 0$ ▷ *Current algorithm phase*
$visited \subseteq S \leftarrow \emptyset$ ▷ *Hash table of visited states*
$Q[2][N][N][I] \leftarrow \emptyset$ ▷ *Queues*

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

`__syncthreads()`

$done \leftarrow$ false
while not $done$ **do**
$\quad$ **for** $i = 0, \ldots, N$ **do**
$\quad\quad$ **while** not queue_empty$(Q[t][\text{threadIdx.x}][i])$ **do**
$\quad\quad\quad$ $state \leftarrow$ queue_pop$(Q[t][\text{threadIdx.x}][i])$
$\quad\quad\quad$ **for** $p \in$ processes **do**
$\quad\quad\quad\quad$ **for** $ndc \in$ nondeterministic choices within p **do**
$\quad\quad\quad\quad\quad$ $succ \leftarrow$ state_successor$(p, ndc, state)$
$\quad\quad\quad\quad\quad$ $visited \leftarrow$ mark_visited$(succ)$
$\quad\quad\quad\quad\quad$ **if** not $visited$ **then**
$\quad\quad\quad\quad\quad\quad$ **if** state_violates$(succ)$ **then**
$\quad\quad\quad\quad\quad\quad\quad$ report path to state
$\quad\quad\quad\quad\quad\quad$ **else**
$\quad\quad\quad\quad\quad\quad\quad$ $next \leftarrow$ random output queue $n \in N$
$\quad\quad\quad\quad\quad\quad\quad$ queue_push$(Q[1-t][next][\text{threadIdx.x}], succ)$

$\quad$ `__syncthreads()`

$\quad$ $done \leftarrow$ queue_empty$(Q[1-t][0 \ldots N][\text{threadIdx.x}])$
$\quad$ $t \leftarrow 1 - t$

$\quad$ `__syncthreads()`

4.5 Validity, Correctness, Completeness, Termination

4.6 State Space Diversification

4.6.1 Diversification Techniques

These diversification techniques exist:

1. State Pruning using Hash Collisions
   Each VT marks its visited states in a limited-size hash table with varying hash functions. 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.

2. Reversing search direction or order

3. Randomizing order of nondeterministic choice

4. Parallel Deep Search (PDS)

5. process-PDS

4.6.2 State Pruning and Start Overs

4.7 Counting Unique States Visited

In Swarm Verification, we achieve a much faster verification by only covering *nearly* 100% of the state space. Resulting from this, it is important to know how much of the state space is actually covered. To find this out, we need to count unique states visited.

To calculate the percentage of state space covered, we have to know the total state space size. As in this thesis we only consider models with known state space size, we can provide it as pre-calculated constant.

The 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 algorithm introduced in "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm" by Flajolet et al. [4], as it is commonly used and relatively easy to implement.

For each VT, we create a HyperLogLog. Inside each VTs state space exploration, we call the `add` operation for each newly discovered state. When a VT has finished, we `merge` its HyperLogLog with those of all other already finished VTs. We then can execute the `estimate` operation on the global HyperLogLog to get the estimation of unique visited states across all finished VTs.

Cardinality estimation with HyperLogLog yields good results on the Dining Philosophers problem with 15 processes and $3^{15} - 1$ states. As expected, the number of unique states visited grows logarithmic until reaching around 100%. However, on the Waypoints model with 2^{32} states, the estimation fails.

The paper "HyperLogLog in Practice: Algorithmic Engineering of a State of the Art Cardinality Estimation Algorithm" by Heule et al. [5] finds out that the estimation of cardinalities beyond one billion fails on the original HyperLogLog algorithm. They solve this problem by using a 64-bit hash instead, calling their improved algorithm HLL++.

Using HLL++, we can count unique states visited.

5 Implementation

Our implementation serves two purposes: First, to reproduce the results from [3]. Second, as foundation for experimenting with 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: 1. The main program, running the CUDA kernels and collecting their results. 2. The CUDA kernel, implementing a parallel state space exploration loop. 3. The queues, providing lock-free communication between threads in a VT. 4. The hash tables in which states are marked as visited. 5. The model, providing successor generation and violation checking. 6. 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 4.1.

In summary, it works as follows: First, it retrieves new input states from its queues. Then, for each input state, the successors are generated. For each successor, we check whether it is already visited in the 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's output queue. This process repeats until no more unvisited successors are discovered.

5.1.3 Queues

The queues provide a data structure for lock-free communication between threads in a VT, as described in Section 4.3. 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 a null pointer.

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 VT with $N = 4$. It aims to make it more clear why we have to multiply each variable with the *width* of all children, doing the same on every level of the multidimensional array, then adding up all products.

0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
To: 0				To: 1				To: 2				To: 3				To: 0				To: 1				To: 2				To: 3			
Phase 0																Phase 1															
VT 0																															

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

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.

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

Algorithm 3 Branching-Free Ternary Operator

---

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

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. Other than in [1], 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.

We translate DVE models into CUDA compatible C++ with the template described in Algorithm 4.

Algorithm 4 On-The-Fly State Generation on the GPU

    ▷ *Global variables* ◁
    $glob \leftarrow 0$
    ▷ *Process-Local variables use arrays. Here, N is the number of processes* ◁
    $state[N] \leftarrow \{0, \ldots, 0\}$
    function state_successor(p, ndc, state)
        ▷ *Evaluate all guards before calculating transitions* ◁
        $guard_1 \leftarrow state[p] = 0$
        $guard_2 \leftarrow state[p] = 1 \wedge glob \leq 2$
        $guard_3 \leftarrow state[p] = 1 \wedge glob > 2$
        ▷ *Transition from state 0 to 1* ◁
        $next.state \leftarrow \text{MyTernary}(guard_1, 1, state.state)$
        ▷ *Transition from state 1 to 1* ◁
        $next.glob \leftarrow \text{MyTernary}(guard_2, state.glob + 1, state.glob)$
        $next.state \leftarrow \text{MyTernary}(guard_2, 1, next.state)$
        ▷ *Transition from state 1 to 0* ◁
        $next.glob \leftarrow \text{MyTernary}(guard_3, 0, next.glob)$
        $next.state \leftarrow \text{MyTernary}(guard_3, 0, next.state)$
        return $next$

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

5.1.5 Hash tables

The hash tables provide a data structure in which states are marked as visited.

Hash table - The data structure in which states are marked as visited - Collisions are used for diversification - Stored in shared memory, thus highly constrained size

5.1.6 HyperLogLog++

HyperLogLog - The data structure that counts unique states visited across all VTs - Each VT has a shared HyperLogLog for all of its threads - Threads only call the 'add' operation on their HLL - The host collects all HLLs, then merges them to calculate the total number of unique states visited

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 4

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 Experiments

6.1 Reproducing the results from [3]

6.2 Low-Connectivity Models

6.3 Optimizing Grapple on Low-Connectivity Models

7 Conclusion

7.1 Future work

7.2 Discussion

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.