Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Visualizing explorations in GenMC

Bachelor Thesis

Valon Lugic

December 24, 2025

Advisors: Prof. Dr. M. Kokologiannakis, M. Roshardt

Department of Computer Science, ETH Zürich

Abstract

GenMC is a stateless model checker for concurrent programs under weak memory models that uses the TruSt algorithm during the exploration process. Due to the implicit, graph-based tree structure of the exploration, the textual report produced by GenMC is unintuitive and hard to comprehend for users. In this thesis, we implement a visualization tool that visualizes the exploration tree induced by the TruSt algorithm in its natural, two dimensional graphical structure. The visualization tool logs the necessary information by injecting hooks in GenMC without changing the algorithm's behavior or correctness. We serialize the collected data into a compact textual log file that represents a set of individual execution graphs as blocks of text, together with their parent-child relationships in the tree-structured exploration. Based on that file, the visualization tool renders the exploration tree and introduces interactivity to the visualization, which improves the overall comprehensibility of GenMC's verification process for non-expert users.

Contents

Chapter 1

Introduction

The correctness of concurrent programs depends on the memory model under which they are executed. Weak memory models, which are implemented by modern architectures, allow for a wide range of possible behaviors that are prone to concurrency errors. Therefore, to detect such errors, an automatic verification tool that performs model checking of concurrent programs under weak memory models is used. Given a program as input, the verifier explores all possible behaviors of the program by checking all of its executions for memory inconsistencies or safety violations.

If such an error is present, a detailed error report and the corresponding faulty execution that triggered the error are returned by the verifier. The generated report is exclusively in textual format. But, by the inherently graphical nature of the underlying verification process that uses graph-based structures to reason about executions, representing this process through a linear, text-based report is unintuitive and difficult to follow, especially for inexperienced users who are not familiar with the verification algorithm of the tool or with weak memory semantics in general, which defies the use of the model checker.

To address this gap, we introduce a visualization tool to represent the verification process in its natural, two-dimensional graphical form. The tool aims to make error reports and execution behaviors more comprehensible, helping both users and researchers in understanding how and why a particular program is memory inconsistent, minimizing the difficulty introduced by solely textual explanations and improving the usability of the verification tool as a whole.

Given the practical nature of this project, the thesis is divided into two parts. The first part introduces the foundational material required to understand the rest of the thesis. The material covers model checking in theory and presents the TruSt verification algorithm, which is applied by the automatic

verification tool GenMC. The focus of this part is to connect the theory directly to the model checker GenMC, for which we implement a visualization tool for in the second part of this thesis. The second part focuses on the practical work, presenting the design and implementation of the visualization tool.

Initially, the visualizer was meant to be implemented as a fully independent component that operates alongside GenMC without modifying the core verification engine. Although possible in theory, complete independence from the model checker was not achieved in the scope of this thesis. The visualizer extracts additional data from GenMC's internal state.

Chapter 2

Model Checking

Before explaining the underlying verification algorithm of GenMC, chapter 2 introduces the foundation that the algorithm is built on. To be precise, it introduces model checking as a whole and explains how it applies to the verification of concurrent programs under weak memory semantics, as is the case with GenMC.

First, the chapter gives an overview of model checking in section 2.1. Then, section 2.2 introduces the modeling of concurrent programs as execution graphs, after which we present the memory consistency properties. Finally, in section 2.3 we state the executability requirements needed for GenMCs verifier to work in an automatic fashion.

2.1 Overview

The automated process of verification used to determine whether a system satisfies a set of specified properties, is called model checking [3]. It exhaustively explores all possible behaviors of a system within a defined model and detects whenever some behavior doesn't satisfy the specified properties.

We divide the model checking into three parts

1. Modeling of the given system using an abstraction model
2. Defining the properties that the system is required to satisfy
3. Exhaustive exploration of the models possible behaviors to check whether defined properties hold or are violated

In the context of the thesis, the system we want to prove correct is a concurrent program. The abstraction model used is an execution graph that abstracts the program into containing only relevant memory operations that define the program's memory-related behavior under a weak memory model.

The properties we want to verify for correspond to the consistency requirements of said memory model

2.2 Modeling

An abstraction model [1] is a precise and abstract representation of a system. It simplifies the system into only its relevant operations to the verification by using semantics that leave out unnecessary details. In the case of concurrent programs, only the memory operations under weak memory semantics are relevant. All other parts of the program, like arithmetic operations or control-flow instructions, are irrelevant and left out, as we want to solely express weak memory behavior.

Concurrent programs written under a weak memory model allow for different reorderings and interleavings of the program. We use a graph-theoretical model for abstracting a given program. The model represents different executions as execution graphs [4] corresponding to all possible behaviors of the program under the given memory model. An execution graph consists of a set of events G.E (nodes) and a set of relations over those events (edges). As already mentioned, the set of all possible executions graphs, where each represents one possible behavior of the program, covers the whole state space of all possible reorderings and interleavings. The state space grows exponentially with the number of memory operations and threads. The model checker verifies a given program by exhaustively checking all states individually, with the defined properties that the program needs to fulfill.

2.2.1 Events

The nodes of the execution graph represent memory-related events performed by the program. To be precise, they are memory operations written with weak memory model semantics. An event $e \in \mathit{Event}$ is either the initialization event *init* or a tuple $\langle t, i, \mathit{lab} \rangle$, where $t \in \mathit{Tid}$ is a thread identifier, $i \in \mathbb{N}$ is a serial number local to the specified thread t, and $\mathit{lab} \in \mathit{Lab}$ is a label describing the kind of event the tuple represents.

The set of labels includes following types:

- **Write labels:** $W_k(l, v)$, where $k \in \mathit{Kind}$ is the kind of write, $l \in \mathit{Loc}$ is the accessed memory location, and $v \in \mathit{Val}$ is the value written.
- **Read labels:** $R_k(l)$, where $k \in \mathit{Kind}$ is the kind of read and $l \in \mathit{Loc}$ is the accessed memory location.
- **Fence labels:** F_k, where $k \in \mathit{Kind}$ is the kind of fence
- **Error labels:** *error*, represents a detected safety violation

The set of events $G.E$ is consists of following subsets (excluding fences):

$$G.W \triangleq \{init\} \cup \{\langle t, i, lab\rangle \mid lab = W_k(l, v)\},$$
$$G.R \triangleq \{\langle t, i, lab\rangle \mid lab = R_k(l)\},$$
$$G.Err \triangleq \{\langle t, i, lab\rangle \mid lab = error\}.$$

Where no two events share the same thread identifier or serial number.

2.2.2 Relations

An execution graph consists of a set of events G.E (nodes) and a set of relations over those events (edges), which are presented in this subsection.

Reads-from relation (rf). The reads-from relation is a function

$$G.rf : G.R \to G.W$$

mapping the read event to corresponding write event from which it reads its value. Both, the read and write event must access the same memory location.

Coherence order (co). The coherence order is a strict partial order

$$G.co \subseteq \bigcup_{l \in Loc} G.W_l \times G.W_l,$$

which is total over the writes to each individual memory location.

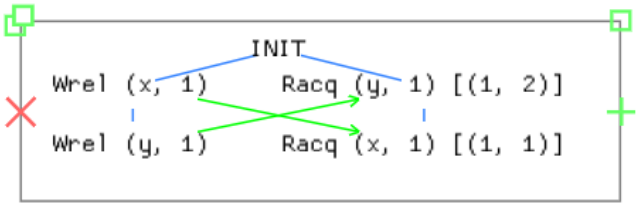
Program order (po). Program order is not defined explicitly. Instead, it is induced by representing the events as a partial order of all events from the same thread, based on their serial numbers. The initialization event is always before all non-initialization events.

These relations form the structural basis upon which memory consistency properties are defined, and which are described in more detail in Subsection 2.4.

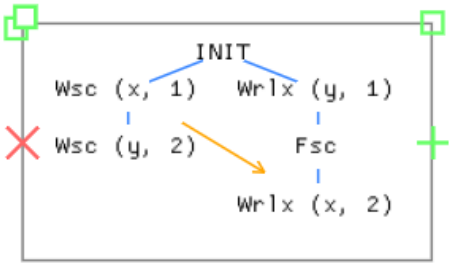
2.3 Example Execution Graphs

To gradually introduce the reader to the visualization tool implemented in this thesis, we present three example graphs in this section that complement the previous formal definitions on events and relations. Each graph corresponds to a single program execution. The examples are arbitrarily chosen and unrelated and are meant to simply showcase the visualization of how

events, relations and events are visualized. The left execution graph is meant to showcase a reads-from (rf) relation, which is drawn as a green directed edge. It connects a read event to the specified write event that it observes its value read from. The right execution graph is meant to show a coherence order (co) relation, which is drawn as an orange directed edge, where the two writes accessing the same memory location are connected to each other based on the coherence order. In the final exploration tree that visualizes the entire verification process of the recursive TruSt algorithm, each node of the exploration tree corresponds to a unique execution graph, like the three examples presented in this subsection. The overall visualization of the exploration tree is introduced later in chapter 3.



(a) Reads-from (green edge)



(b) Coherence order (orange edge)

Figure 2.1: Two example execution graphs visualized using the tool, which are independent and arbitrarily chosen. The serve only as graphical representation of individual nodes within the exploration tree.

2.4 Memory Consistency Predicates

After we've defined the model (execution graphs) with which we abstract concurrent programs into their compact form containing only memory-relevant information necessary for model checking in section 2.2, the next step is to define the properties that the modeled execution graph must satisfy, to fulfil the memory consistency requirements given by the memory model. These

properties are represented by the relations over the graph's events (e.g., program order, reads-from, coherence order, and derived relations).

GenMC's default memory model [5] is RC11 (Repaired C11) and supports the full range of C11 atomics operations and annotations like memory_order_relaxed, acquire, release, acq_rel, etc., which are then used as the different kinds of write, read, or fence event as defined in subsection 2.2.1. From now on, all illustrative examples of execution graphs or exploration trees shown in this thesis are from executions of concurrent programs written under the RC11 memory model.

This thesis does not require any further knowledge of any specific consistency predicate. The key takeaway of this section is that GenMC abstracts executions by the defined execution graph model, which are then checked for memory consistency via the consistency predicate.

2.5 Model Executability

After defining the abstraction model (execution graph model), the required properties for correctness of the execution graph under a given memory model, the final step is to guarantee that the execution graph model is executable. This property of the model is required, as it allows the verification tool to check the whole state space of the program exhaustively in an automated fashion.

The way that GenMC works, this means that execution graphs must be able to be constructed incrementally. After each construction step, we check inplace whether the consistency predicate holds true, giving us a memory-efficient way to verify memory consistency.

GenMC assumes that the underlying memory model is axiomatic. This means that memory consistency is solely defined by the given relations over events. As stated in section 1.3, GenMC uses different consistency predicates in an uniform way, meaning that as a whole this makes it possible for GenMC to work for any memory model in a parametric way, as long as the model is axiomatic and the consistency predicate fulfills the following requirements [4]:

- **Well-formedness:** Memory consistency does not depend on the order in which events are added to the graph by GenMC. In consistent graphs, the causal dependency relation is acyclic.
- **Prefix-closedness:** Any prefix-closed subset of a consistent execution graph is consistent. This property must hold, to enable incremental construction of executions by GenMC's verification process.

- **Maximal extensibility:** Adding a po-maximal event to a consistent graph preserves consistency, if the event is added in a maximal way. For read events, this means that we read from the co-maximal write to the same memory location, and for writes this means being added co-maximally.

  Together, with these properties, we guarantee that GenMC can explore all relevant execution graphs to detect memory inconsistencies and safety violations without ever getting stuck.

Chapter 3

Verification

In the following chapter, automated verification techniques are presented. These techniques make up the foundation of GenMC's model checking algorithm. The goal of this chapter is to gradually introduce the TruSt verification algorithm employed by GenMC. TruSt dictates the verification process by defining how execution graphs are constructed, explored and checked for consistency under the chosen memory model. The visualization tool introduced in the second part of this thesis, tries to visualize the verification process of the TruSt algorithm incorporated in GenMC. Thus, a clear understanding of TruSt is essential. The focus lies on the high-level design of TruSt and the techniques it deploys, as a deep-dive into the low-level implementation details is not relevant for this thesis.

3.1 Automated Verification

Historically, all executions of concurrent programs were represented as interleavings of threads and their corresponding program operations under sequential consistency (SC). But, to be able to reason about weak memory models, we need a new way to represent concurrent programs, as thread interleavings cannot capture the behavior permitted by weak memory models. Thus, GenMC introduced a novel representation of those executions, by constructing them as graphs instead.

The verification process used by GenMC builds upon three key techniques. Each will be presented in their own subsection [4]:

1. Stateless Model Checking (SMC)
2. Dynamic Partial Order Reduction (DPOR)
3. Truly Stateless Optimal DPOR (TruSt)

Each technique will be presented shortly to give the reader an overview of challenges faced by historic model checkers of memory models and the corresponding optimization techniques applied by GenMC to resolve said challenges, leading GenMC to be an overall efficient and scalable automated verification tool, acting as a generic model checker for weak memory models, which did not exist before.

3.1.1 Stateless Model Checking

Stateless model checking (SMC) is an efficient automated verification technique designed to exhaustively explore all possible behaviors of a concurrent program without storing the states visited before. Exploring the entire state space by visiting all possible interleavings of concurrent operations one-by-one benefits from linear memory consumption, but causes a severe efficiency problem. As the program size grows, the number of possible executions grows exponentially with the size of the program. This problem is called state explosion and introduces a major tradeoff between exploration optimality and memory consumption in historical weak memory model checkers.

By keeping no state of previously visited executions, SMC enables parallel exploration, as different parts of the state space can be explored independently, and optimizes for memory consumption. Section 2.2.2 will show that these properties are also fulfilled and refined by TruSt as well.

3.1.2 Dynamic Partial Order Reduction

Before the introduction of TruSt, this trade-off posed a fundamental limitation for verification tools for weak memory models. In the next section, we will introduce the TruSt algorithm,, and how it allows for both exploration optimality and optimal memory consumption.

3.2 TruSt: Truly Stateless Optimal DPOR

TruSt is a dynamic partial order (DPOR) algorithm that is the successor of previous limited DPOR approaches that faced the inherent trade-off between exploration optimality and optimal memory consumption. As stated in its name, TruSt combines the advantages of both stateless model checking (SMC) and dynamic partial order reduction (DPOR) without the trade-off between exploration optimality and memory consumption.

Like stateless model checking, TruSt does not store data on previously visited states, which guarantees linear memory consumption and enables parallelization of the verification process just like SMC does.

Additionally TruSt achieves exploration optimality too, by guaranteeing that we do not explore more than one execution graph per equivalence class,

even though the notion of equivalence class is neither constructed nor any information of already visited equivalence classes is kept during the exploration process by TruSt.

Therefore, TruSt achieves both optimal memory consumption and exploration optimality. Additionally, since TruSt is stateless it is parallelizable. Although this is possible in theory, the thesis doesn't go any further into TruSt's parallelizability, as parallelization and multi-threading has not been implemented by the implemented visualization tool. This is definitely something that could be implemented in future work on this tool.

As a whole, TruSt combines both stateless exploration with optimal DPOR, providing us with an efficient and scalable verification algorithm for verifying concurrent programs under weak memory models. The recursive version of the TruSt algorithm is implemented in GenMC. The design of the visualization tool, which is presented later in this thesis, uses this algorithm as its conceptual foundation.

3.2.1 Incremental, Recursive, Immediate, Depth-First Exploration Process

TruSt explores execution graphs recursively. As already mentioned, step-by-step, TruSt chooses the next memory event from the program to be added to the current execution graph that is being explored, and immediately checks whether the newly constructed graph is memory consistent. Additionally, if there are alternative ways to add the chosen memory event to the current execution graph, we reconstruct the alternative execution graph, check whether it is a valid graph that is memory consistent and immediately explore the alternatives via recursive revisit calls. As we see, TruSt works is a simple, depth-first fashion by following the current execution graph, adding further memory events to it, revisiting alternative explorations until we cannot add any further memory events to the graph, implying that the current execution graph is complete, and therefore terminating the exploration of the current graph and return to a previous execution graph in the construction timeline to further add memory events to it in a unique way. Additional recursive revisit calls of the latest previous execution graph have priority when choosing the next exploration path to take.

All recursive revisit calls of an execution graph start off with the same execution graph structure. All added events and any other changes done to the execution graph from the previous exploration branch are undone after graph completion and termination of the current exploration path, after which TruSt cuts off all changes made before starting the exploration process of the next recursive revisit call.

TruSt guarantees that all exploration graphs built in the process are unique and no identical exploration path is built or revisited twice.

3.2.2 Revisits

In the exploration process, when we add a memory event to the execution graph we might explore alternative ways of adding said event to the current graph immediately via recursive revisit calls. We differentiate between forward and backward revisits. This subchapter will expand on both revisit kinds.

Forward Revisit: There are two kinds of forward revisits. We present both of them. A read forward revisit might happen, when we add a read event to the execution graph in the current exploration path. It depends on whether the read can read-from different writes of the current execution graph.

A write forward revisit happens, when we add a write event to the current execution graph, for which other write events exist that write to the same memory event, so that there exists an alternative coherence-order for corresponding memory location of said writes.

For both kinds of forward revisit, we explore the other possible explorations branches for given execution graph by an immediate recursive revisit call to the alternative execution graph, that either has a read event reading from a different write from the read forward revisit, or a different coherence order for affected memory location, that contains the last added write event to that memory location in a different position in said coherence order.

Figure (a) shows a snippet of an exploration tree where we perform a read forward revisit. As we see in the image, the leftmost thread, which represents the thread with tid 1, performs two release writes in program order (Wrel(x,1); Wrel(y,1)). The second thread adds an acquire read (Racq(y,1)) where depending on the chosen exploration path, there are two alternative writes to pick for the same location y. Read can read from both writes consistently (Wrel (y,1), INIT). Thus, TruSt performs an immediate recursive revisit call where the alternative write is picked, as seen in figure (a) via the pink edge which stands for a revisit event. As already mentioned before in chapter 1, for the visualization tool we chose not to show rf-edges leading from INIT event to other events, as it overwhelms the visualization of execution graphs from having lots of rf-edges all going out of the INIT event. The implicit [INIT] rf-label suffices.

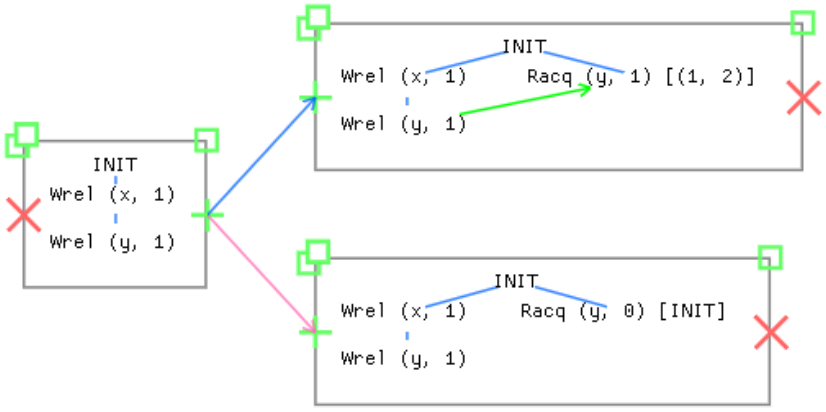


Figure 3.1: Read-forward revisit

Figure (b) shows an excerpt of an exploration tree illustrated via the visualization tool. The excerpt shows a write forward revisit where we have a graph where the first thread contains two write events already (Wsc(x,1) and Wsc(y,2)). Then, when we add an additional write event with the second thread (Wsc(y,2)), TruSt has two alternative ways in which we can add this write event depending on the coherence order to that specific memory location y. Just like for the read forward revisit, TruSt performs an immediate recursive revisit call where the alternative coherence ordering is picked, as seen in figure (b) through the exploration tree edge that is colored pink, which stands for a revisit event.

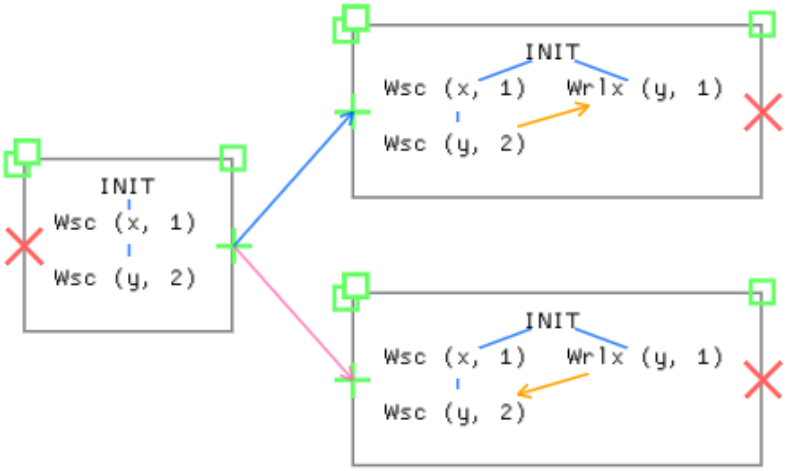


Figure 3.2: Write-forward revisit

Backward Revisit: A backward revisit event happens when we add a write event to a specified memory location in the current execution graph, for which the graph already contains some read event to that memory location. Then, one could have added the write event in an alternative exploration path before that specific read event, so that said read event could read from the newly added write event. Thus, to revisit this alternative exploration path, we delete all events in-between the read event and corresponding write event that are not dependent on the write event, and then revisits the changed execution graph from which we removed some events as stated.

Figure3 shows a snippet of an exploration tree where we perform a backward revisit. In the snippet shown, the initial execution graph on the left of the exploration tree has already observed three memory events. At the branch, the blue edge transitions to an execution graph that corresponds to normally adding the next write event (Wrel(y,1)) to the graph, while the lower pink edge transitions to a graph on which we performed backward revisit on based on the newly added write event (Wrel(y,1)). When we added the next write event to the current graph, TruSt detects that it writes to the memory location y, which already has been read earlier in the current graph (Racq(y,0)). Thus, TruSt initiates through an immediate recursive call the backward revisit, which places the write event and all of its porf-predecessor memory events in the execution graph before the read event chosen for the backward revisit. (here, Racq(y,0)). Logically, we need to remove all events that lie between the chosen read event and the write event, that are not porf-predecessors of the write.

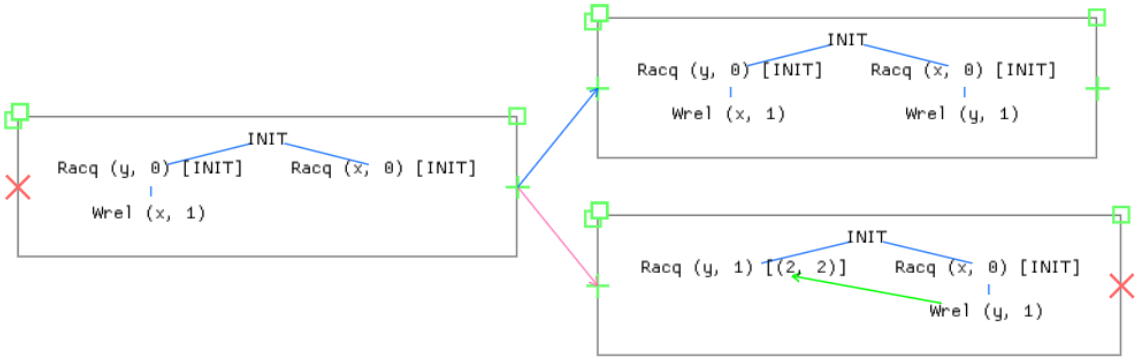


Figure 3.3: Backward revisit

Chapter 4

Exploration Logger

The TruSt exploration process implicitly creates a tree-based structure with execution graphs as nodes. While the algorithm itself does not expose this structure explicitly, as it operates solely on execution graphs, by keeping the current execution graph stored, and by relying on recursion and backtracking to ensure a correct and stateless exploration.

However, for the purposes of visualization, it is required to make the underlying structure observable. The visualization of the verification process should recreate the implied tree-based structure, that contains the incrementally, step-by-step created execution graphs as tree nodes, and execution graph through edges with the one they're derived from from the step before during the execution of TruSt.

This relationship between the individual execution graphs and their predecessor graph has to be extracted from the execution of TruST without any changes to the underlying algorithm itself. We achieve this objective by first finding out what kind of information needs to be extracted by us from the algorithm, to then inject the specific code sections containing that data with logger function calls, which retrieve the necessary data for us. The resulting logged data is then used to construct the visualization of the exploration process.

The goal of the logger is to treat the TruSt algorithm as a black-box verification algorithm and simply inject the code with additional hooks for logging structural information, in the form of a parent-child relationship between each graph and their predecessor in the exploration tree. It is important that the logger's hooks in the code do not change the algorithm's logic or correctness. Ideally the exploration should only extract information that has already been determined by the exploration process's tree-based graphical structure.

The following chapter introduces what kind of information is relevant for

the visualization tool, that we need to extract out of TruSt's algorithm during its execution and why the visualization tool cannot simply produce the tree-structure based on the individual exploration graphs alone without the parent-child relationship between graphs.

Then the chapter presents the TruSt pseudo code including the injected hooks from the logger, to show how we extract the visualization relevant information from the recursive exploration of the TruSt algorithm. Finally, it is explained how the extracted data reflects the implicit tree-based exploration structure produced by TruSt.

4.1 Recursive TruSt Algorithm

The execution of the recursive version of TruSt produces a tree-shaped structure over execution graphs. Whenever we extend the current execution graph by a memory event, recursive revisits might be initiated. Each recursive call explores a unique path in the exploration tree. Both kinds of actions, whether it is extending the current graph by the next memory event, or a new revisit instantiation, the algorithm treats both calls the same way.

Even though this implied tree-based graphical structure is not explicitly constructed, It is made explicit for the purpose of visualization, by injecting the recursive exploration process with a logging mechanism that records parent-child relationships between execution graphs explored during recursive calls. The logging mechanism is not allowed to alter the behavior of the algorithm. It merely collects information that is already available during the recursive exploration process of TruSt.

4.2 Parent-Child Relationships

The recursive function of the TruSt procedure defined in the paper has just a single execution graph G as function argument, next to the program we are trying to prove. Whenever the algorithm constructs a new graph G', either from extending G with a new memory event or by revisiting G, the exploration of G' is done via a recursive call with G' as argument.

The parent-child relationship that is induced by connecting the execution graph G that was passed to the recursive call as the parent graph and the execution G', that we got from further exploration of that call either by extension or alternative revisit, is the child graph.

The given parent-child relationship precisely describes the implicit structure of the exploration tree with the tree edges being between parent and child graphs and those graphs themselves being the tree nodes.

As already mentioned, the parent-child relationships of the exploration tree, occur in the algorithm whenever:

1. A read event extends the execution graph and a forward revisit is performed by selecting a reads-from relation
2. A write event extends the execution graph and a forward revisit is performed by selecting a coherence position
3. A write event is added and we perform a backward revisit under the maximal extension condition

4.3 TruSt + Exploration Logger Pseudocode

Algorithm 1 Exploration logger (with parent–child set)

```
 1: procedure ExplorationLogger(P)
 2:     let ParentToChildSet be a set of pairs G × G, uninitialized or empty
 3:     return PopulateParentToChildSet(P, G, ParentToChildSet)
 4: end procedure
 5: procedure PopulateParentToChildSet(P, G, ParentToChildSet)
 6:     if not memory_consistent(G) then
 7:         return
 8:     end if
 9:     a ← next_p(G)
        a = ⊥
10:     return "Visited full execution graph G"
        a ∈ error
11:     exit("error")
        a ∈ R
12:     for all w ∈ G.W_loc(a) do
13:         ParentGraph ← G
14:         ReadOrForwardRevisitGraph ← SetRF(G, a, w)
15:         add(ParentToChildSet, (ParentGraph, ReadOrForwardRevisitGraph))
16:         PopulateParentToChildSet(P, G, ParentToChildSet)
17:     end for
        a ∈ W
18:     ParentGraph ← G
19:     for all w_p ∈ G.W_loc(a) do
20:         WriteGraph ← SetCO(G, w_p, a)
21:         add(ParentToChildSet, (ParentGraph, WriteGraph))
22:         PopulateParentToChildSet(P, G, ParentToChildSet)
23:     end for
24:     for all r ∈ G.R_loc(a) s.t. ⟨r, a⟩ ∉ G.porf do
25:         Deleted ← {e ∈ G.E | r <_G e ∧ ⟨e, a⟩ ∉ G.porf}
26:         if ∀e ∈ Deleted : Is_MAXIMALLYEXTENDED(G, e, a) then
27:             for all w_p ∈ G.W_loc(x) do
28:                 WriteBackwardRevisitGraph ←
    SetCo(SetRF(G_|G.E\Deleted, r, a), w_p, a)
29:                 add(ParentToChildSet, (ParentGraph, WriteBackwardRevisitGraph))
30:                 PopulateParentToChildSet(P, G, ParentToChildSet)
31:             end for
32:         end if
33:     end for
        otherwise
34:     PopulateParentToChildSet(P, G, ParentToChildSet)
35: end procedure
```

It is clear by now, that the main goal of the logger is to visualize the implicit exploration tree induced by TruSt's recursive exploration process. Since TruSt itself doesn't explicitly construct the tree structure, it is necessary for us to retrieve the structural data required to construct the exploration tree through our visualization tool. In the exploration tree, every execution graph corresponds to a unique node. Whenever TruSt explores the next step during execution it makes a recursive call. Whether the next step is a graph extension by adding the next memory event to the current execution graph, or simply a revisit event, does not matter. Both actions add a new directed edge from the parent graph at the caller site of the recursive call to the child graph being explored by the callee. The modified pseudocode of the recursive TruSt algorithm makes these exploration tree edges between parent and child graphs explicit, through the ParentToChildSet storing pairs of parent-child execution graphs. Conceptually, the resulting set is actually enough for constructing and visualizing the tree-structured exploration process of TruSt.

4.4 Parent-Child Graph Relationships in the Pseudocode

In the pseudocode, the recursive procedure of the original TruSt algorithm has been modified and renamed to PopulateParentToChildSet(P, G, ParentToChildSet). The modified procedure has the current execution graph G as a parameter. Each recursive call, the procedure checks whether the graph G is memory consistent. If not, the procedure returns false. Otherwise, the procedure extracts the next memory event chosen by TruSt to extend G to G'. Now, we have access to both, the parent graph G and the corresponding child graph G', which is connected to the ParentGraph G via a directed edge in the exploration tree. Therefore, we add the pair (G, G') to the ParentToChildSet through ParentToChildSet.add(ParentGraph, ChildGraph) immediately before the next recursive call of the PopulateParentToChild procedure. For cases where no recursive call of the procedure is triggered, we also do not add any parent-child graph pair to the set.

The main take away of this subsection is that the logging itself does not change the behavior or correctness of the TruSt algorithm. It merely adds observational logging hooks to the algorithm, which does not alter the algorithm.

4.5 Hook Injection Strategy

We keep track of the implicit parent-child relationships by introducing a set that stores the parent-child pairs of the corresponding identifiers of the parent and child execution graphs. The set is initialized as empty, before the start of the verification process.

The hook injection strategy is simple and consists of injecting hooks immediately before each recursive call that explores the graph G' that we got from modifying execution graph G by selecting an available reads-from relation or choosing a coherence position or in the case of backward revisit where we modify the execution graph G even further.

The hook itself keeps track of the identifier of the current execution graph (parent) and the identifier of the execution graph passed to the next recursive call (child). The identifier is computed by hashing the execution graph.

4.6 Implicit Tree Structure

As already stated, the logging mechanism keeps track of parent-child relationships exclusively before each recursive call, so that we can derive from the recorded relationship the exploration tree structure by following following following definitions:

- Every execution graph is explored exactly once by the algorithm. Each of these exploration graphs corresponds to a unique tree node
- Each graph extension or revisit step that leads to a new recursive call corresponds to an edge between parent graph G and child graph G'
- The initial execution graph is empty and represents the root of the exploration tree

Overall, the most important attribute of the hook is that it does not interfere with the underlying algorithm, meaning that the implemented logging mechanism does not change which execution graphs are explored, the order in which said graphs are explored, the extension restriction, or the correctness of the algorithm.

4.7 Practical Constraints of Hook Injection in GenMC

The hooking mechanism that we described in the previous sections works in theory, as in the recursive TruSt algorithm presented in the paper. But, in the actual implementation of the TruSt algorithm in GenMC, we cannot simply record the parent-child graph pair whenever a recursive call is made. The actual implementation doesn't allow for such a logging mechanism and requires additional changes.

These changes will be presented in the following subsection, for which the take away is the introduction of an explicit parent identifier stored within revisit objects. The modification is strictly observational and doesn't affect the algorithm's behavior or its correctness.

4.7.1 Atomicity in the Pseudocode

In the pseudocode that we presented in the previous section, the graph transformations are tightly coupled to the recursive calls and are atomic. Every time we modify an execution graph G' from the current execution graph G, we immediately record the parent-child pair (G, G'), store it in the ParentToChildSet, and perform a recursive call with modified graph G'. With this logic, both the parent graph G and the modified child graph G' exist at the moment the recursive call is initiated, which makes recording the parent-child relationship possible by simply storing a pair of graph instances.

4.7.2 Non-Atomicity in the GenMC Implementation

In the GenMC implementation, the graph transformations are not tightly coupled to the recursive calls and do not modify the current graph G into the graph G' atomically before performing the recursive call. Instead, whenever the current graph is modified by extending the graph through adding an event to the graph G, a chain of operations that modify the state of the graph are triggered. The chain includes operations like adding labels, unblocking threads, and computing revisits.

The computed revisit objects are created during this sequence of operations, before the actual child graph exists. Therefore, we cannot record the corresponding parent-child graph pair during the creation of revisit objects. Additionally at the time a revisit is initiated, the current graph has been further modified, so that no connection to the parent graph exists. Consequently, no point in the program flow of GenMC both the parent execution graph and the corresponding child execution graphs exist, so that there is no point in which we could log the parent-child pair.

4.7.3 Queued Revisits in the GenMC Implementation

The second difference concerning the recursive exploration process of the TruSt algorithm and the actual implementation of TruSt in GenMC is that revisits are not immediately explored via recursive calls. In GenMC, the revisits are queued for later execution using a work queue. Therefore, a revisit might be constructed at one point during the execution of GenMC, but will be executed much later in the exploration process.

As mentioned in the previous subchapter on the non-atomicity in GenMC, when a revisit is dequeued and explored, the parent execution graph state that existed during the construction and queuing of the revisit object, does no longer exist in its original form in memory, as TruSt works in a stateless fashion.

This delayed execution of revisits in GenMC, in which the creation of the revisit object, queuing of the revisit in the workqueue and the actual execution of the revisit, are separated and performed during different points of the execution of GenMC. This caused a challenge for the logging system required to record and keep track of the parent-child relationship.

4.7.4 The Timing Problem of the Logging Mechanism

To record a parent-child relationship, only the identifiers of the parent execution graph and the child execution graph are necessary. But, as explained in the previous subchapter, both identifiers are never simultaneously available for revisits. When a revisit instance is created, the parent execution graph is available, but the child graph does not yet exist.

Afterwards, during the revisit exploration, the child execution graph is known, while the parent graph no longer exists,

Thus, it is logically impossible to reconstruct the parent identifier without a complex, independent logging system that makes use of the underlying workqueue revisit system of the GenMC implementation. Although such an approach is possible in theory and would make the logging system as a whole, independent, easier to extend and maintain, such an implementation was not possible in the scope of this thesis and remains to be implemented in the future.

4.7.5 Necessity of an Explicit Parent Identifier

To be able to connect the parent execution graph to the child execution graph, we must compute the parent graph identifier at the moment the creation of the revisit object. Then, the identifier must be kept in memory until the revisit is dequeued from the work queue and executed, at which point we can retrieve the parent graph identifier and pair it with the child graph identifier of the current revisit execution.

This idea is implemented through an additional explicit field called parentHash, which stores the parent graph hash identifier within the revisit object, making the parent graph identifier accessible during the revisit execution.

4.7.6 Observational Logging System

Adding an additional field within the revisit object defies the statelessness and the parallelizability of the TruSt algorithm. But, it is still important to state that introducing such a parent identifier does not modify the TruSt algorithm itself.

4.7.7 Final statement on Logging System

The logging system described above, including the additional parentHash field in the revisit object, makes the implicit exploration tree of the recursive TruSt algorithm explicitly observable without changing the algorithm's behavior or correctness.

However, the collected exploration tree data in the form of parent execution graph and child execution graph pairs has still to be visualized, which will be presented in the following second part of the thesis that introduces the reader to the practical visualization tool implementation.

Chapter 5

Visualization

In the previous chapter, we presented a logging mechanism that makes the recursive exploration process of the TruSt algorithm explicit by extracting the necessary structural information during the exploration itself.

In the following chapter, we will go into how we can construct a visualization of the implicit exploration tree of TruSt from the data that we extracted through logging.

The visualization tool developed in this thesis takes the extracted parent-child relationship data and transforms them into an interactive graphical representation of the verification process. The intended use of the visualization tool is to make the exploration of execution graphs explicitly visible to users of GenMC, so that it makes the underlying verification process more comprehensible by rendering the whole exploration tree that shows both the individual executions as graphs and their parent-child relationship between recursive calls of the TruSt algorithm as edges in the exploration tree.

First, we will introduce the visualization tool pipeline. The process begins by taking the execution graph objects that are produced during the execution of GenMC and serializing them into a compact representation that retains only the information necessary for the final visualization by abstracting away irrelevant data. The serialized form of the exploration process acts as the interface between GenMC and the visualization tool.

Afterwards, the chapter explains how we extract the serialized graphs into an exploration tree whose nodes correspond to the execution graphs and whose edges represent exploration steps initiated by graph extensions and revisits.

In the next section, the chapter will go into the layout algorithm used for the two dimensional exploration tree in the visualization.

In the end, we conclude the visualization centered chapter by presenting the framework used (Dear ImGui) and an overview of the rendering process.

5.1 Serialization of Execution Graphs

We use a serialized representation of the actual execution graphs, which are produced during the verification process of GenMC, to then feed into the visualization tool. In the following section we will explain how we extract execution graphs during the execution of GenMC, how they are then serialized into a compact log format, and how we reconstruct the graphs for the visualization.

Overview of the Extraction Pipeline:

- Serialization during the exploration process, when we log the execution graphs
- Writing the serialized graphs and their parent-child graph relationship to a log file
- Reconstruction of the logged graphs by parsing said log and building corresponding graph objects

The visualization is an independent program, that is entirely decoupled from the verification. During the verification process, GenMC logs explored execution graphs together with the relevant parent-child relationship information to a log file. The log file acts as an interface between GenMC and the visualization tool, which uses only the log file to produce the visualization.

5.2 Hashing Execution Graphs for Identification

To be able to identify execution graphs reliably and connect graphs precisely in the exploration tree through edges by their parent-child relationship information, which consists of pairs of the parent and child graph identifiers, we use a hash value computed via a hash function. The hash function doesn't hash the whole graph object state. Irrelevant information is excluded from the computation.

Only memory-relevant events, their relations in the execution graph and information on the threads is used for the hash function. With these hashes we can reliably identify execution graphs throughout the whole visualization pipeline.

5.3 Capturing the Parent-Child Graph Relationships

To reconstruct the exploration tree we need to derive the connection between execution graphs in the exploration tree, as those connections represent the edges of the exploration tree. As already mentioned in section 4.7.5, this requires an additional field for revisit objects, so that we can capture the parent execution graph identifier when a revisit object is created. The identifier is computed through the hashing function that we presented in section 5.2.

Later, when the revisit is explored, we retrieve the stored parent hash identifier and use it to log the current execution graph of the revisit together with its parent hash to the log. This way, we can bridge the gap between the separate times where the parent graph of the revisit is known and when the child graph is known, which makes it possible for the logging system to create the parent-child relationship reliably during the execution of GenMC.

By the property of the TruSt algorithm that no execution gets explored twice, it is guaranteed that every logged execution graph references one unique parent graph, producing an acyclic exploration tree implicitly induced by TruSt.

5.4 Graph Serialization

To be able to parse execution graphs efficiently the execution graphs are serialized into a compact, textual format when the graphs are logged to the corresponding log file of GenMC's verification process. The serialization produces a simple graph block that will be written to the log file, which contains:

- For each thread a chain of events ordered by program order
- Coherence order information for write events
- Graph hash identifier together with the parent hash identifier connecting the graph to its predecessor in the exploration tree

The serialization leaves out irrelevant information, leaving us only with memory-relevant events for the logged graph blocks. This simplistic representation is enough to log all the relevant information for the visualization tool.

Example. This small example is meant to give the reader some intuition on the graph serialization process. The example graph has been chosen arbitrarily.

```
ExecutionGraph:
(1, 1): Racq (y, 1) [(2, 2)]
(2, 1): Racq (x, 0) [INIT]
(2, 2): Wrel (y, 1)
Hash: 12370887523603231640
ParentHash: 12386487064451908168
Revisit
```

The line `ExecutionGraph:` is used as a marker to separate graph blocks from one to another by putting it at the beginning of every serialized graph block. Next, the subsequent lines cover for each thread a chain of memory events ordered by program order until the line that containing `Hash: ...` and `ParentHash: ...` which is the textual representation of the parent-child graph relationship pair for the current graph in the example. The parent hash connects the graph block to the predecessor graph in the log file, which corresponds to the parent graph in the exploration tree, that is connected to the child graph (example graph) through an edge. The optional `Revisit` line is used as a marker for the visualization tool, to identify graph blocks of revisit events, which as we saw before get colored in pink, instead of the standard blue color.

5.5 Extracting Execution Graphs in the Visualization Tool

To extract the execution graphs, we parse the serialized log file line-by-line, and extract from the textual graph blocks the corresponding graph object used for the visualization by the visualization tool. The parsing process is simple. During parsing, the events are converted into graph nodes, the edges are constructed based on the program order inside of each thread, and reads-from relations and coherence order are also extracted from the logged graph blocks. Additionally, we add a single unique node to the converted graph which corresponds to the init event that got introduced in chapter 2.

Additionally during the parsing, we take notice of the parent hash identifier and the graph hash identifier to construct the parent-child relationship by maintaining a set containing pairs of parent and child graph hashes. These mappings are used by the visualization tool to connect individual execution graphs as nodes in the exploration tree.

Chapter 6

Layouting and Interactivity

The inherent tree-structured exploration tree induced by the verification process of GenMC represents the exploration performed by the TruSt algorithm. To visualize the exploration tree in a dynamic way, allowing for all different kinds of tree-based structures to be displayed by our visualization tool, we have to position the individual exploration graphs in a manner that reflects the hierarchical parent-child graph relationships. The algorithm prevents overlapping execution graphs.

In this section we will describe the layout algorithm used for the exploration tree. The algorithm is a modified version of the classical Reingold-Tilford tidy tree algorithm [8], that extends specific parts of the algorithm for our use case. The biggest problem faced by our layouting algorithm is that the nodes (execution graphs) can have significantly different sizes based on the number of events contained by those execution graphs.

6.1 Overview

The nodes in the exploration tree represent execution graphs. Additionally, the nodes store the parent and child graph pointers based on the parent-child graph relationship information we extracted prior, that are required for the construction of the exploration tree edges.

The algorithm follows the classical multi-walk approach inspired by the Reingold-Tilford implementation, with the exception of an additional third walk for global overlap detection.

The strategy is as followed:

1. **First walk**: Bottom-up computation of temporary vertical node positions and the local overlap between parent and child graphs (siblings) representing predecessor node and current node connected to each other through an exploration tree edge

2. **Second walk**: Accumulating and shifting the temporary vertical positions into the final vertical node positions
3. **Third walk**: Special exploration tree pass for global horizontal overlap detection to detect horizontal overlap between non-sibling exploration tree nodes, which is possible due to the property of arbitrarily sized nodes (execution graphs)
4. **Final adjustments**: Applying the accumulated detected overlap corrections by the third walk to retrieve the final horizontal node positions

The presented algorithm is an extended version of the Reingold-Tilford tidy tree algorithm, which we extended to handle horizontal overlap faced by arbitrarily sized execution graphs (nodes) in the exploration tree. To produce a dynamic generic exploration tree layout for the GenMC verification process to work for arbitrary exploration tree induced by the TruSt algorithm, our tree layout algorithm combined dynamic spacing, and multi-walk overlap detection and overlap correction. The algorithm being dynamic allows the visualization tool to be interactive and change the exploration tree at runtime, as the tree layout algorithm will simply adapt and display a new tree layout for the new tree structure.

6.2 Interactive Exploration of the Tree Layout

The exploration tree layout is interactive, which is possible because of the dynamic overlap detection and correction of the tree layout algorithm. The interactivity allows the user to hide execution graphs, collapse and expand whole subtrees of the exploration tree, and the tree layout must always update based on the modifications on the exploration tree.

The purpose of introducing interactivity in the first place, is to be able to explore extremely large exploration trees that represent the inherent verification process of the TruSt algorithm implemented by GenMC.

6.3 Design

The exploration process can produce a large amount of execution graphs that grows with program size. The overwhelming size of the tree structure is hard to follow. To solve the presented complexity problem, we introduce interactivity to our visualization tool, that allows users to hide irrelevant parts of the exploration tree.

The main challenge is for the interactive elements of the visualization to not change the structure of the exploration tree itself. The visualization tool does this by modifying only visibility of the exploration tree on the screen, meaning that the underlying exploration tree structure is still stored unchanged,

while the visualizer focuses only on rendering the remaining visible graphs on the screen. The dynamic layout automatically recomputes the positioning of the exploration tree and also detects and corrects any overlap between nodes of the visual visible tree structure, making the visualization adaptable for user-driven changes.

6.4 Graph-Level Interaction

Each execution graph is rendered with a set of interactive elements positioned on its bounding box. The interactive actions produced by clicking on said interactive elements is intuitive and will be presented in this section.

Child Connector: Collapse and Expand Descendant Subtrees The interactive element placed at the middle of the graph's right edge is called the child connector. Its purpose is to control the visibility of the whole descendant subtree of the current graph. The element can be toggled at any time, whereas visually, a green plus symbol represents an expanded descendant subtree, and a red X a collapsed descendant subtree.

The underlying mechanism is really simple and based on a single counter. The mechanism allows for nested collapses, meaning that if a descendent subtree gets collapsed and an ancestor of the current execution graph is collapsed as well, then based on the counter, the descendant subtree only gets expanded when both the ancestor child connector and the current graphs child connector get toggled.

Figures (a) and (b) show an example of toggling the child connector of a specific execution graph. In figure (a), the child connector is in its collapsed state, as is visible by it being a red cross. Therefore, all of its descendant subtrees and outgoing edges from the child connector get collapsed and aren't rendered anymore. In figure (b), we toggle the child connector again, to simply show the uncollapsed state of the child connector.

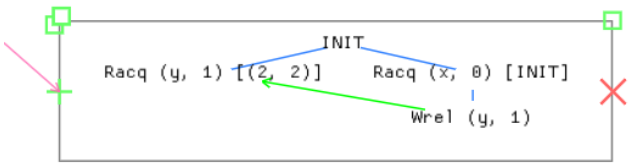


Figure 6.1: (a) Toggled child connector in collapsed state (red X)

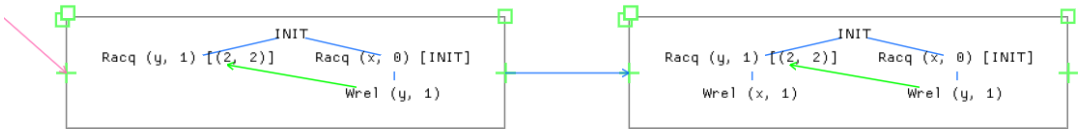


Figure 6.2: (b) Visible child graph that was hidden in (a)

Parent Connector: Collapse and Expand Ancestor and Sibling Subtrees

The interactive element placed at the middle of the graph's left edge is called the parent connector. Its purpose is to control the visibility of its ancestor subtree and the rest of the graph that is not part of the subtree rooted at the current execution graph whose parent connector we've toggled. Therefore, the functionality of this element is to highlight the current exploration subtree and ignore the rest of the exploration tree, by hiding all ancestor and sibling subtrees.

The visual effects of toggling the parent connector are the same as for the child connector.

In the following figures, (a) shows an example of a toggled parent connector in collapse state with all of its ancestors and sibling subtrees being collapsed and not rendered, and figure (b) shows the normal case of GenMC without any interactive effects initiated by the user.

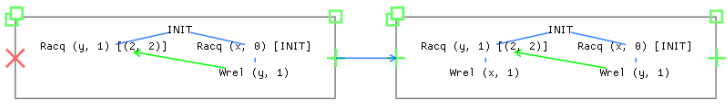


Figure 6.3: (a) Toggled parent connector in collapsed state (red X)

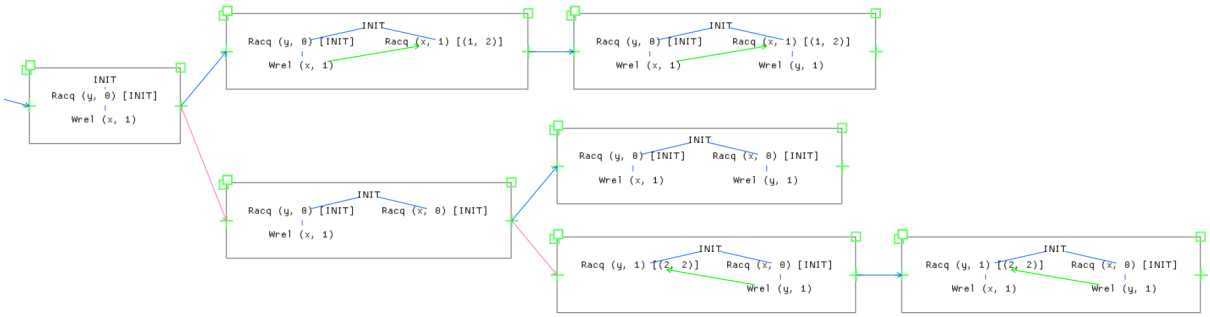


Figure 6.4: (b) Visible parent and sibling subtrees that were hidden in (a)

6.5 Hide Execution Graphs

Hide Graph Button: The hide graph button, that is positioned at the top-right corner of the bounding box, has the purpose to control the visibility of the execution graph inside of said bounding box. When activated, the graph is hidden and only a placeholder empty box is still visible. This shrinks the size of said node to the minimum and puts the focus on the rest of the visualized exploration tree.

When the element is toggled, the button changes color from green, which represents a visible execution graph, to red, which represents a hidden execution graph.

Figure (a) is an example of a hidden execution graph and figure (b) an image where that specific graph is shown.

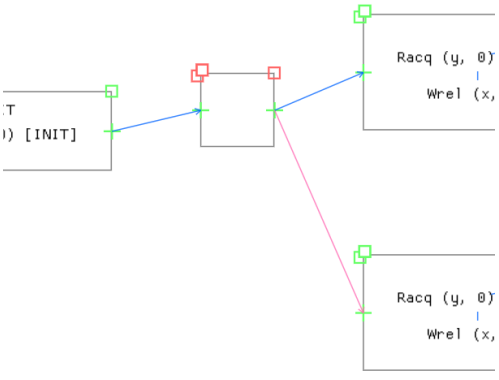


Figure 6.5: (a) Hidden execution graph

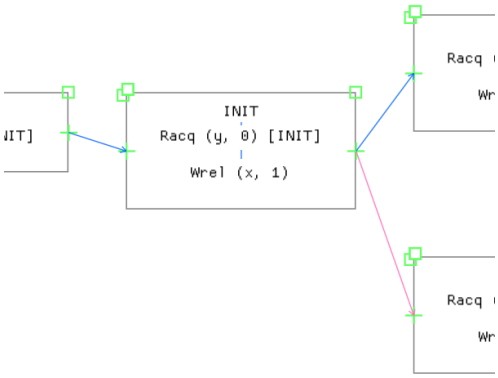


Figure 6.6: (b) Visible execution graph that was hidden in (a)

Hide Subtree Button: The hide subtree button, that is positioned at the top-right corner of the bounding box, has the purpose to control the visibility of the execution graph inside the current bounding box and the visibility of all execution graphs of all of its descendants. The reason the visualization tool provides this functionality next to the child connector interactivity, is to allow users to make the visualization more compact, while still giving users the possibility to toggle the visibility of hidden graphs or entire subtrees, giving users more freedom while interacting with the visualization tool.

Figure (a) is an example where all subtrees and execution graphs are visible, while in figure (b) we have the same example as in (a), except that a subtree is hidden.

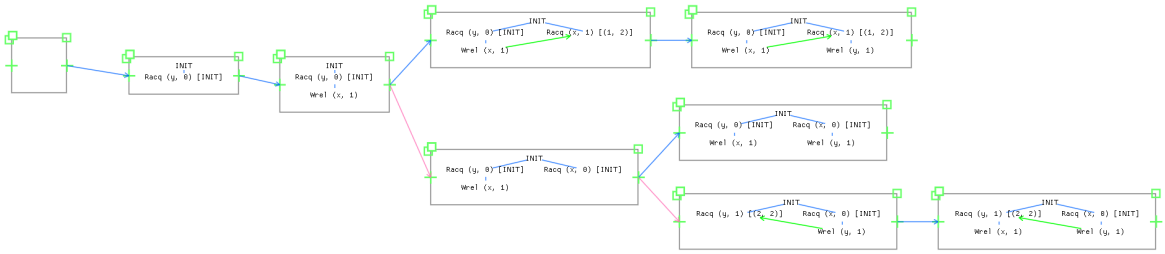


Figure 6.7: (a) Visible exploration tree

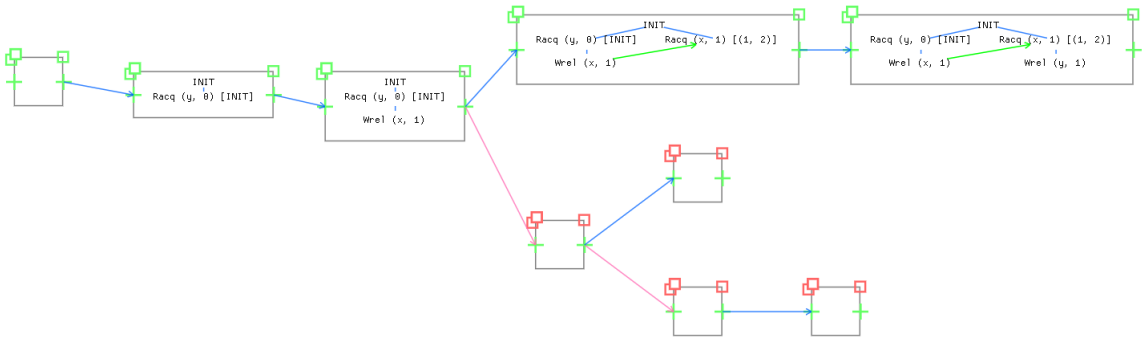


Figure 6.8: (b) Hidden subtree of the exploration tree in (a)

6.6 Compact Mode

In the exploration tree we may have long sequences of execution graphs, where each of the graphs has only one child. Even though these linear sequences are relevant for the verification process, for users trying to navigate and look through the tree structure, these sequences do not add a lot of information for understanding the overall exploration process.

Therefore, we add a compact mode to the visualization that reduces the verbosity of the exploration tree, by removing all linear sequences of execution graphs, where each graph has only a single child, into a compact form, where we start from the beginning node of the linear sequence and directly connect it via an edge to the final node of the sequence, while leaving out all other intermediary. This mode will highlight the branching points, and removes all intermediary nodes that do not create alternative paths, meaning that the focus is put on structural decisions of the TruSt algorithm, instead of the verbose step-by-step exploration process of GenMC.

6.7 Zooming and Panning

As already mentioned before, the size of the exploration tree can be overwhelming and not fit on the visible screen. This makes spatial navigation like zooming, or panning a requirement. Thus, through panning and zooming, the visualization tool provides zooming and panning for navigating on the UI screen.

Panning allows the user to translate the visible screen space across the exploration space that contains the whole size of the exploration tree that may be larger than the screen size itself. Zooming allows the user to change from an

overview of the graph with little to no detail, up to a close view of individual execution graphs when zoomed in really close. All visible elements of the exploration tree scale dynamically to the zoom level, which guarantees that nodes, the graphs those nodes contain, and their interactive elements stay readable and accessible. Together zooming and panning allow users to explore the visualized tree structure in an effective and intuitive way, even when the actual exploration tree exceeds the size of the visible screen.

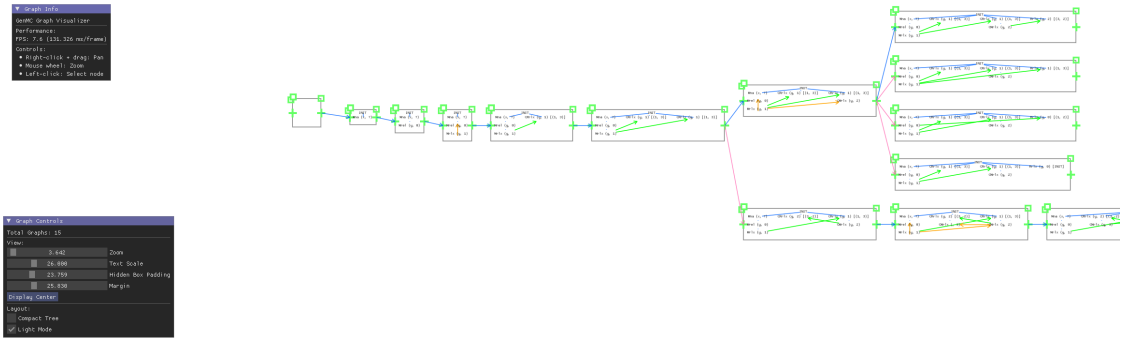


Figure 6.9: (a) Zoomed-out view with overview of the whole tree structure

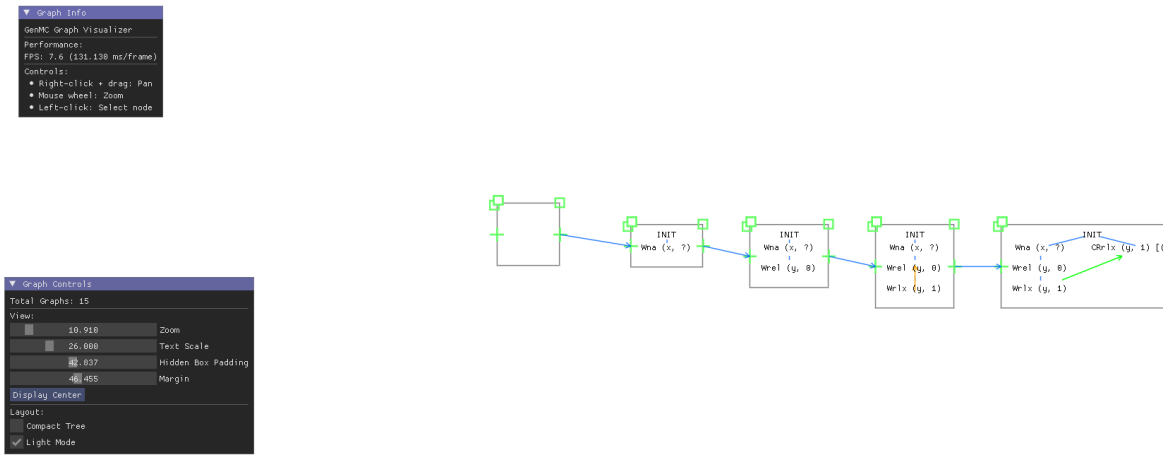


Figure 6.10: (b) Zoomed-in view without overview. Shows detail but lacks overview of exploration tree structure

Figure (a) and figure (b) showcase zooming and panning for navigation on an UI showing some arbitrarily chosen visualization of an exploration tree. In figure (a), the zoom is close to the exploration tree, increasing the overall amount of detail the user can perceive of the visible, individual execution graphs and their relations. Yet, the number of execution graphs fitting on the screen in figure (a) with close zoom is a small subset of the whole exploration tree. Therefore, in figure 2 we zoom out and pan the view on the screen so that the global structure of the exploration tree can be seen, which gives us an overall overview of the tree-structure, yet the amount of detail of the individual execution graphs and their relations decreases drastically.

6.8 Summary

The key take away of interactive functionality of the visualization tool is that the whole process is only possible because of the dynamic relayouting, as without it, interactive collapsing and expanding of subtrees would cause large gaps or overlaps between individual execution graphs, rendering the final visualization useless.

Therefore, we compute the exploration tree layout every time the visible elements of the exploration tree change through interaction of the user with the visualization tool.

To conclude chapter 6 on the dynamic exploration tree layout and the interactive elements, users benefit a lot from the combination of both features, as it allows them to interactively change the level of detail of the visualized exploration tree and be able to keep a clear overview of the possibly overwhelming state space of the TruSt algorithm shown on the screen as through the visualization tool.

Chapter 7

Immediate Mode Graphical User Interface

7.1 Immediate Mode Rendering with Dear ImGui

In this chapter, we will go over the rendering paradigm used for our visualization tool. In our case, the tool utilizes Dear ImGui's immediate mode model [6] that handles rendering, updates and interactions with our visualized exploration tree. The focus of this chapter doesn't lie in presenting the reader with the code in the form of a documentation, but to highlight the conceptual properties of immediate mode rendering, the Dear ImGui architecture, and the main rendering and interaction loop for the visualization of GenMC's execution graphs.

7.2 Core Principles of Immediate Mode Rendering

Given a user interface in the form of an interactive window containing the visualization of the inherent exploration tree produced by TruSt, the immediate mode rendering paradigm [2] is a paradigm that takes the complete user interface and renders it every frame again. The whole user interface (UI) is treated as a description that gets reconstructed each iteration of the main loop, which corresponds to exactly one frame.

Immediate Mode Rendering System:

- UI elements that get re-rendered every frame are not stored in a persistent manner and reconstructed every single frame
- No persistent UI state is stored
- The visualization is determined by the code every single frame anew

Immediate mode rendering follows a strict pipeline where each frame we discard the previous UI state, collect user input, reconstruct the UI and the

UI state, collect drawing commands from the code, and render the final visualization.

By reconstructing the UI state every frame, the immediate mode rendering paradigm guarantees that there are no inconsistencies between the visualization and the code. Thus, immediate mode rendering is one of the most straight-forward and easy-to-learn rendering modes for creating graphical user interfaces (here, the exploration tree).

7.2.1 Advantages of ImGui

The reason we use immediate mode rendering is its simplicity [2]. As already mentioned, each frame we reconstruct the UI state and all UI elements, so that we do not need to persistently store UI objects and maintain them which would require complex state synchronization between application logic and UI objects, lifetime management of objects, and a lot more that is not part part of this thesis.

The simplicity of immediate mode rendering gives us high flexibility and is ideal for dynamic interfaces. Especially for a dynamic interface like our GenMC visualization tool, where the whole visualization changes constantly through user interaction, where structure, size and visible elements are updated frequently.

7.2.2 Trade-off

The main trade-off of immediate mode rendering is between simplicity and efficiency. While immediate mode rendering is simple and easy to understand, the rendering increases the CPU workload, as we reconstruct the UI every single frame.

7.2.3 Comparison of Immediate Mode and Retained Mode Rendering

As already mentioned in the last subsection about the advantages of ImGui, it offers simplicity and flexibility, but has overall worse performance. This is perfect for non-performance-centric, interactive applications the TruSt exploration tree that can shrink, grow, and update the layout frequently, based on user interaction. Implementing such a visualization through retained mode rendering [2], is complicated and would require us to use complex synchronization between UI objects and application state to be able to implement the visualization tool.

Since ImGui avoids these issues by always re-constructing both the UI and the current state every frame, we pick ImGui for the implementation of our visualization tool.

7.3 Dear ImGui Architecture

The immediate mode graphical user interface (ImGui) library Dear ImGui is a library that is designed around the immediate mode paradigm presented in the last subchapter, which states that the UI is rebuilt every frame anew. Dear ImGui provides both high-level widgets and a low-level drawing API.

While the application code orchestrates the rendering and interaction procedures, the platform- and renderer-specific backends required for forwarding user input from the operating system to the ImGui and sending back draw commands to the GPU, which will be drawn onto the OS window that represents the user interface.

Rendering in Dear ImGui is done through draw lists, which as the name suggests, record all draw commands per frame and send them back to the GPU. For our visualization tool, we rendered the exploration tree by using the background draw list, while we keep only control panels and displayed information inside of Dear ImGui managed windows on top of the background.

The set of commands that we can send to the draw list every frame are simple geometric primitives like lines, text, or rectangles.

The rendering process follows for each frame the following structure:

1. Input handling: Collect mouse state and reset the internal UI state which contains mouse interaction information, required for detecting and handling mouse input
2. UI logic: Application code sends drawing commands to the draw list, which will be visible in the next frame
3. Rendering: Converting the draw lists into the correct format and sending them to the GPU

7.4 Rendering Pipeline

The visualization tool follows the presented rendering process structure closely. First, we initialize the parent-child graph relationship set and the individual execution graph objects, then we enter a continuous rendering loop that keeps running until the end of the program.

7.4.1 Rendering loop

The loop starts off by polling input events and updating ImGui's internal input state, that our application code can access during input handling.

Then, the entire exploration tree is visualized by re-rendering everything. That contains the exploration tree, the individual execution graphs, and the interactive elements.

7.4.2 Expensive Layout Recalculation

Since Layout computation is expensive, we do not perform it every frame. We only recompute the layout when parameters that affect the size and positioning of visible events, like changing the zoom level, the text scale of labels, the margin size between individual nodes of the exploration tree, or whether the node or subtree is visible or not.

When no change happens, we simply take the positioning of the UI elements of the last frame.

7.4.3 Graph Rendering

Graphs are rendered sequentially one-by-one and use world-space coordinates during the rendering process. To allow for viewport navigation (zoom and panning) we transform the world-space coordinates into screen-space positions of the individual execution graphs.

Visible execution graph is rendered through following separate components:

- Bounding box and interactive connectors
- Edges: Program-order (po), reads-from (rf), and coherence (co)
- Execution graph node labels that scale with zoom, to allow for better readability

The connections in between graphs in the exploration tree are drawn separately and use different colors depending on the type of exploration step taken. For normal exploration steps where we extend the ancestor graph with another memory event are colored blue. Revisit events are colored pink.

7.4.4 UI Window

As mentioned above, we draw the exploration tree in the background using a background draw list, while we only use ImGui-managed windows rendered using ImGui's standard window system. The windows are used for providing the user with information on the visualization, like the frame rate, the graph count, and viewport navigation details (e.g., how to pan with the mouse), and they are also used for controlling values like zoom level, margin, and text label size. While the background drawing and interactivity logic has to be managed by me, the ImGui-managed windows are managed automatically by Dear ImGui.

7.4.5 Input Handling and Interaction

In this chapter, we introduced the rendering process structure, where we mentioned that the internal input state of Dear ImGui is updated every frame. After that, the next step is for the visualization tool to perform interaction handling by extracting the input state during each frame and recording relevant input information like the mouse position, mouse dragging status, mouse scroll input.

To detect user interaction with our visualization of the exploration tree on the OS window that is used for showing our graphical user interface to the user, we use an invisible, full-screen capture window. This way, we can compute whether a mouse click has clicked an interactive element in the exploration tree visualization, and any other kind of mouse event, like dragging.

Hit detection of UI interactive elements is done through simple geometric tests based on the position of the interactive element and the coordinates of the mouse click. This approach is highly efficient and takes linear time.

7.5 Performance

The main objective of the visualization tool developed in this thesis is to bring GenMC's exploration process closer to users by conveying the exploration in a human-readable, intuitive format. To achieve that goal, we present the exploration process in its natural, two dimensional tree-structure.

Since the tool is meant to be an educational tool meant for researchers and users that are unfamiliar with weak memory models, the visualization tools main focus is to work with small, or medium-sized concurrent programs.

Depending on the computer specifications, the tool runs with approximately 10 to 40 FPS and works for concurrent programs whose exploration tree has maximally hundreds of execution graphs as nodes. In the context of this thesis the focus did not lie in further optimizing the performance of the tool to work with exploration trees that have a number of nodes in the thousands. On the contrary, the visualizations tool should work in an efficient and reliable manner for small or medium sized examples.

It could be a future work idea to work on the performance and scalability of the visualization tool, to be able to visualize the same large programs that GenMC can verify.

7.6 Backend Technologies

For now, we've discussed how we populate the draw lists of Dear ImGui for our visualization tool, yet the draw lists alone are not capable of rendering their content by themselves. Therefore, Dear ImGui translates the contents we buffered into the draw lists into a backend-agnostic command stream [7] that will be sent to some graphics API for displaying the draw list contents on the GUI.

The most common backend, that we used for the visualization tool as well, is implemented using OpenGL (GPU-accelerated rendering) [9] and GLFW (cross-platform window and input management) [10].

The backend handles the interactive notion of our visualization tool and displays the exploration tree through an OS window to the user.

7.6.1 OpenGL: Rendering Backend

OpenGL is a cross-platform graphics API that provides an uniform way to render graphics, like the two dimensional tree-structured graphical exploration of TruSt. OpenGL handles the act of rendering the contents buffered inside of the draw lists on the OS screen.

OpenGL is widely used and user-friendly because of its simple graphics API, which parallelizes heavy, interactive rendering workloads to allow for smooth graphics even when handling GPU-intensive loads. In our case this would benefit interactive elements or the visualization tool like zooming or panning.

7.6.2 Input Handling and Managing OS Windows

GLFW is responsible for processing user input on keyboard or mouse, and for creating and managing an OS window on which our visualization tool will be rendered on and shown to the user. GLFW has been written specifically for these use cases and works together smoothly with OpenGL contexts in a portable fashion.

Chapter 8

Conclusion

Looking back at the initial goal of trying to show the exploration process of GenMC's verification algorithm in its natural, two dimensional tree-structured form, a lot of smart tricks have been applied to get from the standard execution of TruSt, to extracting the exploration tree structure during TruSt's exploration, to then serialize the data into only visualization-relevant textual graph blocks written in a compact log, and finally rebuilding and visualizing the exploration tree via the visualization tool. Additionally, the tool has interactive elements (e.g., expandable/ collapsible child subtrees) and navigation (zoom and panning) to make the overall visualization of the exploration tree more comprehensible and hopefully easier for to learn and understand the exploration process for researchers and inexperienced users of GenMC.

A problematic implementation detail of the visualization tool should be addressed in future work, as currently a parentHash field, to be able to retrieve the parent-child relationship correctly for revisits, is augmented to revisit objects, which defies the goal of a fully independent visualizer operating alongside GenMC.

In the end, the visualization is somewhat of a success, as it can potentially be a helpful tool when debugging the GenMC verification process or could be used for educational purposes, to teach researchers and inexperienced users the verification algorithm applied by GenMC.

Bibliography

[1] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. A practitioner's guide to relaxed memory models. *ACM Computing Surveys*, 2011.

[2] F. Brendel. Exploring the immediate mode gui concept for graphical user interfaces. In *Proceedings of the GI Conference on Graphics Interface*. Canadian Human–Computer Communications Society, 2022.

[3] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[4] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Truly stateless, optimal dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 2022.

[5] Michalis Kokologiannakis and Viktor Vafeiadis. Genmc: A model checker for weak memory models. In *Computer Aided Verification (CAV 2021)*. Springer, 2021.

[6] ocornut (Dear ImGui Contributors). About the imgui paradigm. https://github.com/ocornut/imgui/wiki/About-the-IMGUI-paradigm, 2021.

[7] ocornut (Dear ImGui Contributors). Backends in dear imgui. https://github.com/ocornut/imgui/wiki/Backends, 2025.

[8] E.M. Reingold and J.S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, 1981.

[9] Dave Shreiner, Graham Sellers, John Kessenich, and Bill Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley.

[10] The GLFW Development Team. Glfw – an opengl framework for cross-platform window and input management. https://www.glfw.org/, 2025.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten schriftlichen Arbeit. Eine der folgenden zwei Optionen ist **in Absprache mit der verantwortlichen Betreuungsperson** verbindlich auszuwählen:

☒ Ich erkläre hiermit, dass ich die vorliegende Arbeit eigenverantwortlich verfasst habe, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge der Betreuungsperson. Es wurden keine Technologien der generativen künstlichen Intelligenz[1] verwendet.

☐ Ich erkläre hiermit, dass ich die vorliegende Arbeit eigenverantwortlich verfasst habe. Dabei habe ich nur die erlaubten Hilfsmittel verwendet, darunter sprachliche und inhaltliche Korrekturvorschläge der Betreuungsperson sowie Technologien der generativen künstlichen Intelligenz. Deren Einsatz und Kennzeichnung ist mit der Betreuungsperson abgesprochen.

Titel der Arbeit:

Visualizing explorations in GenMC

Verfasst von:
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Lugic	Valon

Ich bestätige mit meiner Unterschrift:
- Ich habe mich an die Regeln des «Zitierleitfadens» gehalten.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu und vollständig dokumentiert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Eigenständigkeit überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Rheinfelden, 24.12.2025	Valon

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie grundsätzlich gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

[1] Für weitere Informationen konsultieren Sie bitte die Webseiten der ETH Zürich, bspw. https://ethz.ch/de/die-eth-zuerich/lehre/ai-in-education.html und https://library.ethz.ch/forschen-und-publizieren/Wissenschaftliches-Schreiben-an-der-ETH-Zuerich.html (Änderungen vorbehalten).