A VSCode Extension for GenMC

Supervisor: Prof. Dr. Michalis Kokologiannakis
Co-supervisor: Matthias Roshardt
Student: Lia Stratan

1 Introduction

Developers often heavily rely on IDEs such as VSCode or CLion when programming. While such environments provide substantial support for syntax highlighting, on-the-fly compilation, and refactoring, they typically do not come with an automated way to test concurrent code. Concurrency bugs often escape normal testing because their manifestation depends on rare interleavings and weak memory behaviors. Under weak memory, the hardware and compiler may reorder or delay the visibility of reads and writes across threads, so executions that are impossible under sequential consistency can still occur in practice. Model checking can systematically explore these behaviors, but it is typically used from the command line and produces outputs that require manual interpretation.

This project aims to simplify testing of concurrent programs and concurrent data structures by embedding GenMC [2] into VSCode and converting its results into navigable, developer-friendly feedback. Users specify a correctness condition to check (e.g., linearizability, assertion violations, etc.), and the extension invokes the GenMC model checker to explore relevant executions. For linearizability checking of concurrent data structures, the extension relies on GenMC's specification-based checking flow based on Relinche [1]. In linearization mode, users annotate which methods implement abstract ADT operations (e.g., enqueue, dequeue, push, pop) and may additionally mark candidate linearization points inside these methods (e.g., a successful CAS or pointer update that commits the operation). The extension then reports the outcome in an IDE-friendly way, including a clear PASS/FAIL summary, a dedicated counterexample viewer, and editor-level feedback (e.g., highlighted lines) that links counterexample events back to source code.

Such support will substantially decrease the amount of testing code developers have to write, while also providing strong correctness guarantees.

2 Modes of action

Randomized exploration

This mode is designed for fast bug-finding through random sampling of the program's state space. In contrast to exhaustive verification, GenMC uses randomness (controlled by a seed) to make choices whenever multiple exploration alternatives are available. This includes scheduling decisions (e.g., when multiple threads are runnable, which one is picked next), and weak-memory nondeterminism such as selecting among alternative read-from (rf) candidates and coherence order (co) resolutions allowed by the chosen memory model. Different seeds can lead to different thread interleavings as well as different weak-memory behaviors, and a fixed seed makes the overall randomized exploration reproducible. If a particular randomized run triggers a bug, recording the seed lets you rerun with a specified seed and reliably reproduce the counterexample.

Users can choose between running with a fixed seed or running without specifying the seed, and the extension will remember the seed used so that any discovered failure can be replayed.

Beyond single runs, a common “fuzzing” pattern is to execute a fuzzy loop over many seeds (e.g., repeatedly invoking GenMC with different seed values), which is useful because concurrency bugs are often rare and interleaving-dependent—each seed samples a different schedule, so many seeds increase coverage of the space of possible thread interleavings and improve the odds of quickly hitting a problematic execution while still keeping runs lightweight compared to full exhaustive exploration.

If a bug is found, the tool shows the first counterexample trace and allows the user to rerun with the same seed/configuration.

Besides the option for specifying a seed of a fuzzy loop run over multiple seeds, users can specify a specific memory model such as Sequential Consistency, TSO, Release-Acquire, RC11 (default).

Verification

Intended for systematic checking. GenMC’s verification mode performs full (exhaustive) exploration of the program’s behaviors under the selected memory model, systematically enumerating relevant executions to either prove the checked property holds under the memory model or to produce a concrete counterexample when it does not. Users can choose whether they want an estimate of the state-space size before starting verification, and can stop it if it takes too long.

Linearizability checking

Intended for concurrent data structures such as queues and stacks. Users provide a specification file that defines the intended behavior of the ADT operations. Under weak memory, multiple specification variants may be available to capture different synchronization/visibility guarantees. Users annotate which methods implement abstract ADT operations (e.g., enqueue, dequeue, push, pop) and may additionally mark candidate linearization points inside these methods (e.g., a successful CAS or pointer update that commits the operation). The extension parses/validates these annotations, then runs GenMC’s linearizability checking flow. Reporting highlights failures as non-linearizable histories and links operations and/or relevant internal steps back to source lines.

3 Delivarables

3.1 Minimum goals

- Implement a plugin that highlights errors in user code by invoking GenMC. The plugin needs to accept user parameters indicating e.g., whether to perform random or exhaustive exploration, as well as the properties to check for.
- Design a nice error reporting mechanism. At the very least, this mechanism should provide the counterexample found by GenMC, but it would be even better if it could also highlight relevant lines in user code.

3.2 Extension Goals

- GenMC assumes certain properties of the program under test (e.g., no data-non-determinism, data independence for linearizability checking). Modify the plugin so that these assumptions are automatically enforced, and relevant errors are provided to the user.
- Investigate whether GenMC’s linearizability-checking algorithm can be optimized if the user provides certain hints (e.g., annotations).

4 14-Week Work Plan

Week 1: Minimal VSCode extension.

- Create a minimal VSCode extension that can run a single command which invokes GenMC on a single file as a child process and shows the raw output in a VSCode Output Channel.

Week 2: Configure details for GenMC run.

- Add configurations such as file path, mode (randomized/verification), memory model, seed option.
- Add more commands to the command pallete.

Week 3: Add cancellation for long runs and improve UI.

- Add cancellation that terminates the GenMC process reliably.
- Add indicator (maybe in the status bar) to show progress/success/failure.

Week 4: Parse GenMC output into structured results.

- Implement a parser producing a counterexample representation in the case of FAILED.
- Create a panel that displays the counterexample.
- Perhaps add a machine readable output format to GenMC.

Week 5: Diagnostics in the Problems panel and highlighting of relevant lines.

- Map parsed errors to VSCode diagnostics, including severity and message.
- If file:line locations are available, highlight those lines.

Week 6 & 7: CMake integration to GenMC.

- Add a feature to GenMC through compile-commands.json to allow it to automatically detect the correct build flags, required header files, etc. from projects that build with CMake, so that the user doesn't have to provide those manually.

Week 8: Linearization annotations.

- Define an annotation scheme for identifying ADT operations (e.g., enqueue/dequeue/push/pop) and optionally marking candidate linearization points.
- Implement annotation parsing in the extension and use the extracted mapping to generate an adapter layer (wrappers/macros) so the generic GenMC harness and the existing spec files can be reused without requiring users to rename their functions.

Week 9: Display non-linearizable histories.

- Extend parsing/reporting to show non-linearizable histories.

Week 10: Check and report data non-determinism.

- Integrate the existing algorithm for data non-determinism and expose it through the extension.

Week 11: Check data independence assumptions for linearizability.

- Modify the data-nondeterminism algorithm to be able to use for data independence check.

Week 12 & 13: Linearization optimization using hints.

- Implement a performance measurement mechanism for the linearization.
- Add a toggle to run linearization with hints enabled/disabled.
- Modify the linearization algorithm to make use of the hints from the user annotations.

Week 14: Project presentation.

- Prepare the project presentation.
- (Optionally) Write a a blog post on the group website.

References

[1] Pavel Golovin, Michalis Kokologiannakis, and Viktor Vafeiadis. Relinche: Automatically checking linearizability under relaxed memory consistency. *Proceedings of the ACM on Programming Languages*, 9(POPL):2090–2117, 2025.

[2] Michalis Kokologiannakis and Viktor Vafeiadis. Genmc: A model checker for weak memory models. In *International Conference on Computer Aided Verification*, pages 427–440. Springer, 2021.