Marrying Miri and GenMC

Master's Thesis Project Description

Patrick Muntwiler

Thesis Supervisor: Michalis Kokologiannakis
Co-Supervisor: Ralf Jung

Abstract

Miri is an interpreter for Rust. It executes the program strictly according to the rules of the formal abstract semantics of Rust. As such, it can detect most kinds of Rust undefined behavior (UB) , and is usually used to test whether a certain piece of Rust code has UB. Miri works by simulating the Rust program step by step, checking all the UB rules at each step. For example, it checks whether values are well-formed or whether a memory access creates a data race.

Even though Miri can run concurrent programs, it only explores a single execution at a time. However, as concurrent programs are highly non-deterministic (e.g., the order in which threads are scheduled may differ for each program run), this means that bugs that only occur in some executions might be missed. While it is possible to repeatedly run Miri to explore the different ways a concurrent program can be run, this is quite inefficient.

This project aims to use techniques from model checking to make Miri explore this search space in a more efficient way, and make it better at detecting concurrency bugs. Specifically, we aim to combine Miri and GenMC, a stateless model checker for concurrent programs that currently targets C/C++. GenMC consists of two main components: (1) an interpreter, which executes the program, and (2) the model-checking algorithm, which repeatedly calls the interpreter to generate a representative subset of the program executions, and influences which thread is scheduled and where reads are reading from. (These two components can be thought of as "coroutines", each one calling the other.) The goal of this thesis is to combine Miri with GenMC's model-checking algorithm.

Approach

The two involved tools (Miri and possibly also the GenMC model checker) will need to be modified/extended to allow them to communicate and provide each other whatever information is required by the other tool. Miri would basically be taking over the part that is done by the existing interpreter in GenMC, and interpret a given program with the concurrent schedule selected by GenMC.

GenMC has more restrictions on the code it can handle compared to Miri, so one challenge will be to make the combination work for code using a large set of language features. Miri might need to inform GenMC about certain values such as results of intercepted function calls (e.g., for random values). Making this work might require changes to some core GenMC structures.

For the implementation, one issue will be the language barrier; Miri is written in Rust, GenMC is written in C++. This means that there will have to be a communication layer between the two tools that translates between their respective representations.

The resulting tool should be able to operate in both an exhaustive and a sampling mode.

Finally, there is the question of the deployment of the final tool. The changes to Miri should not affect its existing capabilities that do not make use of the new features. Preferably, using the new features shouldn't require a recompilation of Miri, only different runtime parameters. Some options for achieving this would be to also compile the GenMC model-checker when compiling Miri, and to then statically link to it. This would allow for only one combined bundle to be deployed, but it would have negative effects on both compile times and program file sizes. Another option would be to dynamically link to a separate installation of GenMC, which would keep the tools more separated, but would introduce other difficulties such as finding the GenMC executables reliably and to correctly handle version updates to either tool without breakage. The possible options will have to be evaluated and one of them implemented.

Finally, once everything is done, the changes should be merged back into the upstream repositories.

Evaluation

Once the tool is implemented, it should be evaluated on an extensive set of benchmarks, consisting of synthetic benchmarks and/or real-world Rust code. The details about this will be worked out during the project.

Core Goals

- The primary objective of this thesis is to create a working stateless model checker for Rust programs. The tool should have two modes of operation:
    - *Exhaustive exploration of the execution space*: Every possible concurrent execution of a given program will be checked.
    - *Sampling mode*: Since the execution space can be very large, the tool should be able to run in a "sampling mode", where random samples of the state-space are explored.
- The resulting tool should be evaluated on a comprehensive set of benchmarks, to measure its scalability and effectiveness.

Extension Goals

- The integration of Miri and GenMC should be done in a way that does not require a fork but can be integrated upstream.
- After the basic functionality is in place, there are multiple ways to improve the resulting tool:
    - Analyse the performance characteristic, check where bottlenecks exist.
    - Improve performance:
        - Use the performance analysis to reduce bottlenecks in the code.
        - If possible and worth doing, make use of multi-threading for the analysis.
        - Try to reduce the number of explored executions, e.g. by making use of symmetry reduction or other optimizations mentioned in this paper on GenMC (section 5, section 7). Some of these optimizations would have to be implemented in Miri, others are implemented in the GenMC model-checker and would require Miri to provide the appropriate hints to work.
    - Improve the usability of the tool (Deployment/installation, setup, usage, debugging/error reporting, documentation)
- Another possible extension goal is to compare how the resulting tool compares to existing tools with similar capabilities. Two candidates for this are the existing frontend for GenMC based on the LLVM interpreter lli (potentially with some Rust specific tweaks to make it more practical) or Loom (concurrency testing tool for Rust code) in terms of:
    - (Concurrency) bugs they can detect or are missed.
    - Usability of the tools:
        - Effort for using the tools, such as required code changes
        - Supported subset of language features
    - Performance of the analysis
- A third extension goal is to apply the resulting tool to some widely used concurrent Rust libraries to evaluate/improve the tool further (and potentially detect (and fix) concurrency issues in those libraries).

Approximate Work Schedule

Task Description	**Time estimation (Max: 6 months ~= 26 weeks)**
Reading into theory and code of both tools. Collecting information required for analysis by each tool	2 – 4 weeks
Implementing changes to both tools	8 – 12 weeks
Evaluation of resulting tool on set of benchmarks	2 – 5 weeks
Extension goal: Performance analysis and performance improvements	2 – 6 weeks
Extension goal: Comparison to similar tools	2 – 3 weeks
Extension goal: Use the created tool to test existing concurrent Rust libraries and possibly fix issues	2 – 3 weeks
Writing final report	4 – 6 weeks