GenMC: A Generic Model Checker for C Programs

Contents

1 Introduction **2**

2 Basic Usage **2**
- 2.1 A First Example . . . 2
- 2.2 Reducing the state space of a program with `assume()` statements . . . 3
- 2.3 Handling Infinite Loops . . . 3
- 2.4 Error Reporting . . . 4

3 Command-line Options **5**

4 Supported APIs **5**
- 4.1 Supported `stdio` API . . . 5
- 4.2 Supported `stdlib` API . . . 6
- 4.3 Supported `pthread` API . . . 6
- 4.4 Supported SV-COMP [1] API . . . 6

5 Contact **6**

1 Introduction

GenMC is a model checker that is an instantiation of the algorithm described in [2] for programs written under the RC11 [3] memory model. GenMC works for C programs that use `C11` atomics and the `pthread` library for concurrency. It uses a very effective dynamic partial order reduction technique, and can verify safety properties in a sound, complete, and optimal fashion.

GenMC works at the level of LLVM Intermediate Representation (LLVM-IR) and uses `clang` to translate C programs to LLVM-IR. This means it can miss some bugs that are removed during the translation to LLVM-IR, but it is guaranteed to encounter at least as many bugs as the compiler backend will encounter.

GenMC should compile on Linux and Mac OSX provided that the relevant dependencies are installed.

2 Basic Usage

A generic invocation of GenMC looks like the following:

```
genmc [OPTIONS] -- [CFLAGS] <file>
```

In the above command, `OPTIONS` include several options that can be passed to GenMC (see Section 3 for more details), and `CFLAGS` are the options that one would normally pass to the C compiler. If no such flags exist, the `--` can be omitted. Lastly, `file` should be a C file that uses the `stdatomic.h` and `pthread.h` APIs for concurrency.

Note that, in order for GenMC to be able to verify it, `file` needs to meet two requirements: finiteness and data-determinism. Finiteness means that all tests need to have finite traces, i.e., no infinite loops (these need to be bounded; see Section 2.3). Data-determinism means that the code under test should be data-deterministic, i.e., not perform actions like calling `rand()`, performing actions based on user input or the clock, etc. In other words, all non-determinism should originate either from the scheduler or the employed (weak) memory model.

As long as these requirements as satisfied, GenMC will detect safety errors, races on non-atomic variables, as well as some memory errors (e.g., double-free error). Users can provide safety specifications for their programs by using `assert()` statements.

2.1 A First Example

Consider the following program, demonstrating the Message-Passing (MP) idiom:

```c
/* file: mp.c */
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>
#include <stdbool.h>
#include <assert.h>

atomic_int data;
atomic_bool ready;

void *thread_1(void *unused)
{
        atomic_store_explicit(&data, 42, memory_order_relaxed);
        atomic_store_explicit(&ready, true, memory_order_release);
        return NULL;
}

void *thread_2(void *unused)
{
        if (atomic_load_explicit(&ready, memory_order_acquire)) {
                int d = atomic_load_explicit(&data, memory_order_relaxed);
                assert(d == 42);
        }
        return NULL;
}

int main()
{
        pthread_t t1, t2;

        if (pthread_create(&t1, NULL, thread_1, NULL))
                abort();
        if (pthread_create(&t2, NULL, thread_2, NULL))
                abort();

        return 0;
}
```

In order to analyze the code above with GenMC, we can use the following command:

```
genmc mp.c
```

with which GenMC will yield the following result:

```
Number of complete executions explored: 2
Total wall-clock time: 0.02s
```

As can be seen, GenMC will explore two executions: one where $ready = data = 0$, and one where $ready = data = 1$.

2.2 Reducing the state space of a program with `assume()` statements

In some programs, we only care about what happens when certain reads read certain values of interest. That said, GenMC, being a model checker, will explore all the consistent values values for each read of interest, as well as all consistent options for all reads after the reads of interest, leading to the exploration of an exponential number of executions that are not of interest.

To alleviate this problem, GenMC supports the `__VERIFIER_assume()` function (similar to the one specified in [1]). This function takes an integer argument (possibly the value read from a read), and only continues the execution if the argument is non-zero.

For example, let us consider the MP program from the previous section, and suppose that we are only interested in verifying the assertion in cases where the first read of the second thread reads 1. We can use an `assume()` statement to achieve this, as shown below:

```c
/* file: mp-assume.c */
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>
#include <stdbool.h>
#include <assert.h>

void __VERIFIER_assume(int);

atomic_int data;
atomic_bool ready;

void *thread_1(void *unused)
{
        atomic_store_explicit(&data, 42, memory_order_relaxed);
        atomic_store_explicit(&ready, true, memory_order_release);
        return NULL;
}

void *thread_2(void *unused)
{
        int r = atomic_load_explicit(&ready, memory_order_acquire);
        __VERIFIER_assume(r);
        if (r) {
                int d = atomic_load_explicit(&data, memory_order_relaxed);
                assert(d == 42);
        }
        return NULL;
}

int main()
{
        pthread_t t1, t2;

        if (pthread_create(&t1, NULL, thread_1, NULL))
                abort();
        if (pthread_create(&t2, NULL, thread_2, NULL))
                abort();

        return 0;
}
```

Note that the `__VERIFIER_assume()` function has to be declared. Alternatively, one can include the <genmc.h> header, that contains the declarations for all the special function that GenMC offers (see Section 4).

If we ran GenMC on the `mp-assume.c` program above, we get the following output:

```
Number of complete executions explored: 1
Number of blocked executions seen: 1
Total wall-clock time: 0.02s
```

As can be seen, GenMC only explored one full execution (the one where $r = 1$, while the execution where $r = 0$ was blocked, because of the `assume()` statement.

We note that, while the usage of `assume()` does not make any practical difference in this small example, this is not the case in programs where there are a lot of (global) accesses after the `assume()` statement.

2.3 Handling Infinite Loops

As mentioned in the beginning of this section, all programs that GenMC can handle need to have finite traces. That said, many programs of interest do not fulfill this requirement, because, for example, they have some infinite loop. GenMC offers two solutions for such cases, depending on the type of the loop.

For simple spin loops, like the one shown below, GenMC automatically transforms them into `assume()` statements:

```
while (!condition)
        ;
```

The `condition` should be a simple condition (e.g., a load from a global variable), and the body of the loop should have no side-effects. In cases where `condition` is a complex expression, or has side-effects (e.g., if it is a compare-and-exchange instruction), GenMC will *not* transform the loop into an `assume()` statement.

For infinite loops with side effects, one has to use the `-unroll=N` command-line option (see Section 3). This option bounds all loops so that they are executed at most `N` times. Naturally, in this case, any verification guarantees that GenMC provides hold up to that bound.

Finally, note that the loop-bounding happens at the LLVM-IR level, which means that the loops there may not directly correspond to loops in the C code (depending on the enabled compiled optimizations, etc).

2.4 Error Reporting

In the previous sections, saw how GenMC verifies the small MP program. Let us now proceed with an erroneous version of this program, in order to show how GenMC reports errors to the user.

Consider the following variant of the MP program below, where the store to `ready` in the first thread is now performed using a relaxed access:

```
/* file: mp-error.c */
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>
#include <stdbool.h>
#include <assert.h>

atomic_int data;
atomic_bool ready;

void *thread_1(void *unused)
{
        atomic_store_explicit(&data, 42, memory_order_relaxed);
        atomic_store_explicit(&ready, true, memory_order_relaxed);
        return NULL;
}

void *thread_2(void *unused)
{
        if (atomic_load_explicit(&ready, memory_order_acquire)) {
                int d = atomic_load_explicit(&data, memory_order_relaxed);
                assert(d == 42);
        }
        return NULL;
}

int main()
{
        pthread_t t1, t2;

        if (pthread_create(&t1, NULL, thread_1, NULL))
                abort();
        if (pthread_create(&t2, NULL, thread_2, NULL))
                abort();

        return 0;
}
```

This program is buggy since the load from `ready` no longer synchronizes with the corresponding store, which in turn means that the load from `data` may also read 0 (the initial value), and not just 42.

Running GenMC on the above program, we get the following outcome:

```
Error detected: Safety violation!
Event (2, 2) in graph:
<-1, 0> main:
        (0, 0): B
        (0, 1): M
        (0, 2): M
        (0, 3): TC [forks 1] L.30
        (0, 4): Wna (t1, 1) L.30
        (0, 5): TC [forks 2] L.32
        (0, 6): Wna (t2, 2) L.32
        (0, 7): E
<0, 1> thread_1:
        (1, 0): B
        (1, 1): Wrlx (data, 42) L.12
        (1, 2): Wrlx (ready, 1) L.13
        (1, 3): E
<0, 2> thread_2:
        (2, 0): B
        (2, 1): Racq (ready, 1) [(1, 2)] L.19
```

```
       (2, 2): Rrlx (data, 0) [INIT] L.20

Assertion violation: d == 42
Number of complete executions explored: 1
Total wall-clock time: 0.02s
```

GenMC reports an error and prints some information relevant for debugging. First, it prints the type of the error, then the execution graph representing the erroneous execution, and finally the error message, along with the executions explored so far and the time that was required.

The graph contains the events of each thread along with some information about the corresponding source-code instructions. For example, for write events (e.g., event (1, 1)), the access mode, the name of the variable accessed, the value written, as well as the corresponding source-code line are printed. The situation is similar for reads (e.g., event (2, 1)), but also the position in the graph from which the read is reading from is printed.

Note that there are many different types of events. However, many of them are GenMC-related and not of particular interest to users (e.g., events labeled with 'B', which correspond to the beginning of a thread). Thus, GenMC only prints the source-code lines for events that correspond to actual user instructions, thus helping the debugging procedure.

Finally, when more information regarding an error are required, two command-line switches are provided. The `-dump-error-graph=<file>` switch provides a visual representation of the erroneous execution, as it will output the reported graph in DOT format in `<file>`, so that it can be viewed by a PDF viewer. Finally, the `-print-error-trace` switch will print a sequence of source-code lines leading to the error. The latter is especially useful for cases where the bug is not caused by some weak-memory effect but rather from some particular interleaving (e.g., if all accesses are `memory_order_seq_cst`), and the write where each read is reading from can be determined simply by locating the previous write in the same memory location in the sequence.

3 Command-line Options

A full list of the available command-line options can by viewed by issuing `genmc -help`. Below we will describe the ones that are most useful when verifying user programs.

`-wb`	Perform the exploration based on the $\mathtt{porf}$ equivalence partitioning (default).
`-mo`	Perform the exploration based on the $\mathtt{po} \cup \mathtt{rf} \cup \mathtt{mo}$ equivalence partitioning.
`-unroll=<N>`	All loops will be executed at most N times.
`-dump-error-graph=<file>`	Outputs an erroneous graph to file `<file>`.
`-print-error-trace`	Outputs a sequence of source-code instructions that lead to an error.
`-disable-race-detection`	Disables race detection for non-atomic accesses.
`-program-entry-function=<fun_name>`	Uses function `<fun_name>` as the program's entry point, instead of `main()`.
`-disable-spin-assume`	Disables the transformation of spin loops to `assume()` statements.

4 Supported APIs

Apart from C11 API (defined in `stdatomic.h`) and the `assert()` function used to define safety specifications, below we list supported functions from different libraries.

4.1 Supported `stdio` API

The following functions are supported from `stdio.h`:

```
int fclose(FILE *)

int fflush(FILE *)

FILE *fopen(const char *, const char *)

int printf(const char *, …)
```

```
int fprintf(FILE *, const char *, …)
size_t fwrite(const void *, size_t, size_t, FILE *)
```

Note that, in this case, said support is provided only for programs to compile, and is not meant to provide functional substitutes for the actual functions of `stdio.h`. (In addition, reading from files can violate the data non-determinism requirement of stateless model checking.) That said, functions like `printf()` should execute normally on Linux machines.

4.2 Supported `stdlib` API

The following functions are supported from `stdlib.h`:

```
void abort(void)
int abs(int)
int atoi(const char *)
void free(void *)
void *malloc(size_t)
```

4.3 Supported `pthread` API

The following functions are supported from `pthread.h`:

```
int pthread_create (pthread_t *, const pthread_attr_t *, void *(*) (void *), void *)
int pthread_join (pthread_t, void **)
pthread_t pthread_self (void)
void pthread_exit (void *)
int pthread_mutex_init (pthread_mutex_t *, const pthread_mutexattr_t *)
int pthread_mutex_lock (pthread_mutex_t *)
int pthread_mutex_unlock (pthread_mutex_t *)
int pthread_mutex_trylock (pthread_mutex_t *)
```

4.4 Supported SV-COMP [1] API

The following functions from the ones defined in SV-COMP [1] are supported:

```
void __VERIFIER_assume(int)
int __VERIFIER_nondet_int(void)
```

Note that, since GenMC is a stateless model checker, `__VERIFIER_nondet_int()` only ”simulates” data non-determism, and does actually provide support for it. More specifically, the sequence of numbers it produces for each thread, remains the same across different executions.

5 Contact

For feedback, questions, and bug reports please send an e-mail to michalis@mpi-sws.org.

References

[1] SV-COMP. Competition on software verification (sv-comp), 2019. [Online; accessed 27-March-2019].

[2] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, New York, NY, USA, 2019. ACM.

[3] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *PLDI 2017*, pages 618–632, New York, NY, USA, 2017. ACM.