GenMC: A Generic Model Checker for C Programs

Contents

1 Introduction

GenMC is a model checker that is an instantiation of the respective algorithm [2] for programs written under the RC11 [3] and IMM [4] 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 4 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 SV-COMP [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:

```
/* 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 5).

If we run 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 4). 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 Tool Features

3.1 Available Memory Models

By default, GenMC verifies programs under RC11. However, apart from RC11, GenMC also supports IMM. The difference between these memory models (as far as allowed outcomes are concerned) can be seen in the following program:

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

atomic_int x;
atomic_int y;

void *thread_1(void *unused)
{
        int a = atomic_load_explicit(&x, memory_order_relaxed);
        atomic_store_explicit(&y, 1, memory_order_relaxed);
        return NULL;
}

void *thread_2(void *unused)
{
        int b = atomic_load_explicit(&y, memory_order_relaxed);
        atomic_store_explicit(&x, 1, memory_order_relaxed);
        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;
}
```

Under RC11, an execution where both $a = 1$ and $b = 1$ is forbidden, whereas such an execution is allowed under IMM. To account for such behaviors, GenMC tracks dependencies between program instructions thus leading to a constant overhead when verifying programs under models like IMM.

3.1.1 A Note on Language Memory Models vs Hardware Memory Models

RC11 is a language-level memory model while IMM is a hardware memory model. Subsequently, the verification results produced by GenMC for the two models should be interpreted somewhat differently.

What this means in practice is that, when verifying programs under RC11, the input file is assumed to be the very source code the user wrote. A successful verification result under RC11 holds all the way down to the actual executable, due to the guarantees provided by RC11 [3].

On the other hand, when verifying programs under IMM, the input file is assumed to be the assembly code run by the processor (or, more precisely, a program in IMM's intermediate language). And while GenMC allows the input file to be a C file (as in the case of RC11), it assumes that this C file corresponds to an assembly file that is the result of the compilation of some program in IMM's language. In other words, program correctness is not preserved across compilation for IMM inputs.

3.2 Race Detection and Memory Errors

For memory models that define the notion of a race, GenMC will report executions containing races erroneous. For instance, under RC11, the following program is racy, as there is no happens-before between the write of x in the first thread and the non-atomic read of x in the second thread (even though the latter causally depends on the former).

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

atomic_int x;

void *thread_1(void *unused)
{
        atomic_store_explicit(&x, 1, memory_order_relaxed);
        return NULL;
}

void *thread_2(void *unused)
{
        int a, b;

        a = atomic_load_explicit(&x, memory_order_relaxed);
        if (a == 1)
                b = *((int *) &x);
        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;
}
```

Additionally, for all memory models, GenMC detects some memory races like accessing memory that has been already freed, accessing dynamic memory that has not been allocated, or freeing an already freed chunk of memory.

Race detection can be completely disabled by means of `-disable-race-detection`, which may yield better performance for certain programs.

3.3 Lock-Aware Partial Order Reduction (LAPOR)

For programs that employ coarse-grained locking schemes LAPOR [5] might greatly reduce the state space and thus the verification time. For instance, consider the following program where a lock is used (overly conservative) to read a shared variable:

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

#ifndef N
# define N 2
```

```
#endif

pthread_mutex_t l;
int x;

void *thread_n(void *unused)
{
	pthread_mutex_lock(&l);
	int r = x;
	pthread_mutex_unlock(&l);
	return NULL;
}

int main()
{
	pthread_t t[N];

	for (int i = 0; i < N; i++) {
		if (pthread_create(&t[i], NULL, thread_n, NULL))
			abort();
	}

	return 0;
}
```

Running GenMC on the program above results in the following outcome:

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

As expected, as the value of N increases, the executions of the program also increase in an exponential manner.

However, if we run GenMC with `-lapor` on the same program, we get the following output:

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

LAPOR leverages the fact that the contents of the critical sections of the threads commute (i.e., the order in which the critical sections are executed does not matter), and only explores 1 execution for all values of N.

We note that for programs where no further reduction in the state space is possible, LAPOR can be cause a polynomial slowdown.

3.4 System Calls and Persistency Checks (Persevere)

Since v0.5, GenMC supports the verification programs involving system calls for file manipulation like `read()` and `write()`. In addition, using `-persevere` GenMC can verify persistency properties of such programs. Below we discuss some details that are important when it comes to verifying programs that involve file manipulation.

3.4.1 Consistency of File-Manipulating Programs

As a first example consider the program below, where a file "`foo.txt`" is first populated by `main`, and then concurrently read and written by two threads at offset 0:

```
/* file: file-rw.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdatomic.h>
#include <pthread.h>
#include <assert.h>

void *thread_1(void *fdp)
{
	int fd = *((int *) fdp);
	char buf[8];

	buf[0] = buf[1] = 1;
	int nw = pwrite(fd, buf, 2, 0);
	return NULL;
}

void *thread_2(void *fdp)
{
	int fd = *((int *) fdp);
	char buf[8];

	int nr = pread(fd, buf, 2, 0);
	if (nr == 2)
		assert((buf[0] == 0 && buf[1] == 0) || (buf[0] == 1 && buf[1] == 1));
	return NULL;
}

int main()
{
        pthread_t t1, t2;
        char buf[8];

        int fd = open("foo.txt", O_CREAT|O_RDWR, 0);

        buf[0] = buf[1] = 0;
        int nw = write(fd, buf, 2);
        assert(nw == 2);

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

        if (pthread_join(t1, NULL))
                abort();
        if (pthread_join(t2, NULL))
                abort();

        return 0;
}
```

One property we might be interested in in the above program is whether the reading thread can see any other (intermediate) state for the file apart from `00` and `11`. Indeed, as can be seen below, running GenMC on the program above produces an example where the assertion is violated.

```
Error detected: Safety violation!
[...]
Assertion violation: (buf[0] == 0 && buf[1] == 0) || (buf[0] == 1 && buf[1] == 1)
Number of complete executions explored: 1
Total wall-clock time: 0.03s
```

When including headers like `stdio.h` or `unistd.h`, GenMC intercepts calls to `open()`, `read()`, `write()`, and other system calls defined in these header files, and models their behavior. Note that these header files are also part of GenMC so, in general, only the functionality described in Section 5 from said header files can be used in programs.

Note that only constant (static) strings can be used as filenames when using system calls. The filenames need not exist as regular files in the user's system, as the effects of these system calls are modeled, and not actually executed. Thus, it is in general preferable if the contents of the manipulated files maintain a small size across executions.

3.4.2 Persistency of File-Manipulating Programs

In addition to checking whether safety properties of file-manipulating programs with regards to consistency are satisfied (as described above), GenMC can also check whether some safety property with regards to persistency (under `ext4`) is satisfied. This is achieved through Persevere, which can be enabled with `-persevere`.

For example, let us consider the program below and suppose we want to check whether, after a crash, it is possible to observe only a part of an append to a file:

```
/* file: pers.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdatomic.h>
#include <pthread.h>
#include <assert.h>
#include <genmc.h>

#include <fcntl.h>
#include <sys/stat.h>

void __VERIFIER_recovery_routine(void)
{
        char buf[8];
        buf[0] = 0;
        buf[1] = 0;

        int fd = open("foo.txt", O_RDONLY, 0666);
        assert(fd != -1);

        /* Is is possible to read something other than {2,2} ? */
        int nr = pread(fd, buf, 2, 3);
        if (nr == 2)
                assert(buf[0] == 2 && buf[1] == 2);
        return;
}

int main()
{
        char buf[8];

        buf[0] = 1;
        buf[1] = 1;
        buf[2] = 1;

        int fd = open("foo.txt", O_CREAT|O_TRUNC|O_RDWR, S_IRWXU);
        write(fd, buf, 3);

        __VERIFIER_pbarrier();

        write(fd, buf + 3, 2);

        close(fd);

        return 0;
}
```

In the program above, the `__VERIFIER_pbarrier()` call ensures that all the file operations before it will be considered as "persisted" (i.e., having reached disk) in this program. The function `__VERIFIER_recovery_routine()` is automatically called by GenMC and contains the code to be run by the recovery routine, in order to observe the post-crash disk state.

In this case, by issuing `genmc -persevere pers.c` we observe that partly observing the append is indeed possible under `ext4`, as can be seen below.

```
Error detected: Recovery error!
[...]
Assertion violation: buf[0] == 2 && buf[1] == 2
Number of complete executions explored: 2
Total wall-clock time: 0.08s
```

For this program in particular, this property is violated due to the default block size (which is 2 bytes), and the nature of appends in the default data ordering mode of `ext4` (`data=ordered`).

In general, such parameters of `ext4` can be tuned via the `--block-size` and `--journal-data` switches (see Section 4 and `genmc -help` for more information). GenMC currently assumes a sector size of 1 byte.

4 Command-line Options

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

`-rc11`	Perform the exploration under the RC11 memory model
`-imm`	Perform the exploration under the IMM memory model
`-wb`	Perform the exploration based on the porf equivalence partitioning (default).
`-mo`	Perform the exploration based on the po ∪ rf ∪ mo equivalence partitioning.
`-lapor`	Enable Lock-Aware Partial Order Reduction (LAPOR)
`-persevere`	Enable `ext4` persistency checks (Persevere)
`-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.

5 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.

5.1 Supported `stdio`, `unistd` and `fcntl` API

The following functions are supported for I/O:

```
int printf(const char *, ...)

int open (const char *, int , mode_t)

int creat (const char *, mode_t)

off_t lseek (int, off_t, int)

int close (int)

ssize_t read (int, void *, size_t)

ssize_t write (int, const void *, size_t)

ssize_t pread (int, void *, size_t, off_t)

ssize_t pwrite (int, const void *, size_t, off_t)

int truncate (const char *, off_t)

int link (const char *, const char *)

int unlink (const char *)

int rename (const char *, const char *)

int fsync (int)

void sync (void)
```

Note that the functions above are modeled and not actually executed, as described in Section 3.4.

5.2 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)
```

5.3 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 *)
```

5.4 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 nondetermism, and does actually provide support for it. More specifically, the sequence of numbers it produces for each thread, remains the same across different executions.

6 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 *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.

[4] Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis. Bridging the gap between programming languages and hardware weak memory models. *Proc. ACM Program. Lang.*, 3(POPL):69:1–69:31, January 2019.

[5] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Effective lock handling in stateless model checking. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.