GenMC: A Generic Model Checker for C Programs

Contents

1 Introduction

GenMC is a stateless model checker for programs written under the SC [2], TSO [3], RA[4], RC11 [5] and IMM [6] memory models. GenMC verifies safety properties of C programs that use C11 atomics and the pthread library for concurrency. It employs an effective dynamic partial order reduction technique [7, ,kokologiannakis2022:trust] that is sound, complete and optimal.

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 (see README.md).

2 Basic Usage

A generic invocation of GenMC resembles 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 underlying (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
```

GenMC explores 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, by default, GenMC will explore all possible values for all program loads, possibly leading to the exploration of an exponential number of executions.

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 (e.g., the value read from a load), 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. Of course, while the usage of `assume()` does not make any practical difference in this small example, this is not always the case: generally, using `assume()` might yield an exponential improvement in GenMC's running time.

Finally, note that, when using GenMC under memory models that track dependencies (see Section 3.1), an `assume()` statement will introduce a control dependency in the program code.

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 three solutions for such cases.

First, GenMC can automatically perform the ”spin-assume” transformation for a large class of spinloops. Specifically, as long as a spinloop completes a full iteration with no visible side effects (e.g., stores to global variables), GenMC will cut the respective execution. For instance, consider the following simple loop:

```
int r = 0;
while (!atomic_compare_exchange_strong(&x, &r, 1))
        r = 0;
```

Since this loop has no visible side-effects whenever it completes a full iteration, GenMC will not explore more than one execution where the loop fails (the execution where the loop fails will be reported as a blocked execution). The ”spin-assume” transformation has proven to be very effective for a wide range of loops; for more details on whether it applies on a specific loop, please see [8].

Finally, for infinite loops with side effects, we can use the `-unroll=N` command-line option (see Section 4). This option bounds all loops so that they are executed at most `N` times. In this case, any verification guarantees that GenMC provides hold up to that bound. If you are unsure whether you should use the `-unroll=N` switch, you can try to verify the program and check whether GenMC complains about the graph size (`-warn-on-graph-size=<N>`). If it does, there is a good chance you need to use the `-unroll=N` switch.

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 other models like SC and IMM. The difference between these memory models (as far as allowed outcomes are concerned) can be seen in the following program:

```c
/* 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 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 [5].

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).

```c
/* 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 Barrier-Aware Model Checking (BAM)

GenMC v0.6 comes with built-in support for `pthread_barrier_t` functions (see Section 5) via BAM [9]. As an example of BAM in action, consider the following program:

```c
/* file: bam.c */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>
```

```
#include <genmc.h>
#include <assert.h>

#ifndef N
# define N 2
#endif

pthread_barrier_t b;
atomic_int x;

void *thread_n(void *unused)
{
        ++x;
        pthread_barrier_wait(&b);
        assert(x == N);
        return NULL;
}

int main()
{
        pthread_t t[N];

        pthread_barrier_init(&b, NULL, N);
        for (int i = 0u; 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 output:

```
Number of complete executions explored: 2
Total wall-clock time: 0.01s
```

As can be seen, GenMC treats `barrier_wait` calls as no-ops, and they do not lead to any additional explorations. (The two executions explored correspond to the possible ways in which `x` can be incremented).

However, if we disable BAM by means of the `-disable-bam` switch, get get the following output:

```
Number of complete executions explored: 4
Number of blocked executions seen: 4
Total wall-clock time: 0.01s
```

Note that while BAM can lead to the exploration of exponentially fewer executions, it can only be used if the result of the `barrier_wait` is not used. If it is, then using `-disable-bam` is necessary, as GenMC currently does not enforce this limitation.

3.4 State-Space Bounding

Under SC, GenMC can bound the state-space exploration using either preemption bounding [10] and round-robin bounding [11].

For instance, in the following program, running GenMC with `--sc --bound=0 --bound-type=context` will avoid exploring executions that have one or more (preemptive) context-switches.

```
/* file: bound.c */
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>

atomic_int x;

void * thread_1(void * unused)
{
    atomic_store(&x, 1);
    atomic_store(&x, 2);
    return NULL;
}

void * thread_2(void * unused)
{
    atomic_load(&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;
}
```

To guarantee that no execution within the bound is missed, some executions that exceed the bound might also be explored, and are reported appropriately:

```
Number of complete executions explored: 3 (1 exceeded bound)
```

The default bounding type (round), on the other hand, only explores executions within the given bound. For instance, when running GenMC with `--sc --bound=0`, only the single SC execution that can be obtained with zero rounds (i.e., in one go) using a left-to-right round-robin scheduler will be explored.

```
Number of complete executions explored: 1
```

Note that, while the round-robin bound does not explore executions that exceed the limit, the number of executions grows more rapidly as the bound increases (compared to context bounding).

3.5 Symmetry Reduction

GenMC also employs an experimental symmetry reduction mechanism [12], which is beneficial to use when threads are running the same code.

For instance, in the following program, GenMC explores only one execution instead of 6.

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

atomic_int x;

void *thread_n(void *unused)
{
	++x;
	return NULL;
}

int main()
{
	pthread_t t1, t2, t3;

	if (pthread_create(&t1, NULL, thread_n, NULL))
		abort();
	if (pthread_create(&t2, NULL, thread_n, NULL))
		abort();
	if (pthread_create(&t3, NULL, thread_n, NULL))
		abort();

	return 0;
}
```

In order for symmetry reduction to actually take place, the spawned threads need to share exactly the same code, have exactly the same arguments, and also there must not be any memory access (at the LLVM-IR level) between the spawn instructions.

To make GenMC use symmetry reduction, one can use the primitive `__VERIFIER_spawn_symmetric(fun, tid)` (defined in `genmc.h`), the last argument of which is the thread identifier of the last (previously spawned) symmetric predecessor.

3.6 Checking Liveness

GenMC can also check for liveness violations in programs with spinloops. Consider the following simple program:

```
/* file: liveness.c */
#include <stdlib.h>
#include <pthread.h>
#include <stdatomic.h>

atomic_int x;

void *thread_1(void *unused)
{
	while (!x)
		;
	return NULL;
}

int main()
{
	pthread_t t1;

	if (pthread_create(&t1, NULL, thread_1, NULL))
		abort();
```

```
        return 0;
}
```

Since there are no writes to x, the loop in thread_1 above will never terminate. Indeed, running GenMC with -check-liveness produces a relevant error report:

```
Error detected: Liveness violation!
Event (1, 4) in graph:
<-1, 0> main:
        (0, 0): B
        (0, 1): TC [forks 1] L.19
        (0, 2): E
<0, 1> thread_1:
        (1, 0): B
        (1, 1): LOOP_BEGIN
        (1, 2): SPIN_START
        (1, 3): Rsc (x, 0) [INIT] L.9
        (1, 4): BLOCK [spinloop]

Non-terminating spinloop: thread 1
Number of complete executions explored: 0
Number of blocked executions seen: 1
Total wall-clock time: 0.07s
```

The -check-liveness switch will automatically check for liveness violations in all loops that have been captured by the spin-assume transformation (see 4).

3.7 Checking Linearizability

GenMC implements the Relinche algorithm [13] for checking (bounded) linearizability. This algorithm requires using GenMC with the ”Most Parallel Client” (MPC). Such a client for queues and stacks is already provided in GenMC's test suite (e.g., tests/correct/relinche/queue/mpc.c).

To use the client e.g., for a queue, we have to provide a queue implementation that defines the following methods: init_queue(), enqueue(), dequeue() and clear_queue(). For instance, an implementation of a Herlihy-Wing queue can be seen below:

```
/* file: hw-queue.c */
#include <stdatomic.h>
#include <assert.h>
#include <genmc.h>
#include <stdbool.h>

#define MAX_NODES       0xff

typedef struct _queue_t {
        _Atomic(int) tail;
        _Atomic(unsigned int) nodes[MAX_NODES] ;
} queue_t;

void init_queue(queue_t *q, int num_threads)
{
}

void clear_queue(queue_t *q, int num_threads)
{
}

void enqueue(queue_t *q, unsigned int val)
{
        int i = atomic_fetch_add_explicit(&q->tail, 1, release);
        assert(i + 1 < MAX_NODES);
        atomic_store_explicit(&q->nodes[i + 1], val, release);
}

bool dequeue(queue_t *q, unsigned int *ret)
{
        bool success = false;
        while (!success) {
                int tail = atomic_load_explicit(&q->tail, acquire);
                for (int i = 0; i <= tail; ++i) {
                        if (atomic_load_explicit(&q->nodes[i], acquire) == 0)
                                continue;
#ifdef BUG /* Linearizability bug */
                        unsigned int v = atomic_exchange_explicit(&q->nodes[i], 0, acquire);
#else
                        unsigned int v = atomic_exchange_explicit(&q->nodes[i], 0, acq_rel);
#endif
                        if (v != 0) {
                                *ret = v;
                                success = true;
                                break;
                        }
                }
                __VERIFIER_assume(success);
        }
        return *ret;
}
```

To check correctness of the above implementation, we first have to provide a specification. GenMC can produce such a specification file from a reference implementation, but we can also use one of the predefined specification files (e.g., `tests/correct/queue/queue_spec_rc11.in`) as follows:

```
genmc -rc11 -disable-mm-detector --check-lin-spec=spec.in -- -DRTN=2 -DWTN=2 -include hw-queue.c mpc.c"
```

Doing so with the above implementation and spec file, will check for linearizability of all clients with two dequeue and two enqueue operations. GenMC produces the following output:

```
GenMC v0.10.3 (LLVM 16.0.6)
Copyright (C) 2024 MPI-SWS. All rights reserved.

*** Compilation complete.
*** Transformation complete.
Tip: Estimating state-space size. For better performance, you can use --disable-estimation.
*** Estimation complete.
Total executions estimate: 33 (+- 37)
Time to completion estimate: 0.03s
*** Verification complete.
No errors were detected.
Number of complete executions explored: 6
Number of blocked executions seen: 10
Number of checked hints: 1
Relinche time: 0.00s
Total wall-clock time: 0.03s
```

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.

`-sc`	Perform the exploration under the SC memory model
`-tso`	Perform the exploration under the TSO memory model
`-ra`	Perform the exploration under the RA memory model
`-rc11`	Perform the exploration under the RC11 memory model (default)
`-imm`	Perform the exploration under the IMM memory model
`-nthreads=<N>`	Perform verification concurrently (using `N` threads)
`-disable-instruction-caching`	Disables a caching optimization that may help runtime by sacrificing memory
`-disable-bam`	Disables Barrier-Aware Model-checking (BAM)
`-check-liveness`	Check for liveness violations in spinloops
`-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 *, ...)
```

Note that these functions are not guaranteed to work properly in all scenarios.

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)

void *aligned_alloc(size_t, 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_trylock (pthread_mutex_t *)

int pthread_mutex_unlock (pthread_mutex_t *)

int pthread_mutex_destroy (pthread_mutex_t *)

int pthread_barrier_init (pthread_barrier_t *, const pthread_barrierattr_t *, unsigned)

int pthread_barrier_wait (pthread_barrier_t *)

int pthread_barrier_destroy (pthread_barrier_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.kokologiannakis@inf.ethz.ch.

References

[1] SV-COMP. Competition on software verification (SV-COMP), 2019.

[2] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, September 1979.

[3] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-TSO. In *TPHOLs 2009*, pages 391–407. Springer, 2009.

[4] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In *POPL 2016*, pages 649–662. ACM, 2016.

[5] 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.

[6] 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.

[7] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *PLDI 2019*, New York, NY, USA, 2019. ACM.

[8] Michalis Kokologiannakis, Xiaowei Ren, and Viktor Vafeiadis. Dynamic partial order reductions for spinloops. In *FMCAD 2021*, pages 163–172. IEEE, 2021.

[9] Michalis Kokologiannakis and Viktor Vafeiadis. BAM: Efficient model checking for barriers. In *NETYS 2021*, LNCS. Springer, 2021.

[10] Iason Marmanis, Michalis Kokologiannakis, and Viktor Vafeiadis. Reconciling preemption bounding with dpor. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *TACAS 2023*, pages 85–104, Cham, 2023. Springer.

[11] Iason Marmanis and Viktor Vafeiadis. Optimal bounded partial order reduction. In Alexander Nadel and Kristin Yvonne Rozier, editors, *FMCAD 2023*, pages 86–91. IEEE, 2023.

[12] Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. SPORE: Combining symmetry and partial order reduction. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[13] Pavel Golovin, Michalis Kokologiannakis, and Viktor Vafeiadis. RELINCHE: Automatically checking linearizability under relaxed memory consistency. *Proc. ACM Program. Lang.*, 9(POPL), January 2025.