





GENMC: A Model Checker for Weak Memory Models

Michalis Kokologiannakis^(✉)  and Viktor Vafeiadis 

MPI-SWS, Kaiserslautern, Germany
{michalis,viktor}@mpi-sws.org



Abstract. GENMC is an LLVM-based state-of-the-art stateless model checker for concurrent C/C++ programs. Its modular infrastructure allows it to support complex memory models, such as RC11 and IMM, and makes it easy to extend to support further axiomatic memory models.

In this paper, we discuss the overall architecture of the tool and how it can be extended to support additional memory models, programming languages, and/or synchronization primitives. To demonstrate the point, we have extended the tool with support for the Linux kernel memory model (LKMM), synchronization barriers, POSIX I/O system calls, and better error detection capabilities.

1 Introduction

For any software developer or verification engineer, it is no news that concurrent programming is difficult, that concurrent software is often buggy, and that therefore verification of concurrent programs has attracted a lot of research interest. Within the verification community at least, it is also common knowledge that verification of concurrent programs is challenging because of the huge number of interleavings of the threads comprising a concurrent program.

What has changed in the last decade, however, is the importance of *weak memory consistency* [6, 11, 13, 14, 21, 25, 32, 36, 40, 41] as a key factor contributing to the complexity of concurrent programming. Weak memory models do not simply increase the number of thread interleavings; they also confound programmers, who typically have little intuition about how to reason about the behaviors induced by these additional interleavings.

GENMC is a fully automatic verification tool meant for such programmers. It is a *stateless model checker* (SMC) [23] that can be used to verify bounded clients of intricate concurrent algorithms, such as implementations of synchronization primitives and shared data structures (e.g., queues, sets, and maps). It accepts as input a C/C++ program using C/C++11 atomics and/or the concurrency primitives from the `pthread` library, and reports any data races, assertion violations, or other errors encountered. By default, verification is performed with respect to the RC11 memory model [32], but there are command line options for selecting other models, such as IMM [41] and LKMM [10].

© The Author(s) 2021

A. Silva and K. R. M. Leino (Eds.) CAV 2021, LNCS 12759, pp. 427–440, 2021.

https://doi.org/10.1007/978-3-030-81685-8_20

Since the theory underlying GENMC has already been published elsewhere [28, 29, 31], this paper focuses on the overall design of the tool and on various enhancements implemented in it. Our main design goals of GENMC were:

Generality: The tool should be able to verify programs written in a variety of programming languages with respect to a variety of memory models.

Efficiency: The tool should implement a state-of-the-art SMC algorithm and incorporate further optimizations for common programming patterns.

Usability: The tool should provide useful and readable error messages.

Extensibility: The tool should be easily adaptable to support additional models and synchronization primitives, and to tweak its performance. Extensibility is key to achieving the other goals, since it allows gradual improvements to the tool in terms of coverage, performance, and error detection/reporting.

These goals are achieved by a combination of techniques:

GENMC’s core SMC algorithm [29, 31] is parametric in the choice of the memory model—subject to a few minimal constraints (see Sect. 2).

The implementation is based on LLVM, a versatile intermediate language for multiple programming languages.

GENMC follows a modular architecture minimizing dependencies across components (see Sect. 3), which makes it easy to extend with support for additional memory models (Sect. 4) and synchronization primitives (Sect. 5).

Its architecture contains hooks to provide fast approximate consistency checks, which are exploited by the memory model implementations (see Sect. 4).

GENMC contains a number of optimizations that provide noticeable performance benefits on common workloads (Sect. 7).

GENMC keeps additional metadata so as to present error messages in terms of variables names appearing in the source code (Sect. 6).

GENMC has been applied to a few industrial settings, where it has found bugs and/or verified bounded correctness of concurrent libraries [39].

Related Work. There has been extensive work on SMC, with most tools focusing on sequential consistency [7, 8, 15, 23, 37]. Tools that support weak memory models include CDSHECKER [38] that verifies C/C++11 programs under the original C11 memory model, TRACER [5] that verifies C/C++11 programs under the RA model, RCMC [27] that verifies C programs under RC11 [32], and NIDHUGG [1, 2, 4, 12, 13] that supports SC, TSO, PSO and provides limited support for the POWER and ARMv7 memory models. In contrast to GENMC, which uses the same core algorithm for all memory models, NIDHUGG uses multiple different algorithms depending on the memory model.

There has also been work on adapting SAT/SMT-based bounded model checking (BMC) techniques for weak memory models [9, 17, 22]. DARTAGNAN [22] is a BMC tool that is parametric in the choice of the memory model, as it accepts the memory model as input in the `litmus` format [11].

2 Memory Model Requirements

GENMC’s core algorithm is parametric in the choice of the memory model provided that it can be expressed in an axiomatic way and satisfies a few basic requirements that we describe below.

Axiomatic memory models represent the executions of a concurrent program as *execution graphs* [11] that satisfy a certain consistency predicate. Execution graphs comprise a set of *events* (nodes) that represent the individual memory accesses performed by the program, and some relations on these events (edges). Example relations included in all memory models are the *preserved program order* (**ppo**) and *reads-from* (**rf**) relations: **ppo** relates events in the same thread that are ordered (e.g., by a chain of dependencies or a fence), while **rf** relates writes to reads reading from them.

GENMC can be used to verify programs under such a model as long as the model’s consistency predicate fulfills the following requirements:

No-Thin-Air: In consistent graphs, $\text{ppo} \cup \text{rf}$ should be acyclic. This intuitively means that an event cannot circularly depend on itself.

Prefix-Closedness: Restricting a consistent graph to any $(\text{ppo} \cup \text{rf})$ -prefix-closed subset of its events yields a consistent graph. Prefix-closedness enables the algorithm to construct a consistent graph incrementally.

Extensibility: Adding a $(\text{ppo} \cup \text{rf})$ -maximal event to a consistent graph for some choice of an incoming **rf**-edge preserves consistency. This captures the intuitive idea that executing a program should never get stuck if a thread has more statements to execute. In particular, a read of x should always be able to return the value written by the most recent write to x .

These requirements are satisfied by almost all axiomatic memory models (e.g., TSO [40], PSO [42], Power [11], ARMv7 [11], ARMv8 [21], RC11 [32], IMM [41], LKMM [10]). The only known axiomatic memory model that does not satisfy these requirements is the original formulation of the C/C++11 model [13], which has been criticized for its flaws [32, 43].

Although these requirements cannot be satisfied by more advanced memory models that cannot be defined in an axiomatic fashion (e.g., [14, 24, 25, 33]), there is ongoing work to support such a model.

3 Tool Architecture

Verification with GENMC comprises three stages (cf. Fig. 1, left).

The first stage invokes **clang** to compile the source C/C++ program to LLVM-IR. To accommodate programs written in different languages, GENMC also accepts LLVM-IR as its input, provided that it adheres to certain conventions about thread creation.

The second stage transforms the LLVM-IR code to make verification more effective by replacing spinloops by **assume** statements, bounding infinite loops,

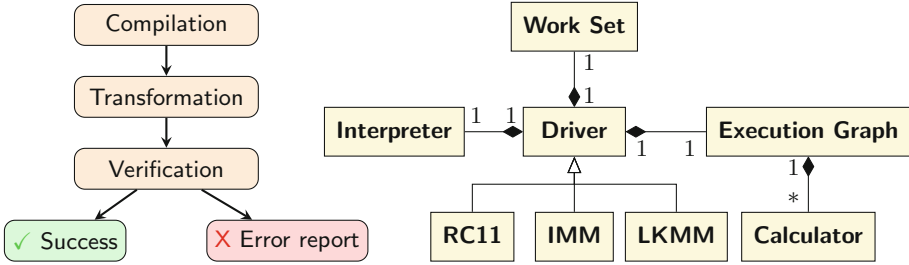


Fig. 1. Overall architecture (left) and dynamic components (right).

and performing sound optimizations, such as dead allocation elimination. It also collects additional debugging information to enable better error reporting.

The third stage invokes the verification procedure, which explores all the executions of the program. If an error is found during this stage, the execution is halted and an error report is produced (see Sect. 6).

The architectural subcomponents of this stage are depicted in Fig. 1 (right). At the center lies the verification *driver*, which owns three independent components: an execution graph, a work set, and an interpreter.

The *execution graph* records the visited execution trace, and has routines for calculating various relation on the graph, such as the happens-before relation. As each memory model comprises different relations, the execution graph contains multiple calculators that are dynamically populated when the graph is created, and the consistency predicate is calculated as a fixpoint of all the selected relations, whenever this is requested by the driver.

The *work set* records alternate options for later exploration, the precise definition of which can depend on the memory model.

The *interpreter* merely executes the user program, notifying the driver each time a “visible” action (e.g., a load/store to shared memory) is encountered. It is directly based on the LLVM interpreter `lli` [35], and is the only part of our code base that heavily depends on LLVM. In turn, the driver modifies accordingly the execution graph, possibly pushes some items to the work set, and returns control back to the interpreter, along with a value that will be used by the interpreter, if necessary (e.g., in the case of a load). In effect, the driver and the interpreter can be thought of as *coroutines* [18]. The interpreter calls the driver whenever it encounters a visible action or finishes running a thread, while the driver monitors execution consistency, schedules the program threads, and discovers alternative exploration options, which are pushed to the work set.

The aforementioned components are all parameterized by the user’s configuration options. The most important of these options is the memory model, which also determines whether dependencies between instructions should be tracked by the interpreter and stored in the execution graph. Another important option is when and how consistency is to be calculated. Since checking consistency at each step can be expensive for some memory models, it is possible to provide an

approximate consistency check to be applied at each step and only perform the full consistency check once an error is detected.

To facilitate memory-model-specific optimizations, the driver is overridden for each memory model. Each instance sets up the (approximate) consistency checks and can provide specialized methods for crucial verification components.

4 Supporting New Memory Models

Adding support for a new memory model entails three basic steps.

First, one has to provide definitions for any memory model primitives that the interpreter should intercept beyond those already supported (i.e., plain memory accesses and C/C++11 atomics). One can either provide a header file mapping these primitives to LLVM-IR instructions or create special event types for them.

Second, one has to provide calculators for the memory model's relations that are not already supported by GENMC. Depending on the memory model, this step may require a variable amount of effort, but it effectively boils down to translating relational calculations into matrix operations.

Third, one can also provide approximations for the consistency checks. Such approximations entail storing crucial information about a memory model's relations as vector clocks (e.g., causally preceding events, for some notion of causality), but deciding what to store is up to the user to decide and encode. Importantly, GENMC's performance depends not only on the calculators provided in the previous step, but also on the effectiveness of the approximations, which quickly filter out inconsistent exploration options. For instance, GENMC's current RC11 driver treats SC accesses as release-acquire (RA) accesses (the consistency of which can be quickly determined), and only checks for full RC11 consistency when an error has been triggered, a heuristic that seems to work well in practice for programs that have both SC and non-SC accesses.

All in all, adding support for a memory model largely depends on the complexity of the model. Adding support for models like SC or RA is trivial, since such accesses are already supported as part of RC11 and IMM. In contrast, adding support for LKMM involved much more work, as we describe below.

4.1 Supporting the Linux Kernel Memory Model (LKMM)

LKMM [10] is a memory model that encompasses a variety of different architectures supported by the Linux kernel. As LKMM differs substantially from RC11 and IMM, supporting it required all steps described above as well as a few other engineering decisions, the most important of which are discussed below.

First, LKMM uses complex constraints for checking consistency of an execution graph. As repeatedly calculating these constraints can be expensive, we designed approximations for them. Unlike most other memory models, LKMM does not define a suitable happens-before relation for checking coherence and detecting races. (Its `hb` relation cannot be used for this purpose.) We thus defined

a custom happens-before relation that can rule out inconsistent executions very quickly, and use it to approximate coherence and race detection checks.

Second, although LKMM dictates that non-atomic accesses (called plain in LKMM’s jargon) only conditionally contribute to `ppo`, we incorporate such accesses in GENMC’s `ppo` (thus arriving at a stronger notion of `ppo`), mostly for technical reasons. Specifically, the calculation of dependencies between only non-plain accesses is difficult because each non-plain access in the source-code level may map to several plain and non-plain accesses in LLVM-IR level.

To increase confidence in our implementation, we ran all litmus tests distributed along with LKMM as part of the Linux kernel (32 tests in total), and compared our results with the results of the HERD [11] memory model simulator. Both tools explored the same number of executions for all tests.

In addition, we extracted some manually written tests from LKMM’s supplementary repository [34] (categories `atomic` and `kernel`). We picked these categories as they contain tests written in C pseudocode (thus easily translatable to C) and do not contain tests with plain accesses, which, as described, GENMC treats slightly differently from what LKMM dictates. In total, these categories amount to another 84 tests, from which we excluded two tests containing unsupported primitives, one test for which HERD did not terminate within 42 h, and three tests that cannot be cleanly translated to C. Out of the remaining 78 tests, GENMC explores the same number of executions for 75 tests. The discrepancies observed in the three remaining tests are due to the different way the two tools produce and calculate dependencies. (In GENMC, control dependencies extend to all subsequent memory accesses of the same thread, whereas in HERD they extend only to the merge point of a conditional statement.)

We note that HERD took about 18 min to run all the above tests, while GENMC needed less than 2 s.

5 Supporting New Languages and Libraries

Supporting additional programming languages is straightforward as long as they can be compiled to LLVM. This was, for example, the case when we extended GENMC to accept C++ (the initial version accepted only C input). All we had to do was to create stub header files for the C++ library, and to extend the interpreter to recognize the memory (de)allocation calls generated by `clang`.

Supporting different runtime environments (e.g., JVM bytecode) requires constructing a new interpreter for the desired runtime system that calls the driver whenever a visible action is encountered. In addition, since the driver and the interpreter communicate using the LLVM type information, it may be necessary to add a translation layer between the interpreter(s) and the driver.

Supporting new concurrency libraries requires localized changes. If the library’s semantics can be implemented in terms of memory accesses, one has to construct an appropriate header file or extend the interpreter to provide the mapping from library calls to the relevant memory access events. If this is not possible and/or if native support for a library is desirable (e.g., for performance

reasons), then the execution graph has to be extended with new kinds of events and the consistency checks have to be adapted accordingly.

Next, we present two such library extensions, one mapping its calls to individual memory accesses, and the other creating new kinds of events.

System Calls. As part of [26], we extended GENMC with support for system calls, such as `open()`, `close()`, `read()` and `write()`, which can be modeled by making multiple primitive calls (reads and writes) to a different address space.

There are two ways one could implement these system calls: either by providing an actual implementation (which would then be compiled to LLVM-IR) or by adding support in the interpreter to internally implement those calls and communicating multiple times with the driver.

We preferred the latter solution because it is more portable. An external implementation would have to be manually ported whenever support for more languages is added. In contrast, the internal implementation needs no change. Further, even if a new interpreter for a different runtime system is added, it should be simple to decouple the system calls from the interpreter, and have the different runtime systems share the infrastructure that handles system calls.

Barriers. N -way barriers are a widely-used synchronization primitive. They have two functions: `barrier_init` and `barrier_wait`. The former initializes a barrier object with the number of threads that will rendezvous at the barrier, while the latter is called every time a thread reaches the barrier. A thread that is calling `barrier_wait` blocks until the initially specified number of threads reaches the barrier, at which point all threads will be simultaneously unblocked, and the barrier value will reset to the one specified with `barrier_init`.

Barriers can be straightforwardly implemented with a shared variable counting the number of threads that have called `barrier_wait`. But doing so yields poor model checking performance. For N threads calling `barrier_wait`, there are $N!$ possible orders in which they can update the shared counter, thus crippling the performance of the tool. Tracking the order of these updates is not only expensive but also completely unnecessary. For many real-world use cases of barriers (e.g., scatter-gather workloads), the order in which different threads reached the barrier is irrelevant, and the thread that reached last unimportant.

We leverage this intuition and provide built-in support for `barrier_init` and `barrier_wait` calls that does not track the relative ordering among `barrier_wait` calls synchronizing with one another, thereby achieving an exponential reduction in verification time. Concretely, in the simple program below where N threads execute `barrier_wait` concurrently, GENMC explores only one execution instead of $N!$ executions:

```
barrier_wait(); || ... || barrier_wait();
```

Our extension is called BAM (Barrier-Aware Model-checking) and is detailed and evaluated in a companion paper [30].

6 Error Detection and Reporting

GENMC detects a number of different kinds of errors: violations of user-supplied regular and persistency assertions, data races, memory errors and simple cases of termination errors. It reports errors by printing an offending execution graph and highlighting the event(s) that caused the violation. Upon request, GENMC can also print a total ordering of the instructions that lead to the violation, or produce the offending execution in the DOT graph description language.

Persistency Errors. To verify persistency properties of programs performing file I/O, we allow user programs to contain a special *recovery routine* [26], which would typically check some invariant over the persisted state.

When such a routine is present, GENMC simulates all possible ways in which the program could have crashed because of a power failure, executing the recovery routine at the end of every such execution. Of course, to avoid the obvious state-space explosion, the simulation of all the possible failures is done in an optimized fashion, driven by the memory accesses of the recovery routine.

The performance of GENMC when verifying persistency properties of programs under the `ext4` filesystem has been evaluated at [26].

Memory Errors. Memory errors refers to accessing uninitialized, unallocated or deallocated memory. In models like RC11 [32], reasoning about memory safety can be tricky at times, as demonstrated by the example below:

$$\begin{array}{l}
 p := \mathbf{alloc}(); \\
 *p :=_{\mathbf{r1x}} 42; \\
 x :=_{\mathbf{r1x}} 1;
 \end{array}
 \parallel
 \begin{array}{l}
 \mathbf{if } x = 1 \mathbf{ then} \\
 \quad a :=_{\mathbf{r1x}} p; \\
 \quad b :=_{\mathbf{r1x}} *a;
 \end{array}$$

This example is erroneous under RC11 because the allocation of p is not guaranteed to have propagated to the second thread by the time it is dereferenced. (Since all accesses are relaxed, there is no synchronization between the threads.)

GENMC also accounts for more complicated scenarios such as p being concurrently freed when accessed, p being freed twice, or p being the address of a local (stack) variable that might not be alive when accessed.

Refining Error Reports. It is often useful to refine the error reporting. For example, in memory models that treat data races as errors (such as RC11), GENMC by default detects data races and reports them as errors. This, however, can be costly in terms of verification time or even prohibit the verification of programs that use compiler/custom primitives to access shared memory, as such programs would almost certainly be considered racy.

To deal with such cases, GENMC provides switches that disable race detection and refine the range of errors that will be reported to the user. Switches of the latter kind are especially useful when dealing with programs that contain system calls. By default, when such system calls fail, GENMC reports an error, which is inconvenient for programs that contain proper error handling, as some

system errors are rather benign (e.g., a file not existing). With the appropriate switch, in case of system errors, an appropriate value is written in `errno`, as dictated by the POSIX standard.

Case Study. We demonstrate the error reporting capabilities of GENMC with a real use case. We consider a flat-combining queue [19] that has been proposed to be ported in Rust’s `crossbeam` library.

This queue serves as a nice case study for a couple of reasons. First, it contains loops that can diverge, and so its verification requires loop bounding, which GENMC can do automatically. Second, it is implemented using compiler primitives for concurrent accesses, and so its verification requires disabling race detection. Third, while experimenting with it, we found it to be buggy.

The error report produced by GENMC can be seen in Fig. 2. The error is quite intricate: it requires three threads to manifest, each of which executes a large number of instructions. The error is due to an ordering bug (relaxed accesses are used instead of release/acquire), which demonstrates the need for model checking tools that handle weak memory models.

```

Error detected: Attempt to read from uninitialized memory!
Event (3, 63) in graph:
<-1, 0> main:
<0, 1> thread_n:
    (1, 18): Urel (cmb.queue, 0) [(0, 36)] L.169: combiner.c
    (1, 19): Urel (cmb.queue, 2565579352) L.169: combiner.c
    (1, 96): Racq (m.msg._meta.next, 2565579416) [(2, 26)] L.228: combiner.c
    (1, 112): Wrlx (cmb.takeover, 2565579416) L.158: combiner.c
<0, 2> thread_n:
    (2, 26): Wrel (m.msg._meta.next, 94798317999592) L.167: combiner.c
<0, 3> thread_n:
    (3, 18): Urel (cmb.queue, 2565579352) [(1, 19)] L.169: combiner.c
    (3, 19): Urel (cmb.queue, 2565579480) L.169: combiner.c
    (3, 50): Rrlx (cmb.takeover, 2565579416) [(1, 112)] L.87: combiner.c
    (3, 63): Racq (m.msg._meta.next, 0) [BOTTOM] L.187: combiner.c
Number of complete executions explored: 2795
Number of blocked executions seen: 6001
Total wall-clock time: 2.12s

```

Fig. 2. An error report by GENMC after removing irrelevant lines.

We note that the error report contains helpful debugging information, such as the names of variables accessed (e.g., `m.msg._meta.next`) and the values read/written. To display this information, GENMC maintains a mapping from addresses to program variables using the additional debugging information collected in the “Transformation” phase.

7 Other Performance Enhancements to GENMC

In this section, we briefly discuss two recent changes to the driver to optimize its performance for certain kinds of programs.

Symmetry Reduction. Many programs, such as the flat-combining queue of Sect. 6, have a symmetric structure: each thread runs the same code. In such cases, many execution graphs are equivalent up to some thread relabeling—a property that is exploited by *symmetry reduction* (SR) [16, 20].

We implemented a simple SR algorithm that detects whether multiple threads with the same code are spawned with no intervening memory accesses, and avoids exploring executions for which a symmetric one (by relabeling such threads) has already been explored. This can yield exponential improvements. For example, a program with N threads incrementing a shared variable atomically has $N!$ executions; employing SR yields only one execution. With SR, the verification time of the corrected flat-combining queue drops from 15 s to 2.5 s.

To further demonstrate the benefits of SR, we measured the performance of GENMC with and without SR on some realistic lock implementations adapted from the literature. The results can be seen in Table 1. All reported times are in seconds, unless mentioned otherwise. We ran both GENMC versions three times for each benchmark, with an increasing number of threads each time (the initial thread number for each benchmark is provided in the second column). As it can be seen, SR leads to a significant performance improvement in all cases.

Table 1. Testing lock implementations (1 h timeout; 4 GB memory limit)

	N	Without SR			With SR		
		N	$N+1$	$N+2$	N	$N+1$	$N+2$
mutex	2	0.02	0.40	41 min	0.03	0.08	164.66
mutex_musl	2	0.01	34.47	OOM	0.01	5.92	OOM
rwlock	2	0.02	0.18	47.34	0.04	0.05	1.94
spinlock	3	0.03	0.08	1.19	0.02	0.03	0.18
ticketlock	4	0.02	0.13	2.35	0.01	0.01	0.01
ttasklock	3	0.06	2.05	38 min	0.08	0.11	33.87
tvalock	3	0.03	0.49	79.68	0.03	0.04	0.36

Lock-Aware Partial Order Reduction. A common problem with locking is that of false sharing, where N threads contend to acquire the same lock even if it is unnecessary for correctness. In such cases, GENMC’s partial order reduction algorithm [29] will explore all $N!$ orders in which the lock can be acquired even though they all lead to the same outcome.

We have implemented *lock-aware partial order reduction* (LAPOR) [28], an enhancement to partial order reduction that does not track ordering among locks unless their critical regions have conflicting accesses, in which case the lock ordering is induced from the ordering among those accesses. With LAPOR, GENMC achieves exponential improvements in lock-based implementations of concurrent libraries that have false sharing, such as search trees with coarse-grained or hand-over-hand locking. LAPOR has been evaluated at [28].

8 Conclusion

We presented GENMC, a state-of-the-art stateless model checker that can be used to verify consistency and persistency properties of C/C++ programs. We described its architecture, and how its modular design can be leveraged to account for new features and memory models. To widen the applicability of GENMC, we have extended it with support for LKMM, basic system calls and additional synchronization primitives. We have also improved its performance with optimizations, such as symmetry reduction and lock-aware partial order reduction that can exponentially decrease its search space.

In the future, we plan to implement a DSL for memory models, so as to make it easier to extend GENMC with new models and quickly tweak their approximation strategies. We are also planning to incorporate further optimizations into the tool to enable more effective verification of lock-free algorithms.

Acknowledgements. We thank the anonymous reviewers for their feedback. This work was supported by a European Research Council (ERC) Consolidator Grant for the project “PERSIST” under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 101003349).

References

1. Abdulla, P.A., Aronis, S., Atig, M.F., Jonsson, B., Leonardsson, C., Sagonas, K.: Stateless model checking for TSO and PSO. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 353–367. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46681-0_28
2. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. In: POPL 2014, pp. 373–384. ACM, New York (2014). <https://doi.org/10.1145/2535838.2535845>
3. Abdulla, P.A., Atig, M.F., Jonsson, B., Lång, M., Ngo, T.P., Sagonas, K.: Optimal stateless model checking for reads-from equivalence under sequential consistency. Proc. ACM Program. Lang. **3**, 150:1–150:29 (2019) <https://doi.org/10.1145/3360576>
4. Abdulla, P.A., Atig, M.F., Jonsson, B., Leonardsson, C.: Stateless model checking for power. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 134–156. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_8
5. Abdulla, P.A., Atig, M.F., Jonsson, B., Ngo, T.P.: Optimal stateless model checking under the release-acquire semantics. Proc. ACM Program. Lang. **2**(OOPSLA), 135:1–135:29 (2018) <https://doi.org/10.1145/3276505>

6. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. *IEEE Comput.* **29**(12), 66–76 (1996)
7. Albert, E., Arenas, P., de la Banda, M.G., Gómez-Zamalloa, M., Stuckey, P.J.: Context-sensitive dynamic partial order reduction. In: Majumdar, R., Kunčák, V. (eds.) *CAV 2017*. LNCS, vol. 10426, pp. 526–543. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63387-9_26
8. Albert, E., Gómez-Zamalloa, M., Isabel, M., Rubio, A.: Constrained dynamic partial order reduction. In: Chockler, H., Weissenbacher, G. (eds.) *CAV 2018*. LNCS, vol. 10982, pp. 392–410. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96142-2_24
9. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) *CAV 2013*. LNCS, vol. 8044, pp. 141–157. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_9
10. Alglave, J., Maranget, L., McKenney, P.E., Parri, A., Stern, A.: Frightening small children and disconcerting grown-ups: concurrency in the Linux kernel. In: *ASPLOS 2018*, pp. 405–418. ACM, Williamsburg, VA, USA (2018). <https://doi.org/10.1145/3173162.3177156>
11. Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.* **36**(2), 7:1–7:74 (2014) <https://doi.org/10.1145/2627752>
12. Aronis, S., Jonsson, B., Lång, M., Sagonas, K.: Optimal dynamic partial order reduction with observers. In: Beyer, D., Huisman, M. (eds.) *TACAS 2018*. LNCS, vol. 10806, pp. 229–248. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_14
13. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. In: *POPL 2011*, pp. 55–66. ACM, Austin, Texas, USA (2011). <https://doi.org/10.1145/1926385.1926394>
14. Chakraborty, S., Vafeiadis, V.: Grounding thin-air reads with event structures. *Proc. ACM Program. Lang.* **3**(POPL), 70:1–70:28 (2019) <https://doi.org/10.1145/3290383>
15. Chalupa, M., Chatterjee, K., Pavlogiannis, A., Sinha, N., Vaidya, K.: Data-centric dynamic partial order reduction. *Proc. ACM Program. Lang.* **2**(POPL), 31:1–31:30 (2017) <https://doi.org/10.1145/3158119>
16. Clarke, E.M., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. *Form. Meth. Syst. Des.* **9**(1/2), 77–104 (1996) <https://doi.org/10.1007/BF00625969>
17. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) *TACAS 2004*. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
18. Conway, M.E.: Design of a separable transition-diagram compiler. *Commun. ACM* **6**(7), 396–408 (1963) <https://doi.org/10.1145/366663.366704>
19. Crossbeam: Flat combining #63. <https://github.com/crossbeam-rs/crossbeam/issues/63>. Accessed 29 Jan 2021
20. Emerson, E.A., Wahl, T.: Dynamic symmetry reduction. In: Halbwachs, N., Zuck, L.D. (eds.) *TACAS 2005*. LNCS, vol. 3440, pp. 382–396. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31980-1_25
21. Flur, S., et al.: Modelling the ARMv8 architecture, operationally: concurrency and ISA. In: *POPL 2016*, pp. 608–621. ACM, St. Petersburg, FL, USA (2016). <https://doi.org/10.1145/2837614.2837615>

22. Gavrilenco, N., Ponce-de-León, H., Furbach, F., Heljanko, K., Meyer, R.: BMC for weak memory models: relation analysis for compact SMT encodings. In: Dillig, I., Tasiran, S. (eds.) CAV 2019. LNCS, vol. 11561, pp. 355–365. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-25540-4_19
23. Godefroid, P.: Model checking for programming languages using VeriSoft. In: POPL 1997, pp. 174–186. ACM, Paris, France (1997). <https://doi.org/10.1145/263699.263717>
24. Jagadeesan, R., Jeffrey, A., Riely, J.: Pomsets with preconditions: a simple model of relaxed memory. Proc. ACM Program. Lang. 4(OOPSLA) (2020) <https://doi.org/10.1145/3428262>
25. Kang, J., Hur, C.-K., Lahav, O., Vafeiadis, V., Dreyer, D.: A promising semantics for relaxed-memory concurrency. In: POPL 2017, pp. 175–189. ACM, Paris, France (2017). <https://doi.org/10.1145/3009837.3009850>
26. Kokologiannakis, M., Kaysin, I., Raad, A., Vafeiadis, V.: PerSeVerE: persistency semantics for verification under ext4. Proc. ACM Program. Lang. 5(POPL) (2021) <https://doi.org/10.1145/3434324>
27. Kokologiannakis, M., Lahav, O., Sagonas, K., Vafeiadis, V.: Effective stateless model checking for C/C++ concurrency. Proc. ACM Program. Lang. 2(POPL), 17:1–17:32 (2017). <https://doi.org/10.1145/3158105>
28. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Effective lock handling in stateless model checking. Proc. ACM Program. Lang. 3(OOPSLA) (2019). <https://doi.org/10.1145/3360599>
29. Kokologiannakis, M., Raad, A., Vafeiadis, V.: Model checking for weakly consistent libraries. In: PLDI 2019, ACM, New York (2019). <https://doi.org/10.1145/3314221.3314609>
30. Kokologiannakis, M., Vafeiadis, V.: BAM: Efficient Model Checking for Barriers. In: NETYS 2021, LNCS, Springer, Heidelberg (2021). <https://plv.mpi-sws.org/genmc>
31. Kokologiannakis, M., Vafeiadis, V.: HMC: Model checking for hardware memory models. In: ASPLOS 2020, pp. 1157–1171. ACM, Lausanne, Switzerland (2020). <https://doi.org/10.1145/3373376.3378480>
32. Lahav, O., Vafeiadis, V., Kang, J., Hur, C.-K., Dreyer, D.: Repairing sequential consistency in C/C++11. In: PLDI 2017, pp. 618–632. ACM, Barcelona, Spain (2017). <https://doi.org/10.1145/3062341.3062352>
33. Lee, S.-H., Cho, M., Podkopaev, A., Chakraborty, S., Hur, C.-K., Lahav, O., Vafeiadis, V.: Promising 2.0: Global optimizations in relaxed memory concurrency. In: Donaldson, A.F., Torlak, E. (eds.) PLDI 2020, pp. 362–376. ACM (2020). <https://doi.org/10.1145/3385412.3386010>
34. McKenney, P.E.: *Automatically generated litmus tests for validation LISA-language Linux-kernel memory models*(2021). <https://github.com/paulmckrcu/litmus>. Accessed: 28 Apr 2021
35. lli - directly execute programs from LLVM bitcode (2003). <https://llvm.org/docs/CommandGuide/lli.html>. Accessed 29 Jan 2021
36. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL 2005, pp. 378–391. ACM (2005). <https://doi.org/10.1145/1040305.1040336>
37. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtii, I.: Finding and reproducing Heisenbugs in concurrent programs. In: OSDI 2008, pp. 267–280. USENIX Association (2008). https://www.usenix.org/legacy/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf

38. Norris, B., Demsky, B.: CDSChecker: Checking concurrent data structures written with C/C++ atomics. In: OOPSLA 2013, pp. 131–150. ACM (2013). <https://doi.org/10.1145/2509136.2509514>
39. Oberhauser, J., et al.: VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models. In: ASPLOS 2021, pp. 530–545. ACM, Virtual, USA (2021). <https://doi.org/10.1145/3445814.3446748>
40. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03359-9_27
41. Podkopaev, A., Lahav, O., Vafeiadis, V.: Bridging the gap between programming languages and hardware weak memory models. Proc. ACM Program. Lang. **3**(POPL), 69:1–69:31 (2019). <https://doi.org/10.1145/3290382>
42. SPARC International Inc., The SPARC architecture manual (version 9). Prentice-Hall (1994)
43. Vafeiadis, V., Balabonski, T., Chakraborty, S., Morisset, R., Zappa Nardelli, F.: Common compiler optimisations are invalid in the C11 memory model and what we can do about it. In: POPL 2015, pp. 209–220. ACM, Mumbai, India (2015). <https://doi.org/10.1145/2676726.2676995>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

