Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Symmetry and Barrier Reduction in GenMC

Bachelor Thesis

Morris Turdo

July 14, 2025

Advisors: Prof. Dr. M. Kokologiannakis, Y. Gu
Department of Computer Science, ETH Zürich

Abstract

GenMC is a stateless model checker that can verify C/C++ programs by enumerating all their executions. It employs various optimizations to reduce the number of possible executions it has to explore, to mitigate the problem of exponential state-space explosion. Symmetry reduction and barrier reduction are two types of optimizations that employ thread symmetry and barrier arrival order, respectively, to avoid exploring executions. Both methods have only been investigated separately and are deployed as SPORE and BAM in GenMC.

This thesis fixes issues with the barrier model and the implementation of BAM used in GenMC. It also explores ideas on combining SR and BR, by using barriers to restore broken symmetry in symmetric threads.

Contents

Chapter 1

Introduction

Concurrent Verification through Stateless Model Checking (SMC) [15] is a tool to test all possible executions of a parallel program for their correctness. Such SMC programs only need polynomial amounts of memory, as they do not need to store a state for every possible execution. However, even state-of-the-art parallel verification algorithms, such as TruSt [18] or in Nidhugg [4, 6] require exponential time, in the number of instructions, to verify a program. This is because for every instruction in the concurrent program, every possible interleaving of the threads has to be taken into consideration. Fortunately, there are methods to reduce the number of thread interleavings that must be explored. We can take two executions and declare them, in a sound way, to be equivalent. Now, we only need to explore one of them to confirm that both behave in a certain way, thereby reducing the state space.

Generic Model Checker (GenMC)[17, 21, 24] is one such SMC verification tool. It employs several of these mentioned methods to reduce the number of executions in addition to its optimal exploration algorithm TruSt. GenMC also includes other features to increase its performance, such as automatic spinloop bounding [22] and including lock-awareness in its exploration algorithm [20]. It is unique in that it allows users to parametrically choose the memory model under which the verification process should take place [24]. Furthermore, there are three important ways in which GenMC (and SMC in general) can reduce the number of explorations, which we have not yet mentioned.

The most general and intuitive approach, although certainly not simple, is called dynamic partial order reduction (DPOR) [13]. The goal is to divide executions into groups that behave in the same way, based on whether their *partial order* changes the execution behavior. This is easy to do for some cases, but a difficult problem to solve optimally [3, 5, 18]. In practice, these groups are created *dynamically* during execution every time a new conflict arises

(e.g., two threads modifying the same variable), additional program traces are explored where the conflicting operations occur in different orders. One easy case where this can be applied can be seen below.

```
x1 = 1;  ||  x2 = 2;  ||  ⋮  ||  xN = N;
```
(ParallelSet)

These N writes are obviously all independent of each other as they all access different variables. Any order of these operations in ParallelSet results in the same outcome, and thus we can group them together to be treated as the same execution behavior.

Another such method, symmetry reduction (SR) [11, 12], exploits the symmetry of threads that execute the same code. It reduces the exploration space by declaring threads interchangeable as in the example below.

```
fai(x);  ||  ⋮  ||  fai(x);
```
(ParallelFai)

The N threads in ParallelFai execute the exact same code (that is, they atomically increment the variable x) and are thus symmetric. A naive SMC implementation would construct every possible interleaving of these threads and thus explore N! many executions. Notice that it does not matter what exact thread executes at what point, as they are interchangeable. For example, having the threads execute from left to right results in the exact same behavior as scheduling them from right to left. We conclude that by exploring one representative execution, we can consider all executions which only change the thread's names to behave in the same way. Notice that in this case, DPOR could not be used to reduce the state space, as the operations would be considered conflicting.

Another method optimizes explorations related to the use of barriers, which can be used in parallel running threads to rendezvous at a specific point in their execution. This can be achieved by using synchronization barriers such as pthread_barrier [1]. Such a barrier forces all participating threads to have arrived before they can all continue their execution. This can be seen in the following example.

```
            barrier_init(b, N);
A[0] = foo();       ||    ||  A[N-1] = foo();
barrier_wait(b);    ||  ⋮ ||  barrier_wait(b);
B[0] = bar(A);      ||    ||  B[N-1] = bar(A);
```
(BarrierArrays)

The main thread first initializes a barrier to N, which means that N threads are required to reach the barrier before they can all continue. There are N threads in BarrierArrays, each of which operates on the same index for two

different arrays. The computations 'bar()', for the second array, depend on the results of the 'foo()' computations, of the first array. A barrier can now be used to ensure that the computations on the first array are fully completed before the computations on the second array are started.

Barrier-Aware Model checking (BAM) [23] takes advantage of the functioning of these *barriers* to *reduce* the number of traces explored in the program. This is why we will also refer to this process as barrier reduction (BR). The main idea is to disregard the order in which the threads arrive at the barrier. In the BarrierArrays program above, we could use DPOR to disregard the interleaving of the writes to the arrays, as they are disjoint. The call to `barrier_wait(b)` would however be noted as a conflict and every possible order of arrivals would be explored. In practice, the order in which the threads arrive at the barrier would not even be observable. We only care that the threads are all at the barrier at the same time and not in what order they arrived. With this in mind, we only need to explore one of the possible interleavings.

We have now briefly seen how these three types of reduction work on their own, but we want to use all three of them in the same verification process. We want the three types of reduction to not hinder one another and beyond this we want them to help each other to perhaps increase their performance even further. BAM has from the beginning been built upon DPOR and is as such well integrated. The fusion of SR and DPOR is more complex and requires more work but has also been tackled in SPORE [19]. The combination of symmetry and barrier reduction has not yet been investigated and will be the subject of this bachelor thesis.

To understand why BR and SR do not combine neatly out of the box, we must first consider again how SR works. In SR, threads can be declared symmetric and as such be used to reduce the number of explorations. Certain types of operations can, however, break this symmetry and leave the rest of the threads' program unsymmetrical. As soon as the threads are not symmetric anymore, we cannot use SR.

The second part we need to understand is that barriers are treated in a special way in the verification process. Typically, barriers use some synchronization primitive, such as locks, and other costly operations in their implementation. But when we enumerate all possible executions, we need to simplify the program as much as possible, as we are chained by the exponential explosion imposed by state exploration. The solution used by BAM is to create a model of the individual barrier components. With this, we can have a relatively simple function to execute when `barrier_wait()` is called for example. The barrier model still adheres to the semantics of a barrier and works in the same way.

The issue with this modeling, or in fact likely with any barrier modeling, is

that it contains operations which break symmetry. Thus, SR is hindered from working whenever we use barriers. Our goal will be to somehow prevent symmetry breaks caused by barriers, and furthermore to try to use barriers to improve the performance of SR.

The interluding problem posed by BAM's modeling, which has to be addressed first, is that it does not fully work as intended. There are cases where the modeling leads to execution explorations that do not happen with real barriers. The majority of this thesis will be dedicated to fixing the modeling and implementing an improved version of it in GenMC. The new implementation will form the basis on which future combinations of SR and BR can be built.

Concretly, we will follow the following structure for the thesis:

§ 2 We give context and present background information, as well as the theoretical foundations on which the rest of the thesis is based.

§ 3 We describe the issues with the barrier modeling used in BAM and its implementation in GenMC. We present an improved barrier model and describe its implementation in GenMC as well as additional improvements to GenMC surrounding the modeling.

§ 4 We describe ideas on how we can restore symmetry after a `barrier_wait()` call and how this can be used to improve SR.

§ 5 We evaluate our contributions to GenMC by comparing versions of it with and without our changes.

Chapter 2

Fast Verification

The inefficiency of pure stateless model checking (SMC) and ways to mitigate the problem will be our first focus. Note that this chapter is intended to establish the theoretical foundation of the thesis and does not present original content. We will first look at how SMC works in general and how this presents challenges. Next, we will look at GenMC, the model checker central to this thesis, and finally, we will see different reduction types that can help improve the performance of SMC, where we will focus on barrier reductions, as they are the most important type for this thesis.

2.1 Model Checking

Early verification techniques for concurrent programs were largely limited to labor intensive proofs, constructed by hand. Due to the lack of scalability imposed by these methods, more efficient techniques emerged, which eventually led to the invention of stateful model checking. The name originates from temporal logic, where early model checkers would try to *check* if a temporal formula was a *model* for the state transition graph of a program. This model checking technique first used the automatic enumeration of all possible executions of a program to prove the program's correctness [10, 16]. Finally, through the adoption of dynamic partial order reduction (DPOR), this led to SMC [15].

SMC has the great advantage, over previous state-tracking model checking techniques, that it uses only polynomial memory. The emerging challenge then is to only explore every possible execution exactly once, without explicitly keeping track of the ones that have already been explored. But even if we manage to explore every possible execution only once, concurrency conflicts can lead to exponential growth in the number of distinct execution traces, resulting in exponential time complexity relative to the number of conflicts. Heuristic approaches try to combat this by limiting the number of

explored executions in some way. As bugs have been found to frequently appear with low numbers of context switches, one approach is to bound the number of allowed preemptive context switches of threads [27]. Another approach is to prioritize explorations that are deemed more likely to expose a bug or complete their exploration in a given CPU budget [8, 9, 28]. We will disregard these and focus on reductions; these give us a theoretically complete approach to making the verification process more efficient.[1] We will see three types of reductions (DPOR, SR, BR) in detail in § 2.3.

A state-of-the-art and one of the most sophisticated model checkers is Generic Model Checker (GenMC). This open source verification tool allows for assertion-safety verification of concurrent C and C++ programs. GenMC also allows for a parametric choice of the memory consistency model used and is in that sense *generic*. The algorithm used is sound, complete, and optimal in accordance with some small conditions on the memory model, which means that every consistent execution is explored exactly once. The model checker additionally employs many optimizations to further improve its performance. [2, 17, 21, 24]

In layman's terms this is to mean that this tool is publicly available on GitHub, is one of the best of it's kind currently available, and might work for your C/C++ program. If you add `assert(cond)` statements to your code with conditions that you want to be true at that point of the program, GenMC will tell you if they always hold or will show you a possible execution that makes one of your assertions fail.

This is the verification tool on which we will base our new barrier modeling and improved barrier reduction implementation (§ 3), as well as the combination of symmetry and barrier reduction (§ 4). GenMC has a lot more features and abilities than we could discuss here, which means that we will have to simplify and cannot go into exact detail for the theoretical parts. The key feature of GenMC that makes it stand out from other stateless model checkers is its ability to work with many different weak memory models. We will not be able to discuss these and assume a sequential consistency (SC) [25] setting while remembering that all of these ideas can be extended for weak memory models.

2.2 Execution Graphs

We will use execution graphs [7] to visualize and reason about reductions, as this is the approach that GenMC uses to reason about executions and in its implementation. Execution graphs are composed of a set of events

[1] The reductions in this chapter also could, and are in practice, combined with heuristic approaches, but we will look at them in isolation and in combination with each other. We will not see them combined with heuristic techniques.

(memory access operations) represented by nodes and some relations on these events represented by edges. We will see an example graph after a formal description, which is mainly adapted from these sources [7, 18, 19, 23].

Definition 2.1 (Event) *An event $e \in$ **Event**, is either the initialization event* init*, or a thread event $\langle t, i, lab \rangle$ where $t \in$ **Tid** is a thread identifier, $i \in$ **Idx** $\triangleq \mathbb{N}$ is a serial number, denoting the index of an event within a thread, and $l \in$ **Lab** is a label that takes one of the following forms:*

- *Read label:* $\mathrm{R}(loc, v)$ *where $loc \in$ **Loc** is the location accessed, $v \in$ **Val** $\triangleq \mathbb{Z}$ is the value read.*
- *Write label:* $\mathrm{W}(loc, v)$ *where $loc \in$ **Loc** is the location accessed, $v \in$ **Val** is the value written*
- *Read-modify-write label:* $\mathrm{RMW}(loc, v_1, v_2)$ *modeling an atomic read-modify-write operation, where $loc \in$ **Loc** is the location accessed, $v_1 \in$ **Val** is the value read, $v_2 \in$ **Val** is the value written*
- *Block label:* block*, denoting a blocked thread*
- *Error label:* error*, denoting a safety violation*

Notice that with this we model atomic and non-atomic integer operations in addition to being able to mark erroneous executions and block a thread if it cannot continue execution at the moment (ex. it is waiting at a barrier). In addition, we assume that the init event sets all variables to 0. We will also use $\langle t, i \rangle$ to denote an event without explicitly mentioning its label. Lastly, let tid, idx, loc, lab, valr, and valw be defined as the functions that return the thread identifier, serial number, location, label, read value, and written value of an event, respectively.

Definition 2.2 (Program Order) *The program order* **po***, is a strict partial order on events that relates every event to the next one in the same thread according to their serial execution.*

$$\mathbf{po} \triangleq \{\langle \text{init}, e \rangle \mid e \in (\mathbf{Event} \setminus \{\text{init}\}\} \cup \{\langle \langle t_1, i_1 \rangle, \langle t_2, i_2 \rangle \rangle \mid t_1 = t_2 \wedge i_1 < i_2\}$$

Definition 2.3 (Well Formed Execution Graph) *A well-formed execution graph $G \in$ **Exec**:*

1. *consists of:*
   - *i a set $G.E$ of events with distinct combinations of a thread identifier and a serial number, that includes a unique initialization event* init *which initializes all locations accessed by the program*
   - *ii a relation $G.\mathbf{rf} \subseteq G.W \times G.R$, called the reads-from relation, that relates each write event to the same-location reads that read from it*

2. *for which holds:*

   i **rf** *only relates writes and reads with matching locations and values, i.e., for every* $\langle w, r\rangle \in G.\mathbf{rf}$ *we have* $w \in G.W$, $r \in G.R$, $\mathsf{loc}(w) = \mathsf{loc}(r)$ *and* $\mathsf{valw}(w) = \mathsf{valr}(r)$

   ii **rf** *is functional on its range, i.e., if* $\langle w_1, r\rangle$, $\langle w_2, r\rangle \in G.rf$ *we have* $w_1 = w_2$

   iii *every read event reads a value, i.e.,* $\forall r \in G.\mathrm{R}.\ \exists w.\langle w, r\rangle \in G.\mathbf{rf}$

$G.\mathrm{E}$ denotes all events and $G.\mathrm{R}, G.\mathrm{W}$ denote all events of the respective types, where RMW operations belong to all three sets. These sets are further restricted by subscripts where, for example, $G.\mathrm{E}_x$ refers to all events that operate on the location x.

Definition 2.4 (Coherence Order) *A relation* **co** *is a coherence order for an execution* G *iff* **co** *is a strict partial order,* $\mathbf{co} \subseteq \bigcup_{l \in \mathbf{Loc}} G.W_l \times G.W_l$, *and for every location* $l \in \mathbf{Loc}$, **co** *is total on* $G.W_l$.

We can use **co** to get the order in which the write events on some location are executed. In a single-threaded SC program, **po** would include **co**.

Definition 2.5 (From-Reads) *from-reads* **fr** *is a relation defined as:*

$$\mathbf{fr} \triangleq \{(a, b) \mid a \neq b \wedge \exists c.(c, a) \in \mathbf{rf} \wedge (c, b) \in \mathbf{co}\}$$

Intuitively, **fr** indicates that a read was performed before the write we are considering and after the previous write (i.e., after the write which is immediately **co**-before).

Definition 2.6 (Sequential Consistency) *G is sequentially consistent iff there is a coherence order* **co** *for G such that the happens-before relation*

$$\mathbf{hb} = \mathbf{po} \cup \mathbf{rf} \cup \mathbf{co} \cup \mathbf{fr}$$

is acyclic.

Now we can look at the example in Fig. 2.1 for an execution, represented as an execution graph, and try to see if it is sequentially consistent.

As there is no cycle in this graph, we know that the corresponding execution is sequentially consistent. We can also notice that the read value is not strictly necessary in the R events, since we can just follow the **rf** to get the read value. The relation **fr** was only defined here for completeness, and we will not use **fr** again for the rest of the thesis, as it is not relevant to our work. We will also not explicitly mention or check sequential consistency, but the reader can assume that all example graphs are SC.

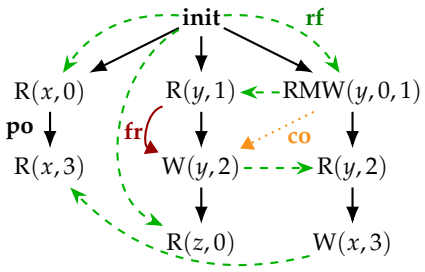

Figure 2.1: Sequential consistent example graph

2.3 Reductions

Reductions improve performance by reducing the number of executions that must be explored while still verifying at least one interleaving per consistent execution. We will use Mazurkiewicz trace equivalence [26] to define classes of equivalent executions to partition all possible interleavings into groups which have the same consistency order and therefore also the same behavior. In our case, this simply means that any two transitions that access the same memory location, and of which at least one is a write instruction, are considered conflicting. Immediately from this, we find that two execution graphs are in the same equivalence class if and only if they have the same execution order for every pair of conflicting instructions.

The first reduction type, dynamic partial order reduction (DPOR), will solve the problem of exploring every execution only once, in addition to reducing the number of explored traces below the number of possible interleavings of all instructions. Symmetry reduction (SR), the second reduction type, will allow us to reduce the number of equivalence classes even further by exploiting the symmetry between threads. The final method will be barrier reduction (BR) and will also be the most important for our purposes. Moreover, we will be more formal and precise when dealing with BR compared to the other two reduction types. In § 3 we will also include some improvements in the implementation and modeling of this method. This section on reductions is mainly based on content from these papers [3, 5, 18, 19, 23, 26].

2.3.1 DPOR

Static partial order reduction (POR) [14] (used in stateful model checking) was originally a process of defining static conditions on when two possible executions have the same behavior. With this, equivalence classes (such as Mazurkiewicz trace equivalence classes) can be created that partition the possible interleavings into groups of executions that have the same consistency order. The main problem with this approach is that dynamic information is mostly disregarded, which results in conservative approximation of indepen-

dence, possibly leading to redundant explorations. This means that multiple equivalent executions will have to be explored as they cannot be recognized as equivalent before starting the verification process.

Dynamic partial order reduction (DPOR) addresses the problem directly by creating equivalence classes on-the-fly. After exploring one sample execution, with possibly arbitrary thread scheduling, every conflict triggers an additional exploration where the two conflicting operations happen in reverse order. This on it's own does not enforce the optimal exploration where every equivalence class is verified through exactly one representative, but there are ways to achieve this [3, 5, 18]. We will not look at how DPOR algorithms work, but will quickly inspect an example of it in action. Consider now the example program W+W+RR with labeled instructions and all execution graphs constructed by DPOR in Fig. 2.2:

```
(#1) x = 1;  ||  (#2) y = 1;  ||  (#3) a = x;
                                  (#4) b = y;
```
(W+W+RR)

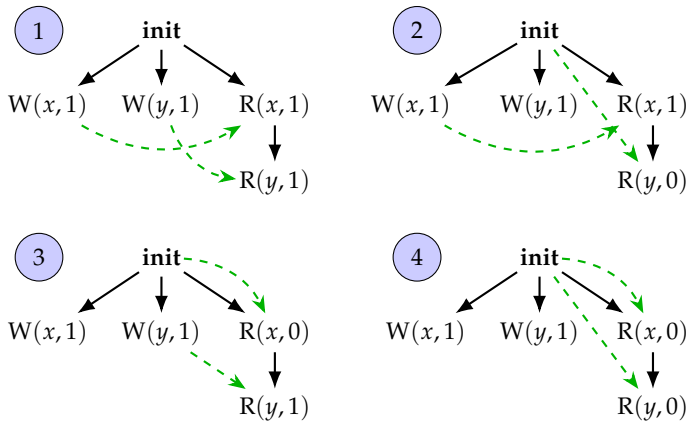

Figure 2.2: Execution Graphs for W+W+RR program

There are $\binom{4!}{2!} = 12$ possible interleavings and only 4 consistent executions, shown in Fig. 2.2. By creating all of the four possible execution graphs, we can verify the behavior of all 12 possible executions. DPOR creates these graphs by starting with one sample execution. In this case, we will start with the execution resulting from scheduling threads from left to right (ltr) which gives us graph ①. During the construction of the graph, a DPOR algorithm would note that there is a conflict between #1 and #3, as well as between #2 and #4. When all instructions have been added to the graph, DPOR starts backtracking and looking at the conflicts. An algorithm might go back and change the order of the two operations in the conflict of #2 and #4 and thus create execution graph ②. Before, when #4 read from #2, #2 must have happened first and when we switch the order, since now #4 happens first, #4

now reads from the initialization event. Through an efficient version of this process, one can construct all possible graphs without having to explicitly check if two executions are equivalent and belong to the same class. This approach can result in an exponential reduction in the time required to verify a program.

2.3.2 SR

Even though optimal DPOR reduces the number of explorations to exactly all consistent executions, it might be surprising that we can still improve and reduce the number of explorations even further. The first of two methods that we will see is symmetry reduction (SR) which works on the simple principle of ignoring thread IDs if the threads have the exact same code. We will now look at how this further improvement is possible and then see how SR is implemented in GenMC.[2]

For now, consider two threads to be symmetric if they run the same code. To see how SR improves DPOR, consider the W+R+R program below, which includes two symmetric threads. See also the four execution graphs that we get in Fig. 2.3.

$(\mathbf{T_1})$: `x = 1;` $\|$ $(\mathbf{T_2})$: `r = x;` $\|$ $(\mathbf{T_3})$: `r = x;` (W+R+R)

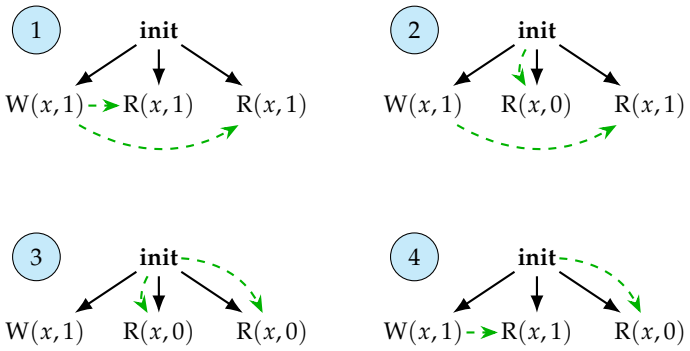

Figure 2.3: Execution Graphs for W+R+R program

We get the four consistent executions that pure DPOR explores. Notice that execution ② and ④ are very similar. The only thing that changes between these executions is the name of the threads. We can get each of the two graphs by changing the names of the threads $\mathbf{T_2}$ and $\mathbf{T_3}$. The name of the thread becomes irrelevant as we do not care which exact thread with which ID executed their code at what point. The result is the same, in the sense

[2]When we talk about SR, we only consider thread-level symmetry, but there also exist other ways of exploiting symmetry in programs, such as internal symmetry in SPORE [19].

that one thread will hold the value 0 and the other will hold the value 1. The state space is reduced by choosing one of the two equivalent executions and discarding the other. This process is only possible for these two executions; the other two do not contribute in SR. In summary, we can explore three of these execution graphs to verify the behavior of all possible executions. We will not see how SR is integrated into an exploration algorithm, but note that SR can lead to an exponential speed-up in programs with high symmetry.

Simplifying symmetry to checking if threads have the same code is not sufficient; we need more elaborate checks. Even if threads have the same code, they might encounter a branch in their execution that depends on some external factor. Through this external factor, the symmetry can be broken. Consider in the example below the two threads that run the exact same program.

```
(T1) :  int id = thread_self();        ||  (T2) :  int id = thread_self();
        if(y == 2)                     ||          if(y == 2)
            x = 3;                     ||              x = 3;
        r = x;                         ||          r = x;
```

Even though the threads do not differ at all in their program, we cannot use SR to reduce the number of executions. Consider the only two consistent executions in Fig. 2.4 where, if you take one of the graphs and change the names of the threads, you do not get the other execution. The `thread_self()` operation breaks the symmetry for the rest of the thread execution. In general, if two threads have the same code, all of their reads will read from the same locations. If two read events, however, read from different write events, the symmetry of the threads is broken at that point. This is because the different reads impose a divergence in the local states of the threads, which poses the risk of influencing the control flow. If the threads' control flow is influenced in a way that they no longer run the same code, they can no longer be considered symmetric.

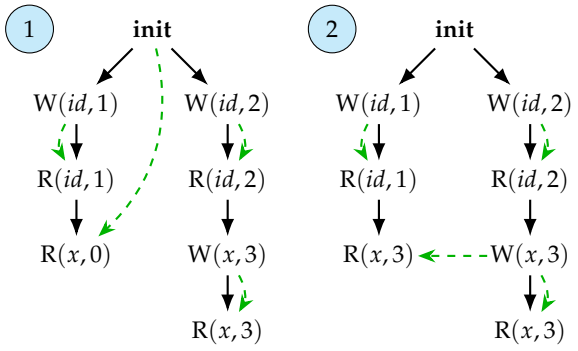

Figure 2.4: Execution Graphs for example program using `thread_self()`

Now we will look at how GenMC handles symmetry. GenMC defines two threads as symmetric if all five of these are true for two threads:

1. both threads have distinct thread IDs
2. both threads have the same parent
3. both threads receive the same input variables when they are created
4. there is no memory access between the creation events of the two threads
5. both threads are created to execute the same function

The first two conditions are quite intuitive. The third and fourth conditions deal with the mentioned issue of reading different values. If two threads execute different operations depending on what value they read from some external source, they behave differently from each other and are clearly not symmetric. Additionally, GenMC forgoes comparing the threads' code and simplifies this check by enforcing that the threads must be created to execute the same function, which is defined in the fifth condition. These conditions do not define a perfect symmetry condition. There might be cases where symmetric threads are not recognized and vice versa. If two threads are symmetric, their symmetry can also still break later on in their execution if their local states diverge (ex. they read different values from a global variable). This point will be very important when we get to the combination of BR and SR in § 4. In short, for now, in the execution of a barrier, the threads break their symmetry because they read some global value for the operation of the barrier. We will then look at how to restore that symmetry.

GenMC also allows a user to define threads as symmetric. In this case, GenMC will still let the symmetry break if some instruction forces it. To create symmetric threads, GenMC provides the `__VERIFIER_spawn_symmetric()` function, which can be used if the `genmc.h` file is included in the program to be verified.

2.3.3 BAM

Barrier Aware Model Checking (BAM) [23], will be the basis for our new barrier modeling, our improvements in its implementation in GenMC, and for the final combination of barrier reduction (BR) and SR.[3] BAM has been developed to solve a very specific problem, namely, using BR to greatly improve the very poor performance of verifiers when encountering barriers.

[3]We use BAM and BR largely interchangeably. Note that BAM [23] is the original concept description of barriers in SMC, where it is presented as creating a DPOR algorithm that is aware of barriers and has specific handling for them. We introduce BR as a description since we show how barriers can be involved in reducing the number of explorations.

Concretely, the issue is that DPOR considers calls to `barrier_wait()` to be conflicting with respect to consistency and must therefore explore every possible order of arrival at the barrier. The order in which threads call wait on a barrier does not affect the execution. As implementations typically also do not report in what order they arrived at the barrier, this is not even observable by the user program.[4] BAM then simplifies the exploration to consider only one arrival order and treat the others as equivalent. We will now first look at why calls to `barrier_wait()` are considered conflicting, then extend our execution graphs to incorporate barriers, consider an example of BAM in action, and finally look at restrictions on when BAM can be used.

In Fig. 2.5, we can see the model used by BAM to simulate the functioning of barriers. For now, assume that this model is a given. We will investigate it in more detail in § 3.1. It becomes immediately clear why DPOR considers calls to `barrier_wait()` to be conflicting. Each of them includes an instruction that writes to the global variable `b`. Recall that by Mazurkiewicz trace equivalence, two instructions are conflicting if they operate on the same memory location and if at least one of the instructions is a write operation. This is clearly the case here. DPOR has to explore every possible interleaving of writing to `b` and therefore has to consider every possible arrival order at a barrier.

```
barrier_init(b, N) :
   b = N
barrier_wait(b) :
   atomic { if (b == 1) then b = N else b = b-1 }
   assume(b == N)
```

Figure 2.5: BAM's modeling of `barrier_init()` and `barrier_wait()`

In pursuit of our goal of exploring only one barrier arrival order, we start by extending our execution graphs to allow for a new barrier event, representing the `barrier_wait()` call. We then create a new relation **sbr** to convey that two of these newly created events are wait calls in the same barrier round.

Definition 2.7 (Barrier-Wait Label) *Barrier-wait label:* $\mathrm{B}(loc)$ *where* $loc \in$ **Loc** *is the barrier location accessed.*

B is a dummy event that is neither a read nor a write. Use $G.\mathrm{B}$ to denote the set of all barrier events in an execution graph G.

We extend the definition of execution graphs Def. 2.3 with an additional relation:

[4] pthread_barrier_wait() for example returns 0 to all threads except one unspecified thread which receives a distinct value when the required number of threads have called the wait function. But even this is typically not used in programs.

Definition 2.8 (Same-Barrier-Round) *Same-barrier-round* **sbr** *is a partial equivalence relation which relates barrier events that synchronize with each other in a rendezvous. It fulfills the following conditions for an execution graph G and a barrier location b initialized with value N:*

- *Events related by $G.\mathbf{sbr}$ act on the same barrier location:* $\forall \langle e_1, e_2 \rangle \in G.\mathbf{sbr}.\ \mathsf{loc}(e_1) = \mathsf{loc}(e_2)$
- *There exists a unique event $w \in G.\mathrm{W}$ such that $\mathsf{lab}(w) = \mathrm{W}(b, N)$, and $\langle w, e \rangle \in G.\mathbf{hb}$ for all $e \in G.\mathrm{B}_b$*
- *There are no more than N barrier events in G which do not appear in $G.\mathbf{sbr}$:* $|G.\mathrm{B}_b \setminus dom(G.\mathbf{sbr})| < N$ *(Where $dom(G.\mathbf{sbr})$ denotes the domain of* **sbr***)*
- *Every barrier round must consist of exactly N barrier events. No barrier event may have a* **po***-successor if it does not participate in a full barrier round:* $\forall e \in G.\mathrm{B}_b.\ |succ_{G.\mathbf{sbr}}(e)| = N \vee succ_{G.\mathbf{sbr}}(e) = succ_{G.\mathbf{po}}(e) = \emptyset$ *(Where $succ_r(e) \triangleq \{e' \mid \langle e, e' \rangle \in r\}$ is the set of successors of e in r)*

We additionally modify the partial-order relation **hb** to include **sbr**, to capture the fact that if an event is **hb**-after one barrier event, it is **hb**-after every barrier event in the same barrier round. We therefore now redefine **hb** as the following (where ; indicates relation composition).

$$\mathbf{hb} \triangleq \mathbf{po} \cup \mathbf{rf} \cup \mathbf{co} \cup \mathbf{fr} \cup \mathbf{sbr}; \mathbf{po} \cup \mathbf{po}; \mathbf{sbr}$$

In Fig. 2.6 we can see three example executions which are not allowed by our previous definitions. We also left out the init event, as it is not relevant for these graphs. In ①, we initialize the barrier location b twice and the second initialization event is not necessarily **hb**-before all events in $G.\mathrm{B}_b$, since the barrier event of the second thread could read from the initial write to b. In ②, even though the second thread correctly blocks its execution, the first thread continues with a read, but there are only $2 < N = 3$ barrier events in the same-barrier-round. The last example ③ contradicts with our new **hb** definition. There exists a **hb** cycle between the barrier events in each thread; this contradicts **hb** being a partial order.

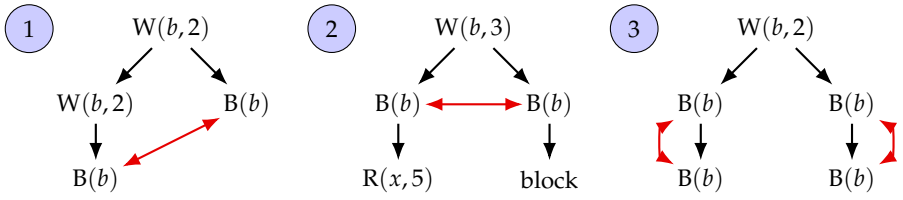

Figure 2.6: Disallowed execution graphs with **sbr**

For an allowed use of **sbr** and a correct example execution graph for the program below, see Fig. 2.7. Here, we can also observe how BAM manages

to reduce the number of explored executions, for this, recall the barrier modeling in Fig. 2.5. ① shows an execution graph following a strict ltr scheduling where the first thread is executed until it terminates, blocks or leads to a safety violation (recall that DPOR would still need to explore all other consistent executions). The first two threads block because the assumed condition of `b == N` was false. In an implementation of BAM, this might be a graph that we get during exploration, but since the third thread reads a new value for b, we would schedule a revisit when writing to b and unblock the first two threads. Here, this would then lead to execution ②, where all assumes read from the third thread. The final execution ③, shows how an exploration would look with our new barrier-specific event implemented. The relations **rf** and **co** are abstracted away, but we include our new **sbr** relation. This leaves us with this graph as the only possible exploration. In conclusion, this is the only possible execution graph which we have to explore, while before, we also had to explore all possible scheduling orders.

```
                    barrier_init(b, 3);
barrier_wait(b); || barrier_wait(b); || barrier_wait(b);
```

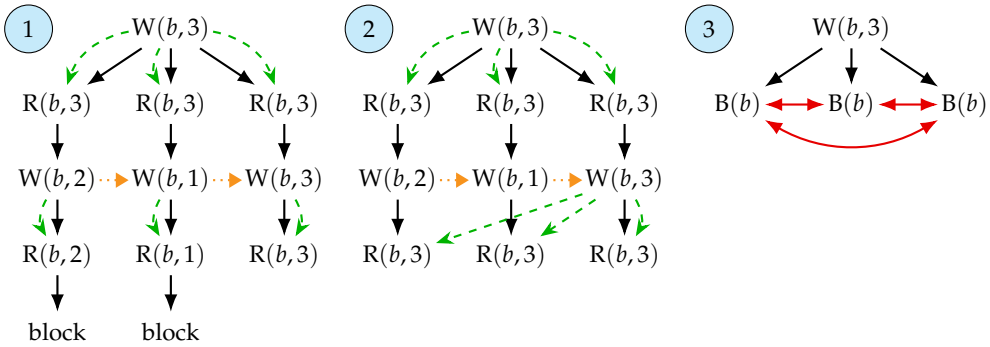

Figure 2.7: Example of BAM

In Def. 2.9 we have a formal description of barrier well-formedness which gives us conditions on when BAM can be used.

Definition 2.9 (Barrier Well-Formedness) *We define an execution graph G as barrier well formed if for all $b \in G.\mathrm{B}$, there are no barrier-wait labels ($G.\mathrm{B}_b = \emptyset$) or if for all $b \in G.\mathrm{B}$ there exist a unique w_0 and a N such that all the following hold:*

1. *There is a unique initialization event:* $w_0 \in (G.\mathrm{W} \setminus \mathsf{init}).\ \mathsf{loc}(w_0) = b$
2. *The initialization event is a plain write:* $N \in \mathbb{N}.\ \mathsf{lab}(w_0) = \mathrm{W}(b, N)$
3. *The initialization event is* **hb***-before all barrier events:* $\forall e \in G.\mathrm{B}_b.\ \langle w_0, e\rangle \in G.\mathbf{hb}$

4. *At most N threads may call the wait function on the barrier location b concurrently:*
   $\forall S \subseteq G.\mathrm{B}_b$ *it holds that* $|S| > \mathsf{valw}(w_0) \implies \exists e, e' \in S.\ \langle e, e' \rangle \in G.\mathbf{hb}$

Barrier well-formedness restrictions preclude the use of barriers if the number of concurrent wait calls is larger than the initialization value N. BAM can therefore not be used in programs like the one below. Such a program is, however, likely not all that useful anyway.

```
                    barrier_init(b, 2);
barrier_wait(b); || barrier_wait(b); || barrier_wait(b);
```

Having a smaller number of concurrent wait calls than the initialization value is still allowed under these conditions, but such a use-case is hard to imagine, as this would indefinitely block the execution of all participating threads. A program like the one below is therefore allowed but is likely useless.

```
          barrier_init(b, 3);
barrier_wait(b); || barrier_wait(b);
```

Chapter 3

Barrier Modeling

Towards our original goal of combining SR and BR in GenMC, we must first investigate their current implementations. For SR, GenMC implements SPORE which combines optimal DPOR and SR in a correct way. The SR implementation can also be improved further; but we will not do so. Our focus will be on BR, which GenMC implements in BAM, using its barrier modeling, which we have seen in § 2.3.3. We will now investigate this model and the implementation thereof in GenMC and conclude that it is not fully adequate for its intended use. Following from this, we will propose a new model that does not have these shortcomings and then describe the exchange of the old implementation in GenMC for our new one. Finally, we will go into some additional changes that we made to GenMC's code around the implementation of our new barrier model.

3.1 Original Model

To simplify and improve the performance of SMC for programs that utilize barriers, it is very useful to model the execution of barrier functions with some simplified code that still captures the semantics of the barrier functionality. Looking first at the barrier model as presented in BAM, we will see it implemented in GenMC and then look at an example case where the model fails.

3.1.1 BAM Implementation

We have already seen the original BAM barrier model, but will now look at it again in detail [23]. First, we explain how an assume is used. The `assume()` is a function used in verification to enforce the conditions passed as an argument. When writing `assume(cond)` we tell the verifier to assume that condition `cond` is true at that point of the execution. The verifier then explores only those executions in which the condition is met. Remember not

to confuse `assume()` with `assert()`. An assertion can be used in verification to check whether a condition always holds, while the assumption tells the verifier to presume that the condition holds.

As shown in Fig. 3.1, BAM models the two most important barrier functions with a counter variable `b`. Initially, the barrier is set to the number of participating threads `N`.[1] Every call to `barrier_wait(b)` first performs an atomic ternary operation where the value of `b` is always decremented, except if the value read from `b` is 1. In this case, the initial value `N` is written at location `b`. Afterwards, the verifier assumes that the barrier variable `b` is again set to `N`.

```
barrier_init(b, N) :
   b = N
barrier_wait(b) :
   atomic { if (b == 1) then b = N else b = b-1 }
   assume(b == N)
```

Figure 3.1: BAM's modeling of `barrier_init()` and `barrier_wait()`

The assume enforces that exactly a multiple of `N` threads must have reached the barrier, as only then will the barrier value be `N`. As we have seen in § 2.3.3, during exploration, the thread is blocked when the condition in the assume is evaluated to false. This is exactly the behavior we want from a barrier; threads that call `barrier_wait()` block until exactly `N` threads have arrived at the barrier, at which point all participating threads can continue their execution.

To now understand how this model is implemented in GenMC, and later understand how our new model is implemented, we must first get to know the architectural components of GenMC. The heart of GenMC is its verification driver. The driver owns three independent components:

- an execution graph where all events and relations are recorded
- a work set of alternate exploration options to be performed later on
- an interpreter based on LLVM which executes the program and informs the driver every time a memory operation is performed which the driver is interested in.

The workflow of verifying a user program then follows this routine:

1. Clang is used to compile the program into LLVM.

[1] We have seen in § 2.3.3, that, in fact, it is also possible to choose `N` larger than the number of participating threads and that this is not very useful to do, which is why we ignore it.

2. Several optimization- and verification-relevant transformations (such as annotations) are performed on the LLVM IR.
3. The interpreter executes the generated LLVM IR and sends relevant information to the driver.
4. The driver receives information on the instructions that the interpreter is performing and stores them in the work set.
5. The driver handles the instructions, records them in the execution graph, and verifies the execution.

The implementation of BAM is based in the interpreter and the driver, of which we present an incomplete list of components here. The list includes all parts of the BAM implementation which we will modify:

- Every time the interpreter encounters a barrier instruction, it tells the driver to handle the relevant stores and reads, in accordance with the barrier model. However, the logic of determining whether and which values are to be stored or read is implemented in the interpreter, where the function is being handled. The implementation also handles the `barrier_destroy()` function, which is not specified in BAM, by storing the value 0 in the barrier variable to mark it as destroyed.
- When the instruction is passed to the driver, the driver checks if the read value means that the thread should block its execution, which, as we know from before, is the case when a different value than `N` is read from the barrier value in the assume.
- Checks in the driver prevent wait calls on uninitialized or destroyed barriers. They also ensure that barriers are initialized once and only with allowed values, i.e., natural numbers without 0.
- The driver optimizes when to revisit the reads. By default, every time a new value becomes available from which an instruction could read, the read operation is revisited to get the new value. When doing this with a barrier, we know that nothing will change when reading the new value until `N` is reached again and all threads are unblocked. Therefore, the driver only schedules revisits for barrier reads if the new value is `N`.

3.1.2 Issue with the Old Barrier Model

Let us now consider the barrier well-formed program DoubleBarrier. This is the most compact example which exhibits the problem we have been teasing at. Note that even though this is a constructed example, the issue we are about to present also occurs in real-world code. The key component which makes this fail is that two (or more) barrier rounds on the same barrier occur in sequence.

```
          barrier_init(b, 2);
barrier_wait(b); || barrier_wait(b);      (DoubleBarrier)
barrier_wait(b); || barrier_wait(b);
```

We can see in Fig. 3.2, in graph (1), the only possible execution using barrier labels and **sbr**, abstracting away **co** and **rf**. This would correspond to a normal run of GenMC with BAM. In graph (2) we can see the problematic execution without using barrier labels and again adding the previously mentioned relations. This is one possible execution which GenMC would explore when disabling BAM optimizations, revealing the underlying instructions of the barrier model. As we can see, this execution leads to both threads being blocked indefinitely. This would be acceptable if the real-world execution could also exhibit this behavior, which, however, is not the case. This execution starts with a rendezvous of the threads after which they should continue their execution, since all participating threads have arrived at the barrier and the barrier value has again been set to `N`.

We can see the atomic ternary operation represented as a RMW and the read operation used for the assume, represented as a R. When the first thread passes its assume, it can continue its execution as it reads the value `N` from the location `b`. The same thread then decrements `b` and creates a situation where both threads get stuck, as both threads read 1 and block their execution. What we would like to happen is for thread 2 to realize that the previous barrier round is already over and that it can continue to the next barrier. This is exactly what our new modeling will enable.

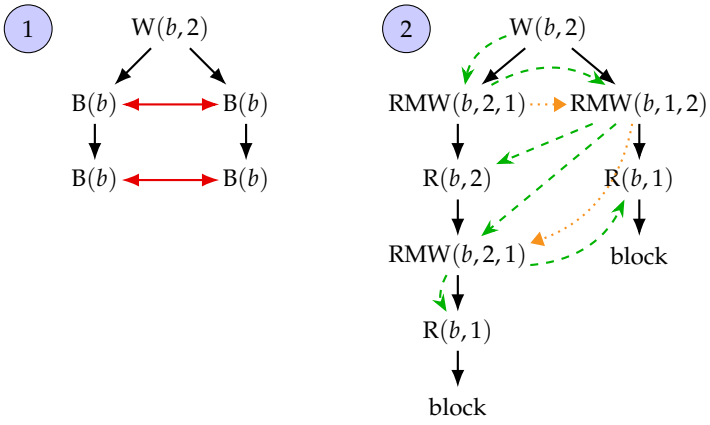

Figure 3.2: The issue with the barrier modeling in BAM

In addition to this fundamental problem in the modeling, we will also address the following list of issues with the implementation in the next section (§ 3.2).

- Illegal use of `barrier_destroy()` is allowed: It should not be allowed to call `barrier_destroy()` on a destroyed or uninitialized barrier.
- Barrier well-formedness is not always enforced: It is possible to construct barriers with more than `N` participants. The program consisting of `N`+1 threads only performing a rendezvous at a barrier, for example, is not recognized as being ill-formed.
- The return value of `barrier_wait()` can be used: As we have seen in § 2.3.3, it should not be allowed to use the return value of wait calls, as this would make it possible to distinguish the threads; where the whole point of BR is to make the wait calls in the same barrier round indistinguishable from each other.

3.2 Improved Model

Central to our implementation improvements is the new barrier modeling, which we will see first in this section. The new model will then also be implemented in a different way than its predecessor, which we will describe next. Lastly, we will see a selection of other changes that were made to improve performance, functionality, and integration of the new modeling.

3.2.1 New Modeling

In addition to the two functions presented in the old model, we also add the destruction method and the `barrier_t` barrier type which stores the current barrier value `c` and the initialization value `i`. See in Fig. 3.3 the old model for reference.

```
barrier_init(b, N) :
   b = N
barrier_wait(b) :
   atomic { if (b == 1) then b = N else b = b-1 }
   assume(b == N)
```

Figure 3.3: Old modeling of `barrier_init()` and `barrier_wait()`

Our newly proposed model in Fig. 3.4 shows the four parts that we implemented in GenMC. The barriers now always set their value `c` to 0 in the beginning and store the number of participating threads `N` in the second variable `i` of the barrier. When the destroy function is called, the barrier is marked as destroyed by setting the current value `c` to -1. Every time wait is called on a barrier, the thread atomically fetch-and-increments (FAI) the current value and then later fetches the current value again. Both the value read before the FAI and the newly read value are divided by the init

value `i` through integer division which always rounds the number down. As results we get the active barrier round before the wait call and the active barrier round after the wait call. We then use `assume()` to ensure that we only continue execution if the current barrier round has been fully completed, i.e. the barrier round number after the call is higher than before.

```
barrier_init(b, N) :
   b.c = 0
   b.i = N

barrier_wait(b) :
   assume(FAI(b.c)/b.i < b.c/b.i)
```

```
barrier_destroy(b) :
   b.c = -1

barrier_t :
   int c
   int i
```

Figure 3.4: New modeling of `barrier_init()`, `barrier_wait()`, `barrier_destroy()` and `barrier_t`

To see the new model in action and compare it with the old model, we again consider the faulty execution of the old model we have seen before. In Fig. 3.5, on the left in ① the original execution is shown using the original model. In ② we can see the new model, where we removed all accesses to the initialization value `i`, as it is always set to `N` throughout the lifespan of the barrier. The key difference we can observe is that in ②, the second thread's read can read from the second barrier round and still advance its execution. This is because our new modeling incorporates a distinction between different barrier rounds which the second thread can use to tell that the first barrier round is already over. The obvious issue of the barrier counting only upwards and leading to an overflow, which the model cannot handle, should not be a problem. As we have seen extensively in § 2, it is very hard to efficiently verify a program, and as a result, we would have to run the verification for a very long time before this could lead to an overflow.

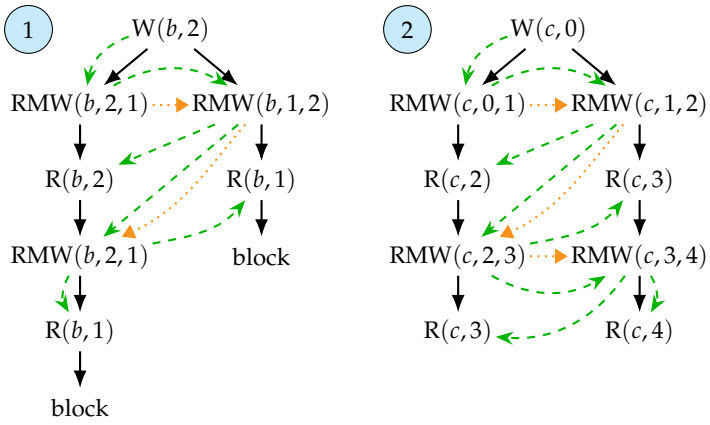

Figure 3.5: Comparing old and new barrier modelings for a double barrier on two threads

3.2.2 Implementation Changes

As we have mentioned before in § 3.1, the new implementation is done differently from the old one. All old code, handling barrier functions in the interpreter, has been deleted in exchange for embedding the model into GenMC's custom `pthread.h` file. Doing this simplifies the implementation, but users can still utilize the functions as before. This subsection comprises a selection of the changes that were made to GenMC to incorporate the new model. We will start with the core implementation in `pthread.h` and then follow the order described below. In addition to these changes, we also added a new `GENMC_KIND_BARRIER` label for annotating barrier functions. This will be very useful at various points of our implementation in distinguishing barrier functions from others. These new barrier-annotated operations have also received dedicated handling in the driver. New error types to denote encounters with barriers that are not well formed and another error type for illegal uses of `pthread_barrier_destroy()` have been added.

In Lst. 1 we can see the newly defined barrier type, and in Lst. 2, 3, 4, the implementation of barrier functions in `pthread.h` which try to follow the semantics of pthread barriers [1]. The function `pthread_barrier_wait()` uses the newly created internal assume, for which we can see code in Lst. 5, 6. The BarrierReturnUnusedPass.cpp LLVM pass has three functions, which are shown in Lst. 7, 8 and 9. Lst. 10 shows how illegal uses of barrier functions are prevented. The function in Lst. 11 checks for barrier well-formedness. The adapted barrier revisit optimization can be seen in Lst. 12, it also uses a helper function depicted in Lst. 13 and is called from Lst. 14.

```
1  typedef struct {
2          int64_t __private;
3          int __count;
4  } __VERIFIER_barrier_t;
```

Listing 1: __VERIFIER_barrier_t in genmc_internal.h explained in Desc. 1

Lst. 1: The definition of `barrier_t` in `pthread.h` refers to this definition, which is used in the barrier functions. Note that `__private` needs to be explicitly defined as `int64_t`, as it is at some point used together with unsigned values, for which we need to know what value -1 has, to recognize destroyed barriers.

```
/* Initialize BARRIER with the attributes in ATTR.  The barrier
   is opened when COUNT waiters arrived.  */
__attribute__((always_inline)) static inline int
pthread_barrier_init(pthread_barrier_t *__restrict __barrier,
                     const pthread_barrierattr_t *__restrict __attr,
                     unsigned int __count)
{
        static bool warned = false;
        if ((__attr != NULL) && !warned) { // only warn first time
                warned = true;
#ifdef __cplusplus
                std::printf("WARNING: pthread-barrier-init "
                            "attribute, "
                            "Ignoring non-null argument given "
                            "to pthread_barrier_init.\n");
#else
                printf("WARNING: pthread-barrier-init "
                       "attribute, "
                       "Ignoring non-null argument given to "
                       "pthread_barrier_init.\n");
#endif
        }

        __barrier->__count = __count;

        __VERIFIER_annotate_begin(GENMC_KIND_BARRIER);
        __barrier->__private = 0;
        __VERIFIER_annotate_end(GENMC_KIND_BARRIER);

        return 0;
}
```

Listing 2: pthread_barrier_init() in pthread.h explained in Desc. 2

Lst. 2: The actual model is implemented in the lower half of the code. On the top, we make sure that users are warned about giving a barrier attribute to the function, as we do not handle them. This should not pose an issue as this feature is rarely used. The warning is printed for C++ using `cstdio` and with `cstdio.h` for C programs. The `__count` variable represents our initialization value i in the model and is set analogously. We also set the c variable, represented by `__private`, in the same way as the model, but additionally use annotation functions to mark the instructions as part of a barrier function. One of the transformations that are run on the LLVM code goes over all of these annotation functions and replaces them with annotations in the LLVM IR. In this case, the instruction will be recognized through this as a barrier initialization write and will be represented as a `BInitWrite` in

the execution graph, where `BInitWrite` is a subclass of `Write` which in turn is a subclass of `Event`.

```
1  /* Wait on barrier BARRIER.  */
2  __attribute__((always_inline)) static inline int
3  pthread_barrier_wait(pthread_barrier_t *__barrier)
4  {
5          int iVal = __barrier->__count; // __count is never modified
6                                         // after initialization
7
8          __VERIFIER_annotate_begin(GENMC_KIND_BARRIER);
9          int b_old = __atomic_fetch_add(&__barrier->__private, 1,
10                                        __ATOMIC_ACQ_REL);
11         __VERIFIER_annotate_end(GENMC_KIND_BARRIER);
12
13         __VERIFIER_annotate_begin(GENMC_KIND_BARRIER);
14         int b_new = __atomic_load_n(&__barrier->__private,
15                                     __ATOMIC_ACQUIRE);
16         __VERIFIER_annotate_end(GENMC_KIND_BARRIER);
17
18         __VERIFIER_assume_internal(
19                 b_old / iVal < b_new / iVal,
20                 GENMC_ASSUME_BARRIER); // mark as barrier assume
21
22         int res = (b_old % iVal != 0) ? 0
23                                       : PTHREAD_BARRIER_SERIAL_THREAD;
24
25         return res;
26 }
```

Listing 3: pthread_barrier_wait() in pthread.h explained in Desc. 3

Lst. 3: The `pthread_barrier_wait()` also closely follows the model, where we have split the different parts into separate lines in order to annotate them. We first get the number of participating threads. Then we annotate the FAI operation, which later will be recorded split into a `BIncFaiRead` and a `BIncFaiWrite`. The second read is also annotated and will later be recorded as a `BWaitRead`. We will see how the `__VERIFIER_assume_internal()` works in Lst. 5, but for now consider this to be a normal `__VERIFIER_assume()` where we happen to mention that it has been used in a barrier. The return value `res` follows the pthread semantics of returning `PTHREAD_BARRIER_SERIAL_THREAD` to one single thread when the barrier has completed a full round.

```c
/* Destroy a previously dynamically initialized barrier BARRIER.
 */
__attribute__((always_inline)) static inline int
pthread_barrier_destroy(pthread_barrier_t *__barrier)
{
        __VERIFIER_annotate_begin(GENMC_KIND_BARRIER);
        __barrier->__private = -1; // mark as destroyed
        __VERIFIER_annotate_end(GENMC_KIND_BARRIER);

        return 0;
}
```

Listing 4: pthread_barrier_destroy() in pthread.h explained in Desc. 4

Lst. 4: Destroying a barrier is very simple. Like in the model, we set the current value to -1 and are done. In the implementation we also annotate the instruction and later record it as a `BDestroyWrite`.

```c
typedef enum {
        GENMC_ASSUME_USER,
        GENMC_ASSUME_BARRIER
} assume_kind_t;

/*
 * Blocks the current execution if the argument is false, allows
 * for choice of barrier type Should match interpreter's assume
 * call
 */
extern void __VERIFIER_assume_internal(bool cond, assume_kind_t type)
        __attribute__((__nothrow__));
```

Listing 5: __VERIFIER_assume_internal() and assume_kind_t in genmc_internal.h explained in Desc. 5

Lst. 5: We can see the definition of the `__VERIFIER_assume_internal()` function, which can be used by including GenMC's internal functions within `genmc_internal.h`. Currently, there are only two types of assume which are handled in the interpreter (Lst. 6). The normal `__VERIFIER_assume()` can also still be accessed within `genmc.h` and internally leads to a call to the internal function using `GENMC_ASSUME_USER`.

```
void Interpreter::callAssume(
        Function *F, const std::vector<GenericValue> &ArgVals,
        const std::unique_ptr<EventDeps> &specialDeps)
{
        if (ArgVals[0].IntVal.getBoolValue()) {
                return;
        } else {
                switch (ArgVals[1].IntVal.getSExtValue()) {
                case GENMC_ASSUME_USER:
                        CALL_DRIVER(
                                handleBlock,
                                UserBlockLabel::create(currPos()));
                        break;
                case GENMC_ASSUME_BARRIER:
                        CALL_DRIVER(
                                handleBlock,
                                BarrierBlockLabel::create(currPos()));
                        break;
                default:
                        BUG();
                }
        }
}
```

Listing 6: Interpreter::callAssume() in Execution.cpp explained in Desc. 6

Lst. 6: The assume first checks if the condition supplied as an argument holds, and if it does not, tell the driver that the thread is going to block. Here we can see traces of the reason why we needed to be able to distinguish the different types of assume. Depending on the type of assume we get, we will create different blockage types. For our barrier function, we would create `BarrierBlock` events in the execution graph. This is especially useful when dealing with barrier revisits, as we will see in Lst. 12. Note that annotating the assume as we did for the other barrier instructions would not work here. This is because the annotation pass can currently only deal with operations composed of a single instruction in between the start and end of the annotation. In our case, we are also performing some additional operations, such as divisions, which we would have to relocate somewhere else. This would make it cumbersome to use the assume function.

```
auto BarrierReturnUnusedPass::run(Module &M,
                                  ModuleAnalysisManager &MAM)
        -> PreservedAnalyses
{
        for (Function &f : M) {
                for (Instruction &i : instructions(f)) {
                        /* look for assume_internal (used in
                         * pthread_barrier_wait()) */
                        auto *inst = dyn_cast<CallInst>(&i);
                        if (!(inst))
                                continue;
                        const Function *calledFunc =
                                inst->getCalledFunction();
                        if (!calledFunc)
                                continue;

                        if (calledFunc->getName() ==
                            "__VERIFIER_assume_internal")
                                checkFunctionUse(inst);
                }
        }
        return PreservedAnalyses::all();
}
```

Listing 7: BarrierReturnUnusedPass::run() in BarrierReturnUnusedPass.cpp explained in Desc. 7

Lst. 7: As we explain in § 2.3.3, it is not allowed to use the return value of `pthread_barrier_wait()` when using BR. To prevent usage, we implemented an LLVM pass, which looks for uses of the return value and decides if they are legal. There is one single case which we still want to allow because it is used frequently in code using barriers. Users might want to make sure that the wait function returned a legal value, i.e. executed correctly, and check if the returned value is one of the values `0` or `PTHREAD_BARRIER_SERIAL_THREAD = -1`. To facilitate this, we allow the return value to be compared to these two values and do not want to allow any other use. Notice that this does not allow the user to distinguish between threads, as we still do not allow comparison with only one of the two, as we will see in more detail in Lst. 9. Thus, enforcing this still lets us use BR. Note that this is a heuristic and will not always be able to catch illegal usage.

```
1  static void checkFunctionUse(const CallInst *inst)
2  {
3          /* only pthread_barrier_wait() should use GENMC_ASSUME_BARRIER
4           * in it's assume call */
5          const auto *op1 =
6                  dyn_cast<ConstantInt>(inst->getArgOperand(1));
7          const Instruction *critInst = inst->getNextNode();
8          if (!(op1 && critInst &&
9                op1->getZExtValue() == GENMC_ASSUME_BARRIER))
10                 return;
11
12 /* Look for the select instruction which determines the return value
13  */
14 #if LLVM_VERSION_MAJOR < 16
15         ...
16 #else
17         while (critInst && (isa<DbgInfoIntrinsic>(critInst) ||
18                             !isa<SelectInst>(critInst)))
19 #endif
20                 critInst = critInst->getNextNode();
21
22         if (!critInst)
23                 return;
24
25         for (auto *u : critInst->users()) {
26                 auto *instUser = dyn_cast<Instruction>(u);
27                 if (!instUser)
28                         continue;
29 #if LLVM_VERSION_MAJOR < 16
30                 ...
31 #else
32                 if (isa<DbgInfoIntrinsic>(instUser))
33 #endif
34                         continue;
35                 if (!mightBeAllowedBarrierReturnUse(u))
36                         ERROR("The return value of "
37                               "pthread_barrier_wait() must not be "
38                               "used! "
39                               "You can disable this restriction with "
40                               "-disable-bam\n");
41         }
42 }
```

Listing 8: checkFunctionUse() in BarrierReturnUnusedPass.cpp explained in Desc. 8

Lst. 8: The `run()` function runs the pass by going through all instructions to look for uses of the internal assume, which we know is used in `pthread_barrier_wait()`. We can then check if it is a barrier assume, to confirm that we are in a call to the wait function, as only the barrier wait function uses a barrier assume. The computation of the

return value is what really interests us, as we will check how it is used later on. What we can do now, is use our knowledge of how the `pthread_barrier_wait()` implementation looks (Lst. 3) and of how this compiles into LLVM IR, to find the instruction which determines the final return value. It so happens that the instruction we are looking for normally compiles to a select instruction which we can easily find. Every time we find such a select, we then call mightBeAllowedBarrierReturnUse() (Lst. 9) and return an error if our function determines that it is not possible for it to be a legal use case.

```
1  static bool mightBeAllowedBarrierReturnUse(const Value *V)
2  {
3          auto *IC = dyn_cast<ICmpInst>(V);
4          auto *CIFirst = dyn_cast<ConstantInt>(IC->getOperand(1));
5          int64_t CI1 = CIFirst->getSExtValue();
6
7          if (!(IC && CIFirst))
8                  return false;
9
10         if (!(IC->getPredicate() == ICmpInst::ICMP_NE))
11                 return false;
12
13         if (!(CI1 == 0 || CI1 == PTHREAD_BARRIER_SERIAL_THREAD))
14                 return false;
15
16         /* Check if conditional branch options lead to the same basic
17          * block
18          * --> Will likely be a legal use */
19         if (IC->getNextNode() && isa<BranchInst>(IC->getNextNode())) {
20                 const BranchInst *BI =
21                         dyn_cast<BranchInst>(IC->getNextNode());
22
23                 if (BI->isConditional()) {
24                         const BasicBlock *BB0 = BI->getSuccessor(0);
25                         const BasicBlock *BB1 = BI->getSuccessor(1);
26
27                         if (BB0 == BB1)
28                                 return true;
29                 }
30         }
31         /* Default: might be an allowed use case */
32         WARN("The return value of pthread_barrier_wait() must not be "
33              "used! "
34              "You can disable this restriction with -disable-bam\n")
35         return true;
36 }
```

Listing 9: mightBeAllowedBarrierReturnUse() in BarrierReturnUnusedPass.cpp explained in Desc. 9

Lst. 9: `mightBeAllowedBarrierReturnUse()` makes sure that we indeed are trying to compare the return value to 0 or -1 and then performs a heuristic check to decide whether the usage should be allowed. If after this check, we are still not sure if the usage is legal (i.e., we know that it is being compared to 0 or -1, but not for sure if it is being compared to both), we print a warning to the user and return that the usage might be legal.

Lst. 10: As mentioned in § 3.1.2, checks for the legal usage of barrier functions are incomplete. Implemented already are checks for legal initialization here and legal use cases for waits in a different function. This is why we additionally add a check for destroying uninitialized barriers and for destroying already destroyed barriers. To be more precise, every time a read is performed, we check to see if it is a `BDestroyWrite` and in that case check for legal use. How these checks work is that the most recent BInitWrite or BDestroyWrite is retrieved and then used to determine whether the barrier is destroyed or not. We can firstly report an error if the most recent such event has been a destroy and we want to destroy it again. Secondly, we report an error if we find neither a destroy nor an initialization event and try to destroy.

Lst. 11: Checking for barrier well-formedness is a bit more elaborate. It is performed every time we encounter a barrier wait call, which we recognize by the `BIncFaiWrite`. The previous implementation only performs this check every time a barrier round is completed, which is problematic. We err on the side of caution and check every `barrier_wait()` call. We start by retrieving the number of participating threads in the barrier, which is conveniently read every time we wait on a barrier, two events before performing the write part of the FAI. We then look for the most recent initialization event for the barrier, to ensure that it has been initialized. (If it had been destroyed since then, the check in Lst. 10 would have caused an error.) Third, we go over all the wait calls that have been made since the initialization of this barrier. From all these calls, we can then count the number of participating threads to confirm that at most `N` threads are participating in the barrier. Finally, we print a warning about the barrier not being well-formed if not using BR, and returning an error if we are using BR.

Lst. 12: The idea of this function is to try and optimize when to revisit barrier functions and then also do so. We first check if we need to do a revisit, determined by the `shouldRevisitBarrierReads()` function shown in Lst. 13. We then collect all read events for which we need to schedule a revisit. We are looking for threads' `BWaitRead` and if the thread has already blocked, it should be by a `BarrierBlock`. All of these events are then revisited in-place.

```
1  VerificationError
2  GenMCDriver::checkInitializedMem(const WriteLabel *wLab)
3  {
4          auto &g = getExec().getGraph();
5          /* Unlocks should unlock mutexes locked by the same
6           * thread */
7          ...
8          /* Barriers should be initialized once, with a proper
9           * value */
10         const auto *bInitLab = LLVM::dyn_cast<BInitWriteLabel>(wLab);
11         if (bInitLab) {
12                 /* Check init with legal value */
13                 ...
14                 /* Check initialized twice */
15                 ...
16         }
17         /* Barriers should not be destroyed before they are
18          * initialized or multiple times in a row*/
19         const auto *bDLab = LLVM::dyn_cast<BDestroyWriteLabel>(wLab);
20         if (bDLab) {
21                 auto revCo = g.rco(bDLab->getAddr());
22                 auto recent = std::find_if(
23                         revCo.begin(), revCo.end(),
24                         [bDLab](const auto &lab) {
25                                 return (LLVM::isa<BInitWriteLabel>(
26                                                 lab) ||
27                                         LLVM::isa<BDestroyWriteLabel>(
28                                                 lab)) &&
29                                        &lab != bDLab;
30                         });
31                 /* Check uninitialized */
32                 if (recent == revCo.end()) {
33                         reportError({wLab->getPos(),
34                                      VerificationError::
35                                              VE_InvalidBDestroy,
36                                      "Called barrier_destroy() on "
37                                      "uninitialized barrier!"});
38                         return VerificationError::VE_InvalidBDestroy;
39                 }
40                 /* Check destroy twice */
41                 const auto *rsLab =
42                         LLVM::dyn_cast<EventLabel>(&*recent);
43                 if (!rsLab)
44                         BUG();
45                 if (LLVM::isa<BDestroyWriteLabel>(rsLab)) {
46                         reportError({wLab->getPos(),
47                                      VerificationError::
48                                              VE_InvalidBDestroy,
49                                      "Called barrier_destroy() "
50                                      "multiple times!"});
51                         return VerificationError::VE_InvalidBDestroy;
52                 }
53         }
54         return VerificationError::VE_OK;
55 }
```

Listing 10: GenMCDriver::checkInitalizedMem() for writes in GenMCDriver.cpp explained in Desc. 10

```
1  void GenMCDriver::checkBarrierWellFormed(BIncFaiWriteLabel *sLab)
2  {
3          auto &g = getExec().getGraph();
4          const auto *cLab = LLVM::dyn_cast<ReadLabel>((g.po_imm_pred(
5                  g.po_imm_pred(sLab)))); // init value read
6          const auto *cInitLab = LLVM::dyn_cast<WriteLabel>(
7                  cLab->getRf()); // init value write
8
9          if (!cInitLab || !cLab)
10                 BUG();
11         /* get the barriers initilization value */
12         const SVal iVal = cInitLab->getVal();
13         int iValS = iVal.getSigned();
14
15         /* find the most recent BInitWriteLabel */
16         auto revCo = g.rco(sLab->getAddr());
17         auto recent = std::find_if(
18                 revCo.begin(), revCo.end(), [](const auto &lab) {
19                         return LLVM::isa<BInitWriteLabel>(lab);
20                 });
21         if (recent == revCo.end())
22                 BUG();
23         /* count threads calling same barrier round concurrently */
24         const auto *rsLab = LLVM::dyn_cast<EventLabel>(&*recent);
25         auto coSuccs = co_succs(g, rsLab);
26         auto tView =
27                 coSuccs | std::views::filter([&g, sLab](auto &lab) {
28                         auto lLab = LLVM::dyn_cast<BIncFaiWriteLabel>(
29                                 &lab);
30                         if (!lLab)
31                                 return false;
32                         return (lLab->getAddr() == sLab->getAddr());
33                 }) |
34                 std::views::transform(
35                         [](auto &lab) { return lab.getThread(); });
36         std::set<int> ts(std::ranges::begin(tView),
37                          std::ranges::end(tView));
38         ts.insert(sLab->getThread());
39         int threads = ts.size();
40
41         /* BAM */
42         if (threads > iValS) {
43                 if (getConf()->disableBAM)
44                         reportWarningOnce(
45                                 sLab->getPos(),
46                                 VerificationError::
47                                         VE_NotBarrierWellFormed);
48                 else
49                         reportError({sLab->getPos(),
50                                      VerificationError::
51                                              VE_NotBarrierWellFormed,
52                                      "bam-well-formed, Execution "
53                                      "not barrier well-formed!\n"});
54         }
55         return;
56 }
```

Listing 11: GenMCDriver::checkBarrierWellFormed() in GenMCDriver.cpp explained in Desc. 11

```
bool GenMCDriver::tryOptimizeBarrierRevisits(
        BIncFaiWriteLabel *sLab, std::vector<ReadLabel *> &loads)
{
        if (getConf()->disableBAM)
                return false;

        /* If the barrier_wait() does not write a multiple of the
         * initial value, nothing to do */
        auto &g = getExec().getGraph();
        if (!shouldRevisitBarrierReads(sLab))
                return true;

        /* Otherwise, revisit in place */
        auto bsView =
                g.labels() |
                std::views::filter([&g, sLab](auto &lab) {
                        if (!LLVM::isa<BWaitReadLabel>(&lab))
                                return false;
                        auto *pLab =
                                LLVM::dyn_cast<BIncFaiWriteLabel>(
                                        g.po_imm_pred(&lab));
                        auto *bLab = g.po_imm_succ(&lab);
                        if (bLab &&
                            !LLVM::isa<BarrierBlockLabel>(bLab))
                                return false;
                        return pLab->getAddr() == sLab->getAddr();
                }) |
                std::views::transform([](EventLabel &lab) {
                        return LLVM::dyn_cast<BWaitReadLabel>(&lab);
                });
        std::vector<ReadLabel *> bs(std::ranges::begin(bsView),
                                    std::ranges::end(bsView));

        for (auto *lab : bs) {
                revisitInPlace(*constructBackwardRevisit(lab, sLab));
        }
        return true;
}
```

Listing 12: GenMCDriver::tryOptimizeBarrierRevisits() in GenMCDriver.cpp explained in Desc. 12

Lst. 13: The determining factor for if we should schedule revisits, is if all participating threads have arrived at the barrier. We can perform this optimization because we know that if fewer than `N` threads have arrived at the barrier, they cannot advance anyway and there is nothing to gain in scheduling a revisit.

```
bool GenMCDriver::shouldRevisitBarrierReads(const WriteLabel *wLab)
{
        auto &g = getExec().getGraph();
        const auto *iLab = LLVM::dyn_cast<ReadLabel>(
                (g.po_imm_pred(g.po_imm_pred(wLab))));
        const auto *InitLab =
                LLVM::dyn_cast<WriteLabel>(iLab->getRf());

        const SVal wVal = wLab->getVal();
        const SVal iVal = InitLab->getVal();

        return (wVal % iVal) == SVal(0);
}
```

Listing 13: GenMCDriver::shouldRevisitBarrierReads() in GenMCDriver.cpp in Desc. 13

```
bool GenMCDriver::tryOptimizeRevisits(WriteLabel *sLab,
                                      std::vector<ReadLabel *> &loads)
{
        auto &g = getExec().getGraph();

        /* BAM */
        if (auto *faiLab = LLVM::dyn_cast<BIncFaiWriteLabel>(sLab)) {
                checkBarrierWellFormed(faiLab);
                if (!getConf()->disableBAM) {
                        if (tryOptimizeBarrierRevisits(faiLab, loads))
                                return true;
                }
        }

        /* IPR + locks */
        ...

        /* Confirmation: Do not bother with revisits that will
         * lead to unconfirmed reads */
        ...
        return false;
}
```

Listing 14: GenMCDriver::tryOptimizeRevisits() in GenMCDriver.cpp explained in Desc. 14

Lst. 14: This is a general purpose function for optimizing revisits. It is called every time we encounter a new `WriteLabel`. If the write is a `BIncFaiWrite`, we know that we are at a barrier and check the well-formedness of the barrier. If we also have BAM enabled, we try to optimize barrier revisits as well.

Chapter 4

Ideas on Combining BR and SR

As previously mentioned, the combination of SR and DPOR has already been investigated and implemented as SPORE in GenMC. BR has been constructed on top of DPOR from the beginning; as such, the combination of BR and DPOR is already implemented as BAM in GenMC. We can therefore say that GenMC uses a barrier-aware combination of SR and DPOR in which SR does not interact with BR. Our goal is to combine these two types, where the approach of imposing SR onto BR is not very promising, as barriers are already symmetric by nature. That is why we will investigate the opposite possibility. We use BR within SR to improve its performance and, through this, combine all three types of reduction. We will now describe why BR and SR do not naturally combine, how we can do so none the less, and describe loose ideas on how to implement the combination. Afterwards, this will lead us to use our newly found ideas to use BR to, in a sense, reactivate SR. However, note that none of these ideas have actually yet been implemented in GenMC.

Our first hurdle is that BR interrupts the functionality of SR. The key reason is that every wait on a barrier includes a read of the global barrier variable.[1] As we have seen in § 2.3.2, symmetric threads reading different values can lead to their local state diverging, thus breaking symmetry. This should not happen at barrier reads, as the barrier variable accessed is usually not stored and the thread symmetry should still hold after a `barrier_wait()`. We want threads to be considered symmetric if they are indistinguishable from one another when disregarding the thread ID and ordering imposed by scheduling. Since the whole point of BR has been to make barrier wait events indistinguishable, the barrier should not affect symmetry.

[1] Both the old modeling and our new version include reads to a global variable, and it is hard to imagine a model that could avoid doing so, as the threads need some central authority to determine the barrier state.

To illustrate this, consider the IncBarrier example program below. The threads first perform a fetch-and-increment operation, where we consider them to be symmetric because they do not make the threads distinguishable. The threads might perform different operations, but this is only imposed by the scheduling of the threads. We can still permute the thread IDs to get the other executions. The threads then rendezvous at the barrier, and afterwards, the threads again perform a symmetric operation. We would like these threads to be considered symmetric, as the symmetry after the barrier does not seem to be affected by the wait. But since the threads call `barrier_wait()`, their symmetry breaks at that point of the execution. To facilitate this program being recognized as symmetric, or rather avoiding the breaking of symmetry, we need to set a condition, which determines if a potentially symmetry-breaking operation is indeed symmetry-breaking. This condition should allow these types of barrier waits to not break the symmetry. We now therefore discuss how to set this condition.

```
            barrier_init(b, N);
fai(x);           ||   ||  fai(x);
barrier_wait(b);  ||  ⋮ ||  barrier_wait(b);
fai(x);           ||   ||  fai(x);
```
(IncBarrier)

Given that the issue we observe is that the local states diverge, we could define the following condition. If the local states diverge, the symmetry does not break if the threads do not access the local state for the rest of the execution (i.e., do not use any local variables). This would keep the threads indistinguishable from one another as there no longer is a local variable that could be used to determine the control flow of the execution. The method would work for the example IncBarrier above. Local states diverge at the wait, but since we do not use any local variables, the threads are still symmetric. However, the condition is too restrictive to be used in practice, as programs often want to reuse their local variables. One such program LocalBarrier can be seen below. The threads access the local variable `l` after the potential symmetry break in `barrier_wait()`. This means that they can no longer be considered symmetric by this condition, even though the threads cannot be distinguished from each other. It is hard to find a real-world program that would work with this condition, which is why we go over to a second idea.

```
            barrier_init(b, N);
int l = 0;        ||   ||  int l = 0;
fai(x);           ||   ||  fai(x);
barrier_wait(b);  ||  ⋮ ||  barrier_wait(b);
fai(l);           ||   ||  fai(l);
```
(LocalBarrier)

Since we cannot disallow all usage of local variables, we do the next best thing. We require that all variables, which are used later in the execution, hold the same value in all symmetric threads (i.e., all symmetric threads have the same local state). This would again work for our first example IncBarrier and would now also work for our LocalBarrier program. At the barrier we would check if all local variables (here only `l`) have the same value in all symmetric threads. This is the case here, as all threads have symmetrically set the value to 0. We would then be able to keep the threads symmetric for the rest of the execution. This gives us a decent condition to increase symmetry between threads, but still does not guarantee that we always recognize that symmetry has not really been broken. To see this we will consider the TrickBarrier program.

```
            barrier_init(b, N);
int l = fai(x);  ||     ||  int l = fai(x);
barrier_wait(b); ||  ⋮  ||  barrier_wait(b);     (TrickBarrier)
l = 0;           ||     ||  l = 0;
fai(l);          ||     ||  fai(l);
```

TrickBarrier is quite similar to LocalBarrier. But this program here assigns different values to the local variable `l` in different threads. When symmetry breaks in the barrier's wait call, we would by our new condition check if all locals have the same value, which clearly is not the case here. But right after the barrier, the local values synchronize again by setting them to 0. This then again results in the exact same state as we had in LocalBarrier right after the barrier. But in the previous example, we still considered the threads to be symmetric, while here we no longer do so. The condition is therefore clearly flawed. We will for now be content with our second condition.

Our new condition of all live variables having to hold identical values across symmetric threads does not impose any restrictions on the program before the barrier. From this we can construct an even more effective technique of embedding BR into SR. We know that every time threads rendezvous at a barrier, they will restart their execution from that point. This gives us a similar starting point to the initial running of the threads. If the threads have the same state after the barrier, they can again be considered symmetric. For this to be true, we can again employ our newly found condition. Thus, every time we get to a barrier, running on threads with the same code, we can compare all live variables and, if all hold the same values, restore symmetry between all participating threads that run the same code.

We will now investigate how our ideas could be implemented in GenMC. First, to find all live variables, we could implement an LLVM pass which collects all live variables at every `barrier_wait()` call. As part of the pass,

the wait instructions could be annotated with the set of all live variables or recorded in some other way. When running the verification process, every time we encounter a `barrier_wait()` on threads running the same code, we could check all live variables across all participating threads and declare the threads symmetric which hold the same values for all live variables.

Chapter 5

Evaluation

As we have implemented the new barrier model in GenMC, we evaluate it compared to the previous BAM implementation. This will involve two test arrangements. In the first, we compare the old implementation with our new implementation, while disabling barrier reductions. This will allow us to compare the barrier models with each other while reducing the impact of other implementation changes on the comparison. Secondly, we will compare the complete implementation with its predecessor without any deactivation. We call the old implementation 'BAM' and the new implementation 'BAM_v2'.

The goal of our implementation changes has not been to improve the performance of GenMC but mainly to tackle other issues with the implementation. We do not expect our implementation to affect programs that do not use barriers. To confirm this, we ran both versions on GenMC's own fast-driver test suite, where only a small fraction of tests include barriers. The results were very similar (8 min 39 s and 8 min 36 s), which supports our claim that we only significantly affect programs with barriers. From our new model, we expect to get different numbers of explorations, as we no longer explore the faulty blocked executions we saw in § 3.1.2. We predict that less blocked executions should be explored, as we should now be able to always successfully revisit the barriers' read events. Also, we expect more unblocked executions, as our model now includes more conflicting instructions than the old one. For comparison of the complete implementation, we expect similar, if not slightly worse, performance for our new version because our new features might slow down the execution.

We ran all tests on an Apple MacBook Pro (2021) running macOS Sequoia 15.5, Apple M1 Pro, 16GB of RAM. We use LLVM 18.1.8 and set a timeout limit of 5 minutes (denoted by 🕒). All listed times are in seconds.

The first test setup compares GenMC's implementation without our changes, using the old faulty barrier model and our new version, which adds the

Table 5.1: Synthetic benchmarks with -disable-bam

	Executions		Blocked		Time	
	BAM	**BAM_v2**	**BAM**	**BAM_v2**	**BAM**	**BAM_v2**
2thread-1barrier	2	2	2	0	0.03	0.03
3thread-1barrier	6	6	30	0	0.03	0.03
4thread-1barrier	24	24	552	0	0.03	0.03
5thread-1barrier	120	120	14 280	0	0.13	0.04
2thread-2barrier	4	8	10	0	0.03	0.03
3thread-2barrier	36	216	462	0	0.03	0.04
4thread-2barrier	576	13 824	36 816	0	0.25	0.61
5thread-2barrier	14 400	1 728 000	5 156 880	0	25.16	90.68
2thread-3barrier	8	32	26	0	0.03	0.03
3thread-3barrier	216	7776	3054	0	0.06	0.42
4thread-3barrier	13 824	🕒	907 152	🕒	6.26	🕒
5thread-3barrier	🕒	🕒	🕒	🕒	🕒	🕒

Xthread-Ybarrier: X threads rendezvous at a barrier Y times in a row.

new model in the new implementation. We use the `-disable-bam` flag that prevents barrier reductions, disables the check to use the return value of `barrier_wait()` and transforms well-formedness checks to only cause warnings instead of errors. All other implementation changes are still used. We run both versions on synthetic test programs that only use barriers.

The results in Tab. 5.1 largely align with our expectations. The old model explores the faulty blocked executions, which, through our new modeling are now ignored. We see an increase in the number of unblocked executions for our new model, which stems from an increase in the number of explored interleavings of the instructions in our barrier model. This increased number of interleavings as well as our well-formedness check (and other checks) could explain the significant slowdown. Especially since the well-formedness check is not involved at all in the old implementation when deactivating BAM.

The second test set-up will first run the complete implementations on the same type of synthetic barrier tests we used before and then on some realistic tests which use barriers.

As expected, the results in Tab. 5.2 show that we continue to get only one explored execution, which is what we expect from programs that only use barriers. There is a significant slowdown in the adapted version, for which we continue to suspect the reasons we mentioned before. An additional reason here for the slowdown is that our new implementation checks barrier well-formedness every time a `BIncFaiWrite` is encountered (which is the

Table 5.2: Synthetic benchmarks with BAM enabled

	Executions		Blocked		Time	
	BAM	**BAM_v2**	**BAM**	**BAM_v2**	**BAM**	**BAM_v2**
10^2thread-1barrier	1	1	0	0	0.04	0.12
10^3thread-1barrier	1	1	0	0	1.07	4.24
10^4thread-1barrier	1	🕒	0	🕒	249.93	🕒
10^2thread-2barrier	1	1	0	0	0.04	0.07
10^3thread-2barrier	1	1	0	0	1.46	16.01
10^4thread-2barrier	🕒	🕒	🕒	🕒	🕒	🕒
10^2thread-5barrier	1	1	0	0	0.06	0.21
10^3thread-5barrier	1	1	0	0	3.74	99.70
10^4thread-5barrier	🕒	🕒	🕒	🕒	🕒	🕒
10^2thread-10barrier	1	1	0	0	0.13	0.70
10^3thread-10barrier	1	🕒	0	🕒	18.24	🕒
10^4thread-10barrier	🕒	🕒	🕒	🕒	🕒	🕒

Xthread-Ybarrier: X threads rendezvous at a barrier Y times in a row.

case every time we call `barrier_wait()`). The old implementation performs this check only when the barrier value is set to `N` which is problematic, as we have seen in § 3.1.2.

The results in Tab. 5.3 continue to follow our expectations. Our new implementation shows performance similar to that of the original one. We get a lower number of blocked executions but might get a higher number of non-blocked explorations. In programs which rely heavily on barriers, our check for barrier well-formedness and the structure of our new barrier model might slow down the execution.

Table 5.3: Realistic benchmarks with BAM enabled

	Executions		Blocked		Time	
	BAM	**BAM v2**	**BAM**	**BAM v2**	**BAM**	**BAM v2**
checkReturn(2)	16	16	0	0	0.05	0.12
checkReturn(3)	1638	4096	1062	0	1.13	7.17
checkReturn(4)	🕒	🕒	🕒	🕒	🕒	🕒
determinant(2)	1	1	0	0	0.07	0.17
determinant(3)	1	1	0	0	0.05	0.06
determinant(4)	1	1	0	0	0.05	0.06
determinant(5)	1	1	0	0	0.05	0.06
test_lock(2)	4	4	0	0	0.03	0.03
test_lock(3)	36	36	0	0	0.04	0.04
test_lock(4)	576	576	0	0	0.59	0.60
test_lock(5)	14 400	14 400	0	0	32.19	32.89
transitive_closure(2)	1	1	0	0	0.11	0.26
transitive_closure(3)	1	1	0	0	0.09	0.17
transitive_closure(4)	1	1	0	0	0.07	0.18
transitive_closure(5)	1	1	0	0	0.07	0.19

checkReturn(N): N threads check that their barriers work correctly and that exactly one thread receives `PTHREAD_BARRIER_SERIAL_THREAD` upon completing a barrier round.

determinant(N): N threads calculate the determinant of a 4×4 matrix.

test_lock(N): N threads rendezvous at a barrier before being locked and incrementing a value, to test the lock implementation.

transitive_closure(N): N threads calculate the transitive closure of a 4×4 matrix.

Chapter 6

Conclusion

We presented improvements to BAM and ideas on how to combine SR and BR in GenMC. The improvements to BAM are threefold. A new barrier-model which better conforms to how barriers work has been implemented in GenMC, allowing for distinction between different barrier rounds and eliminating exploration of wrong blocked executions. Secondly, we implemented checks to ensure that the barrier functions are used correctly. Lastly, restrictions on the use of BAM have improved a lot compared to the previous implementation. However, our check for barrier well-formedness leads to a significant slowdown when running GenMC on programs that heavily rely on barriers. We presented ideas on how to use BR in SR to improve performance on programs using barriers, and thus lay the foundation for future work to extend and implement the ideas in GenMC. Future work may also try to generally improve our new BAM implementation and especially try to mitigate the impact of well-formedness checks on the runtime.

Bibliography

[1] pthread.h linux manual page. https://man7.org/linux/man-pages/man0/pthread.h.0p.html, 2017. (Accessed: 2025-06-06).

[2] Genmc github public repository. https://github.com/mpi-sws/genmc/, 2020. (Accessed: 2025-06-06).

[3] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 373–384, New York, NY, USA, 2014. Association for Computing Machinery.

[4] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for TSO and PSO. *Acta Informatica*, 54:789–818, 2017.

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Sarbojit Das, Bengt Jonsson, and Konstantinos Sagonas. *Parsimonious optimal dynamic partial order reduction*, pages 19–43. 2024.

[6] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Tuan Phong Ngo. Optimal stateless model checking under the release-acquire semantics. *Proc. ACM Program. Lang.*, 2(OOPSLA), 10 2018.

[7] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2):1–74, 2014.

[8] Ben Blum. Landslide: Systematic dynamic race detection in kernel space. Master's thesis, Carnegie Mellon University, 2012.

[9] Ben Blum and Garth Gibson. Stateless model checking with data-race preemption points. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 477–493. Association for Computing Machinery, 10 2016.

[10] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of Programs*, pages 52–71, Berlin, Heidelberg, 1982. Springer Berlin Heidelberg.

[11] Edmund M Clarke, Reinhard Enders, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. *Formal methods in system design*, 9:77–104, 1996.

[12] E. Allen Emerson and Thomas Wahl. Dynamic symmetry reduction. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 382–396, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[13] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *ACM Sigplan Notices*, 40(1):110–121, 2005.

[14] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer, 1996.

[15] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 174–186, New York, NY, USA, 1997. Association for Computing Machinery.

[16] Orna Grumberg and Helmut Veith. *25 years of model checking: history, achievements, perspectives*, volume 5000. Springer Science & Business Media, 2008.

[17] Michalis Kokologiannakis, Rupak Majumdar, and Viktor Vafeiadis. Enhancing genmc's usability and performance. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 66–84, Cham, 2024. Springer Nature Switzerland.

[18] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Truly stateless, optimal dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 6:1–28, 01 2022.

[19] Michalis Kokologiannakis, Iason Marmanis, and Viktor Vafeiadis. Spore: Combining symmetry and partial order reduction. *Proceedings of the ACM on Programming Languages*, 8:1781–1803, 06 2024.

[20] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Effective lock handling in stateless model checking. *Proc. ACM Program. Lang.*, 3(OOPSLA), 10 2019.

[21] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 96–110, 2019.

[22] Michalis Kokologiannakis, Xiaowei Ren, and Viktor Vafeiadis. Dynamic partial order reductions for spinloops. In *2021 Formal Methods in Computer Aided Design (FMCAD)*, pages 163–172. IEEE, 2021.

[23] Michalis Kokologiannakis and Viktor Vafeiadis. Bam: efficient model checking for barriers. In *International Conference on Networked Systems*, pages 223–239. Springer, 2021.

[24] Michalis Kokologiannakis and Viktor Vafeiadis. *GenMC: A Model Checker for Weak Memory Models*, pages 427–440. 07 2021.

[25] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[26] Antoni Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 278–324, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[27] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 446–455, New York, NY, USA, 2007. Association for Computing Machinery.

[28] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 485–502, 2012.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every written paper or thesis authored during the course of studies. **In consultation with the supervisor**, one of the following two options must be selected:

☒ I hereby declare that I authored the work in question independently, i.e. that no one helped me to author it. Suggestions from the supervisor regarding language and content are excepted. I used no generative artificial intelligence technologies[1].

☐ I hereby declare that I authored the work in question independently. In doing so I only used the authorised aids, which included suggestions from the supervisor regarding language and content and generative artificial intelligence technologies. The use of the latter and the respective source declarations proceeded in consultation with the supervisor.

Title of paper or thesis:

Symmetry and Barrier Reduction in GenMC

Authored by:
If the work was compiled in a group, the names of all authors are required.

Last name(s):	**First name(s):**
Turdo	Morris Marcus

With my signature I confirm the following:
- I have adhered to the rules set out in the Citation Guidelines.
- I have documented all methods, data and processes truthfully and fully.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for originality.

Place, date	**Signature(s)**
Mägenwil, 14.07.2025	

If the work was compiled in a group, the names of all authors are required. Through their signatures they vouch jointly for the entire content of the written work.

[1] For further information please consult the ETH Zurich websites, e.g. https://ethz.ch/en/the-eth-zurich/education/ai-in-education.html and https://library.ethz.ch/en/researching-and-publishing/scientific-writing-at-eth-zurich.html (subject to change).