LLM-Driven Synthesis of Concurrent Data Structures with SMR under Weak Memory

Alexandru Dumitriu

Thesis submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

Programming Languages Group

Department of Software Technology

Faculty EEMCS, Delft University of Technology

Delft, Netherlands

Abstract

Developing correct concurrent data structures under weak memory models presents significant challenges due to subtle concurrency errors arising from relaxed ordering guarantees and complexities in Safe Memory Reclamation. Existing synthesis methods largely assume sequential consistency, overlooking critical reorderings allowed by realistic architectures.

This thesis introduces a synthesis-verification pipeline that iteratively generates concurrent data structures from partial code specifications using Large Language Models. The pipeline is expanded by integrating an advanced model checker, GenMC, enhanced specifically to verify SMR correctness under weak memory through automaton-based hazard pointer verification. This integration provides memory safety guarantees across diverse execution scenarios.

We evaluate our approach using established concurrent data structure benchmarks, demonstrating rapid convergence to correct implementations, outperforming state-of-the-art methods. These results highlight the pipeline's effectiveness and scalability, illustrating its potential to support researchers in developing novel, reliable concurrent data structures under weak memory models.

Acknowledgements

I would like to express my deepest gratitude to all the people who have supported me throughout this journey. First and foremost, I thank Soham Chakraborty, my daily supervisor, for his guidance and constant availability whenever I needed feedback or advice. I am also extremely grateful to my thesis advisor, Arie van Deursen, whose insightful suggestions challenged me to do my best work. A special thanks goes to Andreea Costea for serving on my thesis committee.

To my parents, Elena Simona Dumitriu and Eugen Dumitriu, thank you for your love, sacrifices, and unwavering support. I could not have wished for better parents. Thank you for teaching me that true achievement requires hard work and dedication.

Finally, to those closest to me, thank you for your love, encouragement, and for always being there for me over the years. Your presence in my life has always been a constant source of happiness and motivation to be the best version of myself.

This thesis would not have been possible without each of you. Thank you.

Table of Contents

1 Introduction

Developing concurrent data structures with specific properties presents significant challenges due to complexities in ensuring thread safety, synchronisation, and correctness [1]. Modern hardware architectures frequently operate under weak memory models, where reordering of memory operations invalidates assumptions commonly made under Sequential Consistency (SC) [2, 3, 4]. As a result, even well-designed structures could have hidden race conditions or performance anomalies that appear only under particular interleavings. Additionally, lock-free (non-blocking) designs face difficulties in safely reclaiming memory from deleted dynamic nodes [5]. These challenges considerably slow development processes, as researchers must rigorously verify designs across diverse execution behaviours.

Despite advancements in automated synthesis techniques, existing approaches largely assume sequential consistency, thereby overlooking subtle yet critical reorderings allowed under weak memory semantics. Separately, they often fail to account for Safe Memory Reclamation (SMR), missing opportunities for memory reuse and ABA prevention using techniques that require no operating system support [5]. Lastly, traditional methods rely heavily on exhaustive search or manual refinements, becoming computationally impractical for nontrivial structures involving complex memory operations. This *gap* reveals a significant research opportunity: efficient synthesis of correct concurrent data structures leveraging SMR under realistic weak memory settings. To address this opportunity, we adopt an efficient approach proven to excel at program synthesis, namely Large Language Models (LLMs) trained on massive text and code corpora [6].

We choose Large Language Models (LLMs) trained extensively on vast text and code repositories because they inherently capture complex coding patterns, including subtle concurrency mechanisms and memory-ordering usages [7]. These models encode both syntactic structures and semantic behaviours from concurrent programming examples encountered during training. Leveraging statistical inference, LLMs generate plausible code completions without explicitly evaluating each possible permutation. By implicitly encoding realistic weak memory behaviours through exposure to relevant concurrent code, LLMs efficiently synthesise implementations consistent with weak memory semantics, significantly reducing computational complexity.

This thesis investigates the following central question:

> *How can Large Language Models and advanced verification tools integrate effectively to synthesise concurrent data structure implementations targeting weak memory models?*

We investigate four sub-questions to address this:

- **RQ1:** How can stateless model checking be adapted to verify against a specification concurrent data structures that employ SMR techniques under weak memory models?
- **RQ2:** What prompting techniques for LLMs should researchers use to generate functionally correct C code?
- **RQ3:** How effectively can LLMs synthesise low-level concurrent C code in a zero-shot setting?
- **RQ4:** How does incorporating structured verifier feedback into LLM prompts affect convergence to correct concurrent data structure implementations?

To bridge this gap, we introduce a **pipeline**, as shown in Figure 1, that unites **LLM**-based synthesis with an **SMR**-aware extension of a model checker to form an **agentic feedback loop** tailored for weak memory. Instead of relying solely on exhaustive search or manual refinements, our pipeline utilises the generative capabilities of LLMs to complete partially specified code. We use **GenMC**, a specialised stateless model checker that we extend with SMR support, to verify all potential program behaviours and memory reclamation operations, thus providing robust correctness guarantees under weak memory models [8]. Far from acting as a static verifier, GenMC operates as an autonomous agent, reflecting the recent development of **agentic AI** in software analysis, by monitoring synthesised code, detecting concurrency violations, and providing structured feedback via an **Execution Graph** for iterative improvement [9]. By combining synthesis and verification, our pipeline substantially speeds up convergence times compared to traditional manual or exhaustive approaches, enabling rapid exploration of novel concurrent data structures. The full source code of the pipeline is available as an open-source artifact*.

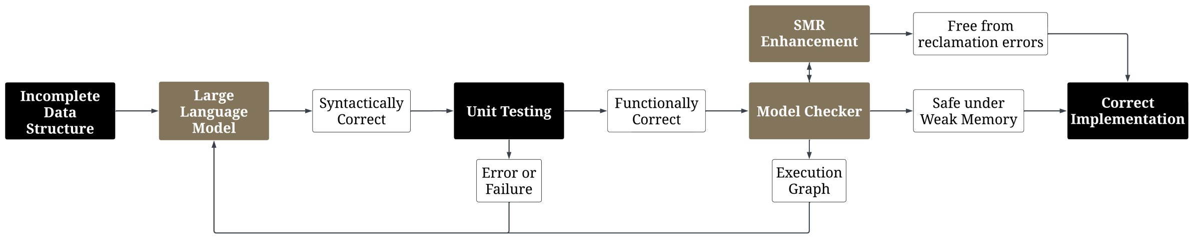


Figure 1: Overview of the proposed program synthesis & verification pipeline.

The remainder of this thesis unfolds as follows. Chapter 2 provides background on weak memory concurrency, focusing on the memory accesses in C/C++. Chapter 3 surveys prior work on concurrent data structure synthesis, highlighting sketch-based and ASP-based methods and their limitations under weak memory models. Chapter 4 addresses **RQ1** by detailing the enhancements made to GenMC for verifying SMR via automaton-based reasoning, demonstrating correctness checks on implementations like Michael and Scott Queue and Harris List [10, 11]. Chapter 5 details our synthesis–verification pipeline, including LLM-based prompt strategies (**RQ2**) and iterative refinement mechanisms. This chapter also defines evaluation metrics for model accuracy and convergence rate and speed. Finally, Chapter 6 presents the empirical results: evaluating LLM synthesis accuracy (**RQ3**) and measuring the impact of structured verifier feedback (**RQ4**), and benchmarking our pipeline's end-to-end performance against state-of-the-art solvers. The thesis concludes in Chapter 7 with a summary of findings, limitations discussion, and suggestions for future research in automated concurrent data structure design.

*`https://github.com/Alexandru-Dumitriu/ProgramSynthesisVerifier`

2 Background

This section discusses the foundations upon which the remainder of this work is built. We begin by reviewing sequential consistency and emphasising how performance-driven compiler optimisations led to weak-memory models. Subsequently, we present execution graphs and utilise them in a two-threaded program, showcasing both SC and non-SC behaviours. Expanding upon this visual framework, we describe the fundamentals of the C11 model and examine a Treiber-stack `pop` operation to exemplify release, acquire and relaxed synchronisations in real-life applications. Lastly, we examine non-blocking structures and the ABA problem prior to formalising Safe Memory Reclamation approaches that address it and their limitations. These foundations provide the reader with the necessary terminology and formal tools to comprehend the verification and synthesis methods presented in subsequent sections.

2.1 From Sequential Consistency to Weak Memory

Modern computer systems exploit thread-level parallelism to meet ever-growing performance and responsiveness requirements. In concurrent programming, multiple threads execute simultaneously, often interacting through shared variables in memory. The interactions between these threads can result in subtle and complex behaviours, making the correctness verification of concurrent programs a challenging task, especially when these interactions are influenced by hardware caches and compiler optimisations.

Sequential Consistency The simplest and most intuitive theoretical model for reasoning about concurrent programs is sequential consistency, first formalised by Lamport, and it remains the conceptual baseline used in most introductory texts [12]. Under sequential consistency, the memory operations of all threads can be understood as occurring in a single, global total order consistent with the program order of each individual thread. Therefore, verifying a concurrent algorithm reduces to checking one total order rather than many possible reorderings. This makes reasoning about correctness relatively straightforward, as programmers can reason about one interleaved sequence of operations, without having to account for the reorderings that real machines may perform.

Weak Memory Models However, practical considerations and performance optimisations in modern hardware and compilers typically relax these strong ordering guarantees, resulting in weak memory models. Under weak memory, different threads can observe memory operations in different orders, due to various forms of allowed reordering and buffering at both the hardware and compiler levels. Examples of *weak memory models* include TSO (Total Store Order, used by Intel architectures), ARMv8, POWER, and more abstract models used for language standards such as the C/C++ memory model (C11/RC11) [3, 4, 2, 13]. Such models bring further complexity, as concurrent programs may display erroneous behaviours that are impossible under SC. We formalise these relaxed behaviours and illustrate their consequences with an example program in the following subsection.

2.2 Execution Graphs

Execution graphs provide a visual representation of a concurrent program's behaviour under a particular interleaving and memory model. Each graph shows the execution as a set of events (reads and writes) together with edges that capture the ordering between those events. By inspecting the shape of the graph, whether edges form cycles or whether a read observes a given write, we can decide if a behaviour is allowed or forbidden by the chosen memory model.

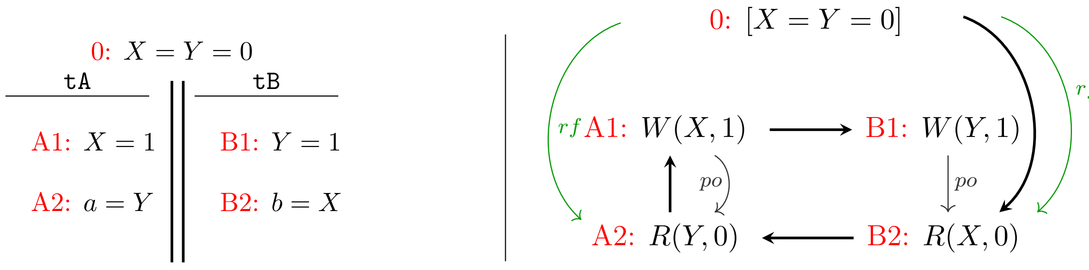
Execution Graph Semantics An *execution graph* G is a pair $G = (E, R)$.

- E is a finite set of *events*. We write an event as (i, n, ℓ) where
  - $i \in \mathbb{N}$ is the thread identifier,
  - $n \in \mathbb{N}$ is the position of the event in that thread's program order,
  - ℓ is a label such as $W(X, 1)$ for a write of value 1 to X or $R(Y, 0)$ for a read of value 0 from Y.
- R is a set of binary relations over E. In this section we use only po and rf, described in Table 1, reserving additional relations for Section 4.3.

Relation	Meaning
po	**Program order** preserves the order in which events appear in the same thread's source code, drawn as a grey arrow.
rf	**Reads-from** links a read to the write whose value it observes, drawn as a green arrow.

Table 1: Execution graph relations used in this section. Other relations will be introduced in the subsequent sections as required.

Weak Memory Program Example Figure 2 shows a weak-memory program on the left. The program consists of two threads: thread `tA` writes to `X` and then reads from `Y`, while thread `tB` writes to `Y` and then reads from `X`. The surprising result $a = 0 \wedge b = 0$, shown on the right, is impossible under sequential consistency (SC), as it would require that both reads occur before their corresponding writes in a way that cannot be linearised. Specifically, for `a = 0`, the read from `Y` must happen before `B1`, and similarly for `b = 0`, the read from `X` must happen before `A1`, which creates a cyclic dependency. However, under a weak memory model, such as C11, compilers and processors are allowed to reorder instructions across different memory locations [13]. This makes the outcome $a = 0 \wedge b = 0$ possible if, as in the depicted example, the write and read in `tA` and `tB` are reordered.



```
          0: X = Y = 0
     tA          ||      tB
 A1: X = 1       ||  B1: Y = 1
 A2: a = Y       ||  B2: b = X
```

Figure 2: Weak Memory Program Example (left) and a Non-SC Execution Graph (right). *tA* and *tB* represent two concurrent threads. *0, A1, A2, B1, B2* represent executed events. Program order (po) edges are drawn as grey arrows, while reads-from (rf) edges are rendered in green to emphasise data dependencies. Solid black arrows indicate the execution order. We write *W(X, 1)* for a write of value 1 to X and *R(Y, 0)* for a read of value 0 from Y.

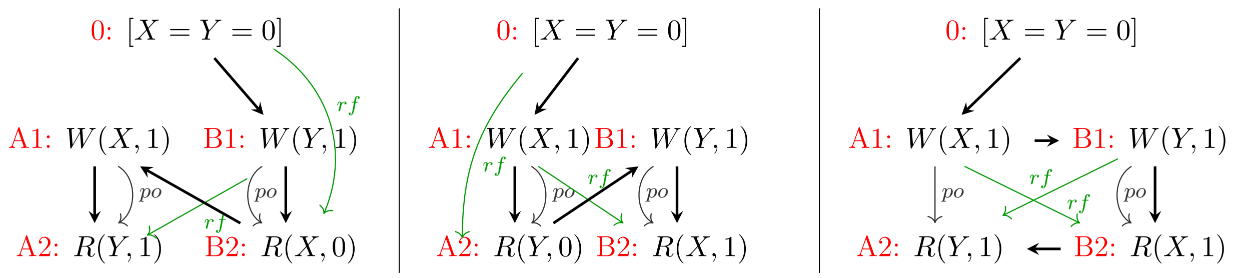


Figure 3: The Execution Graphs of the program depicted in Figure 2 resulting in the 3 possible outcomes under Sequential Consistency. The semantics used are the same as in Figure 2.

By contrast to the previously explained execution graph in Figure 2 which reflects non-SC behaviour, Figure 3 shows three valid execution graphs under SC, corresponding to the outcomes $a = 1 \land b = 0$, $a = 0 \land b = 1$ and $a = 1 \land b = 1$. Each of these preserves the thread-local program order and adheres to a global execution order that avoids cycles. These examples illustrate how execution graphs can capture both allowed and disallowed behaviours depending on the memory model, highlighting the importance of verification in weak memory settings.

2.3 C/C++ Concurrency (C11) Essentials

This subsection clarifies the necessary concepts for understanding the ordering guarantees involved in the synthesised implementations, as the code generated with the assistance of LLMs is reliant upon C11 memory-order semantics. We begin by defining atomic operations and the memory-order tags, subsequently applying these concepts to the Treiber-stack pop implementation outputted by our pipeline.

Atomicity & RMW Operations In C11, an *atomic* operation on a shared variable behaves as a single, indivisible event: no other thread can observe a partial result. A common example is a read-modify-write (RMW) instruction, such as

```
atomic_compare_exchange_strong_explicit(&x, &expected, desired, mo, moFail),
```

where:

- `x` is the target atomic variable.
- `expected` holds the value we believe `x` currently has.
- `desired` is the new value we want to store if `x` equals `expected`.
- `mo` and `moFail` are memory-order tags that specify ordering guarantees on success or failure. These tags will be explained in the following paragraph.

Conceptually, an RMW event executes in three logical stages:

$$\underbrace{\text{read old value}}_{\text{step 1}} \longrightarrow \underbrace{\text{compute new value}}_{\text{step 2}} \longrightarrow \underbrace{\text{write new value}}_{\text{step 3}}.$$

To every other thread, these three steps appear to occur simultaneously, preventing any interleaving that could observe a *half-update*. As a result, once the RMW operation completes, either the old value remains (if the comparison failed) or the new value is fully in place, with no intermediate state visible. This indivisibility is the basis for building higher-level synchronisation guarantees using memory orders.

$$\begin{tikzcd}
 & & \text{Release} \arrow[dr] & & \\
\text{Non-Atomic} \arrow[r] & \text{Relaxed} \arrow[ur] \arrow[dr] & & \text{Release-Acquire} \arrow[r] & \text{SC} \\
 & & \text{Acquire} \arrow[ur] & &
\end{tikzcd}$$

Figure 4: Memory-access orderings under C11, from weakest to strongest guarantees.

Memory Accesses In concurrent C programming, threads communicate through shared memory using operations such as reads and writes. Each atomic operation sets a memory order that controls how its effects are seen by other threads, an essential aspect for ensuring correctness in lock-free data structures. Figure 4 outlines the C11 memory orders, which can be further described from the weakest to the strongest as follows:

- **Relaxed**: enforces only atomicity of the single operation, with no ordering or visibility guarantees beyond preventing incomplete reads or writes.
- **Acquire**: prevents any subsequent memory accesses (reads or writes) in the same thread from being reordered before this acquire operation; it synchronises with a matching **Release** store in another thread, ensuring visibility of prior writes.
- **Release**: prevents any preceding memory accesses (reads or writes) in the same thread from being reordered after this release operation; it ensures that all prior writes become visible to any thread that performs a matching acquire load.
- **Acquire-Release**: combines acquire semantics on load and release semantics on store in one operation that prohibits reordering of memory accesses around it.
- **Sequentially Consistent**: provides the strongest guarantee by enforcing a single global order of all SC-ordered operations across all threads, making the execution appear as if every operation took effect in some interleaved total order consistent with program order.

Concurrent Data Structure Implementation Example In our pipeline, the user can supply a partial Treiber-stack pop implementation with *holes* (marked in yellow) where operations with important memory-order annotations belong. The synthesis tool then fills these holes (shown in orange) with the correct atomic operations and memory-order tags. Algorithm 1 illustrates one such input/output pair: the initial part marks with "???" the unknown code, and the following lines showcase how our tool populates the hole.

Algorithm 1: Treiber-stack `pop` with a user-provided "hole" (yellow) and synthesised operations with memory-order annotations (orange).

```
Function pop
    s : mystack_t*
pointer oldTop, newTop, next;
node_t *node;
bool success;
int val;
while true do
    // hole: missing memory operations (marked with ???)
    ???
    // hole filled: acquire load from top
    oldTop = atomic_load_explicit(&s->top, acquire);
    if get_ptr(oldTop) == 0 then
        return 0;
    node = &s->nodes[get_ptr(oldTop)];
    // hole filled: relaxed load from node->next
    next = atomic_load_explicit(&node->next, relaxed);
    newTop = MAKE_POINTER(get_ptr(next), get_count(oldTop) + 1);
    // hole filled: compare-exchange with release and relaxed
    success = atomic_compare_exchange_strong_explicit( &s->top, &oldTop,
     newTop, release, relaxed);
    if success then
        break;
val = node->value;
// Reclaim the used slot
reclaim(get_ptr(oldTop));
return val;
```

Next, we deep dive into the specific justifications for each memory-order tag in the loop, explaining how each annotation affects the visibility of shared variables, enforces proper ordering of operations, and ultimately ensures a correct `pop` function call in C/C++.

1. Acquire load on `top`:

```
oldTop = atomic_load_explicit(&s->top, acquire)
```

We use **acquire** here to ensure that, if another thread has performed a corresponding **release** store to `top`, then all writes that preceded that release are visible in this thread. This can be translated into the following behaviours:

- If Thread A successfully published a new head pointer `newHead` (e.g., fully executing `atomic_store_explicit(&s->top, newHead, release)`), then Thread B's execution of `atomic_load_explicit(&s->top, acquire)` is guaranteed to observe `newHead` (or a later pointer). Without **acquire**, Thread B could read an outdated or partially updated node, leading to incorrect data retrieval or even undefined behaviour.
- The acquire semantics introduce a synchronisation point: no memory access in the current thread can be reordered before this load. This guarantees that subsequent lines (such as reading `&node->next`) occur only after the correct pointer is obtained.

2. Relaxed load on `node->next`:

```
next = atomic_load_explicit(&node->next, relaxed)
```

Once we have performed the acquire load on `top`, we already know that all writes to `node->next` made by the thread that inserted the node are visible, leading us to:

- A **relaxed** load suffices to read `node->next` because no further inter-thread ordering is required due to the earlier acquire having already synchronised with the publisher's release, guaranteeing visibility.
- Using **relaxed** here avoids imposing extra memory-fence overhead, improving performance.

3. Compare-and-exchange on `top` with release/failure-relaxed:

```
success = atomic_compare_exchange_strong_explicit(
  &s->top, &oldTop, newTop, release, relaxed);
```

In this final step, we attempt to atomically "pop" the node by replacing the head pointer `oldTop` with `newTop`. We choose:

- **release** on the successful store. This ensures that all memory writes that occurred before this compare-and-exchange become visible to any thread that later performs an **acquire** load on `top`. Additionally, the release prevents any preceding memory operations from being reordered past the successful store, therefore sharing the popped state with the other threads.
- **relaxed** on failure. If the compare-and-exchange fails (because another thread popped the node first), we do not require any additional ordering and we just retry the loop. Using **relaxed** in the failure case minimises unnecessary fences.

By filling the *hole* with precisely these memory orders: **acquire** for the head load, **relaxed** for reading the next pointer, and **release** (with **relaxed** on failure) for the compare-and-exchange, we guarantee that each pop operation synchronises correctly with concurrent push or pop operations on other threads. This input/output example highlights the complexity of the task our synthesis pipeline approaches, transforming a partial Treiber-stack `pop` function into an C11-correct implementation.

2.4 Lock-Free Data Structures

A shared object is considered lock-free (or non-blocking) if, when multiple threads concurrently attempt operations on the object, the system as a whole ensures that at least one thread completes its operation in a finite number of steps, even if other threads experience arbitrary delays or failures [14]. This property prevents lock-free objects from deadlocking, since at least one thread always makes headway regardless of others' states. Furthermore, lock-free implementations often deliver strong performance under high contention, as delayed or stalled threads do not block overall progress. In the dynamic setting, lock-free data structure implementations for stacks or queues combine the non-blocking properties with the flexibility of being unbounded, and several designs have been proposed to realise these benefits in practice [10, 11, 15].

The ABA Problem Most lock-free objects rely on the atomic compare-and-swap (CAS) instruction, such as `atomic_compare_exchange_strong` in C11, to perform updates without locks [16]. CAS, a specific type of RMW, atomically reads a value, compares it to an expected value, and only writes a new value if the comparison succeeds. However, CAS is susceptible to the ABA problem: if a location which reads value `A` is changed to `B` by another thread, and then changed back to `A` before the original CAS, the operation will erroneously succeed, considering no change occurred [17]. One case in which this issue arises is when removing an item from a lock-free list: the node is deallocated, and a new node is later allocated at the same address (due to most-recently-used memory allocation). In that scenario, a pointer to the new item equals the pointer to the old one, causing an ABA failure. Consequently, even though lock-free designs avoid deadlock and provide system-wide progress, they must address ABA to prevent subtle correctness violations.

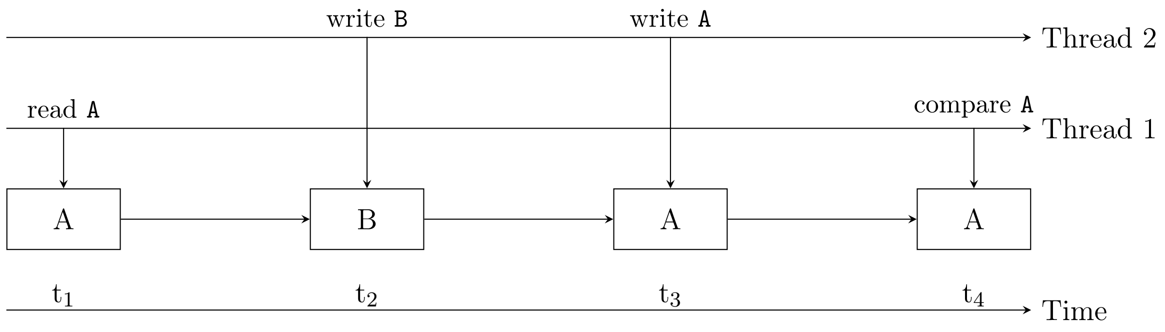


Figure 5: Timeline of the ABA scenario on a shared pointer. Thread 1 reads `A` at t_1, Thread 2 changes `A`→`B` at t_2, then `B`→`A` at t_3, and Thread 1's compare at t_4 still sees `A`.

To help paint a clearer picture, Figure 5 illustrates a typical ABA situation on a shared pointer `ptr`. Thread 1 examines the pointer at time t_1 and sees the value `A`. Thread 2 performs two modifications before Thread 1 continues to do a CAS. At time t_2, it changes `ptr` to `B`, and at time t_3, it changes it back to `A`. Thread 1 resumes at time t_4 and does its CAS. It still sees `A`, so it thinks that `ptr` was never changed. The pointer has actually changed twice, which could mean that Thread 1 may access a node that has already been removed or repurposed. This hidden change can lead to memory corruption or logical errors in the data structure, since Thread 1 operates under the false assumption that no other thread intervened.

2.5 Safe Memory Reclamation

As previously mentioned, the design of lock-free data structures offers robustness in the face of concurrent operations and thread failures. However, these benefits come at a significant cost regarding dynamic memory management. In contrast to lock-based designs, where mutual exclusion ensures safe node reclamation, lock-free algorithms must coordinate memory reuse without relying on external synchronisation. The fundamental challenge is ensuring that memory reclaimed by one thread is no longer accessible to others. This is the domain of SMR techniques, which provide mechanisms to safely reuse memory without introducing use-after-free memory errors (dereferencing a pointer whose memory has already been freed) caused by ABA sequences [5].

An effective SMR technique must ensure that no node is reclaimed while it may still be accessed. In lock-based algorithms, this guarantee is trivial: only the thread holding the lock can safely free removed nodes. In lock-free algorithms, however, the lack of mutual exclusion means there may be a delay between when a node is removed from the shared structure and when other threads stop accessing it. A thread might have read a reference to the node just before it was removed, potentially leading to a use-after-free memory error if reclamation is not properly coordinated. SMR schemes address this by delaying reclamation until safety can be established. However, designing SMR mechanisms that are both correct and performant is notoriously difficult and remains an active field of research [18, 19, 20, 21].

Hazard Pointers Among the earliest and most influential practical SMR schemes is Michael's hazard pointer algorithm [5]. Hazard pointers allow threads to explicitly publish the addresses they intend to access, preventing those nodes from being reclaimed prematurely. The protocol follows a well-defined lifecycle, illustrated in Figure 6: a node is allocated, then the thread `protects` it by storing its address in a hazard pointer; once the node is unlinked from the shared structure, it is `unprotected` and `retired`. Only after a global scan confirms that no thread's hazard pointer protects it can the node be reclaimed. While hazard pointers effectively prevent ABA by delaying reuse, they have limitations in more complex lock-free designs. In particular, optimistic traversal can break the assumptions needed for safe reclamation.

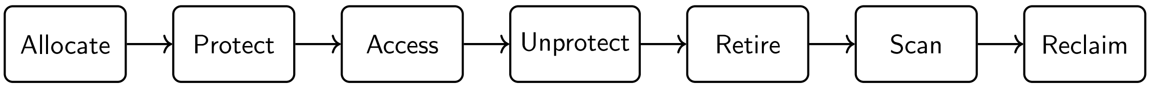


Figure 6: Lifecycle of a hazard pointer-protected memory access. Memory is only reclaimed after a full scan of all thread-local hazard pointers to ensure it is no longer protected.

Limitation - Harris List One data structure using optimistic traversals is the hazard-pointer variant of the Harris linked list, a widely studied lock-free set that uses pointer marking for logical deletion [11]. Although hazard pointers are intended to ensure memory safety, the original Harris design allows nodes to be traversed even after being logically marked for deletion, but before being physically unlinked. As shown in Figure 7, if one thread traverses into a chain of logically deleted nodes while another thread retires and reclaims those same nodes, the first thread may access reclaimed memory, causing the previously described use-after-free error. Prior work identified this issue and proposed fixes that make use of other techniques alongside hazard pointers, highlighting the limitations of this SMR algorithm [22]. The verification component of our tool detects this violation, and an in-depth investigation is conducted in Section 4.6.

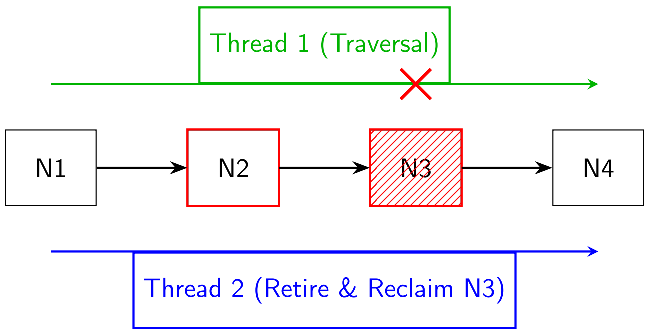


Figure 7: Concurrent traversal and reclamation in a Harris list: Thread 1 accesses N3 during optimistic traversal after Thread 2 has retired and reclaimed it being a logically deleted node.

3 Related Work

Automating the synthesis and verification of concurrent data structures has been approached from multiple angles, each aiming to reduce the extensive manual effort typically required to ensure thread safety, synchronisation, and correctness. Existing techniques can be broadly grouped into traditional methods, such as *sketch-based* concurrency synthesis, exemplified by PSKETCH, and *reasoning-driven* synthesis via Answer Set Programming (ASP), as well as more recent *AI-based approaches* using LLMs [23, 24, 9]. Both PSKETCH and ASP-based methods focus on concurrency guarantees under SC, and cannot synthesise implementations with the memory orderings shown in Algorithm 1 due to the explosion of the search space in terms of the number of possible interleavings. By contrast with both approaches, our work specifically targets concurrency design under *weak memory models*, and, to the best of our knowledge, no existing tool currently specifically addresses the synthesis of concurrent data structures in this setting.

3.1 Sketch-Based Synthesis with PSKETCH

PSKETCH extends the original SKETCH language and synthesis framework to handle concurrency by allowing developers or concurrency researchers to write *partial programs* (sketches) in which critical operations, such as atomic updates, lock placements, and pointer manipulations, remain unspecified [23]. These deliberately incomplete portions of code are called *holes*, and each hole indicates a missing expression, statement ordering, or memory-access pattern that the synthesis engine must resolve. Rather than forcing the user to fully explain where and how concurrency is enforced, the PSKETCH language provides additional constructs for automatically filling in these holes. This mechanism relieves the programmer from detailing low-level synchronisation at every step, while retaining control over the rough outline of the algorithm.

Once a sketch is provided, PSKETCH systematically completes the holes via *counterexample-guided inductive synthesis* (CEGIS). The user defines a high-level concurrency structure (such as a lock-free queue) without specifying the precise ordering or arrangement of operations. In each iteration, the system enumerates candidate completions for the holes and checks them against an SC-based correctness specification using a model checker. If a candidate fails due to a data race, memory corruption, or deadlock, the model checker returns a counterexample execution trace. PSKETCH uses this trace to prune large parts of the search space and refine subsequent attempts, converging on a valid solution in relatively few iterations.

Several key innovations make this feasible. First, *regular-expression generators* let users specify families of pointer expressions or synchronisation steps as a bounded grammar. Second, *reorder blocks* allow statement sequences to be permuted in any order the synthesiser deems correct, which is particularly useful when deciding the exact order in which locks, unlocks, or atomic instructions should appear. Third, concurrency primitives (e.g. threads, lock/unlock) are compiled into guarded atomic regions that the model checker analyses for concurrency errors. These constructs can collectively represent extremely large candidate spaces (10^8 order or more).

Limitations While this approach drastically reduces the manual burden of enumerating concurrency strategies, particularly in designs like lock-free queues, it also has recognised limitations. PSKETCH assumes *sequential consistency* when verifying each candidate, thus omitting reorderings permitted by weaker memory architectures. Moreover, although CEGIS prunes failing candidates, the initial candidate pool can still be immense, and generating or compiling the resulting verifiers for each iteration may become expensive in large-scale sketches. Nonetheless, PSKETCH has demonstrated an ability to synthesise challenging concurrent data structures from concise partial descriptions, highlighting the power of holes: developers specify only the high-level concurrency insights and rely on the synthesiser to fill in crucial details.

3.2 ASP-Based Reasoning-Driven Synthesis

A second branch of work explores how *reasoning frameworks* such as Answer Set Programming (ASP) can synthesise thread-safe code directly from sequential data structure definitions. In the methodology of Varanasi *et al.*, the user encodes structural invariants (*e.g.*, reachability or acyclicity) and concurrency axioms (*e.g.*, when a node must be locked before modification) as first-order logic rules [24]. These axioms capture both the sequential semantics of the data structure and the thread-interference scenarios that arise when multiple operations are executed in parallel. ASP solvers, which can efficiently explore large combinatorial spaces under logical constraints, use these axioms to infer synchronisation strategies, such as lock placements or atomic updates, that uphold the specified invariants under SC-based interleavings.

A key insight is that every sequential step (*e.g.*, linking a new node into a list) can be *lifted* into a concurrent step that incorporates additional logic. If the solver identifies that an update risks being broken by concurrent mutations, it prescribes lock acquisition or RCU-style validation to safeguard the data structure invariants. In essence, this framework systematically emulates the expert reasoning required to ensure that a pointer-based structure can withstand overlapping inserts, deletes, and lookups. Tools like Locksynth automate these reasoning tasks by taking high-level knowledge about pointer representations, concurrency axioms, and sequential code blocks, then outputting a correct concurrent variant that acquires the *right* number of locks or performs the *right* validations.

Limitations While ASP-based synthesis can thereby uncover delicate concurrency flaws and prove correctness for highly optimised data structures, creating the encodings demands considerable low-level expertise. Detailed axioms must describe pointer reachability, permissible reorderings of node modifications, and potential concurrency hazards such as stale references or concurrent deallocations, all of which require a deep understanding of both concurrency theory and the target data structure. Moreover, as with sketch-based techniques, the default ASP solver operates under sequential consistency assumptions and does not automatically account for weaker memory models. Although extending the axioms could incorporate additional reorderings, it increases the complexity of the logical specification and requires further architectural knowledge. Nonetheless, for researchers aiming to develop novel data structures, ASP reasoning offers a powerful way to derive concurrency safety without manually enumerating or verifying every possible synchronisation scheme.

3.3 Summary

Sketch-based and ASP-driven concurrency methods lead to a decrease in the manual effort involved in designing sophisticated data structures, but their reliance on sequential consistency leaves them vulnerable to reordering issues on real hardware. Moreover, these frameworks typically employ exhaustive or combinatorially large explorations whose complexity escalates drastically for more advanced designs, despite pruning heuristics. By contrast, our approach integrates a *weak memory model verifier* to overcome the SC limitation and ensures correctness under realistic reorderings, while an LLM drives a *dynamic* exploration of the search space. Rather than exhaustively enumerating thousands of partial programs or carefully crafting specialised axioms, we iteratively refine designs based on discovered concurrency violations, thereby avoiding the exponential increase in search and making the synthesis pipeline more scalable and broadly accessible.

4 Enhancing GenMC with SMR Verification for Weak Memory Models

In order to thoroughly address our first research question (**RQ1**): *How can stateless model checking be adapted to verify against a specification concurrent data structures that employ SMR techniques under weak memory models?*, this chapter is structured to provide a clear progression from foundational concepts to specific implementations and results. The following sections comprehensively explore the background of program verification under weak memory, the capabilities and architecture of GenMC and of the adapted GenMC model checker, formalising SMR through automata, and our novel integration of this automaton into GenMC [8, 25, 18]. Finally, we conclude with experimental validations that demonstrate the effectiveness of our enhancements.

4.1 Weak Memory Program Verification

Verifying concurrent programs under weak memory models is challenging due to the relaxed ordering guarantees these models offer. Traditional SC assumptions no longer hold, as previously detailed in Section 2.1, requiring sophisticated automated verification approaches.

Fuzzing Automated verification primarily involves fuzzing and model checking. Fuzzing executes the program numerous times with random inputs or thread schedules, efficiently discovering certain types of errors [26, 27]. However, fuzzers provide no exhaustive guarantees and may miss infrequent or subtle concurrency issues.

Stateful vs. Stateless Model Checking In contrast, model checking systematically explores all potential program behaviours under a specified memory model, guaranteeing exhaustive verification [2, 28]. Model checkers can be categorised as stateful or stateless. Stateful model checkers maintain explicit records of previously explored states to avoid redundancy. Although effective, their applicability is severely limited by state space explosion, especially under weak memory conditions, making them impractical for large-scale verification tasks.

Stateless model checkers address this issue by systematically enumerating executions without explicitly storing the state [29, 30]. They utilise execution graphs, which represent program behaviours through nodes (events such as loads and stores) and edges (relations like po and rf), as outlined earlier in Section 2.2. Stateless model checkers efficiently navigate these graphs to thoroughly explore allowed interactions under weak memory models.

4.2 GenMC

GenMC is an advanced stateless model checker explicitly developed for weak memory concurrency verification [8]. It incrementally constructs execution graphs from concurrent C/C++ programs, continuously verifying the consistency of each execution under the chosen weak memory model (e.g., RC11). GenMC keeps track of alternative execution scenarios arising from different rf choices, maintaining these alternatives in a structured work queue. Rather than storing separate graphs for every possible execution, GenMC records only the read–write pairs that must be revisited, significantly reducing memory overhead. Its revisit-based exploration uses the work queue to record read–write pairs and distinguishes *forward revisits* (triggered by new reads) from *backward revisits* (arising from writes), marking the latter as explored to prevent redundant execution. By pruning redundant revisits, GenMC achieves optimality (each execution is visited exactly once) while maintaining soundness and completeness in its verification [8].

Verification Algorithm GenMC begins with a graph containing only the initial writes and memory state. At each step, it chooses the next program event (a write or a read) and adds it to the graph, checking consistency under RC11. Whenever a load can legally observe multiple prior writes, GenMC leaves the current rf edge in place and enqueues any alternative write(s) as revisits. Each revisit is tagged as a forward or backward revisit. Forward revisits, once processed, are removed from the queue; backward revisits are marked "explored" but remain in the queue until all intervening events have been revisited to avoid duplicate exploration. This loop repeats until the work queue is empty (all rf choices have been tried) or a protocol violation is detected.

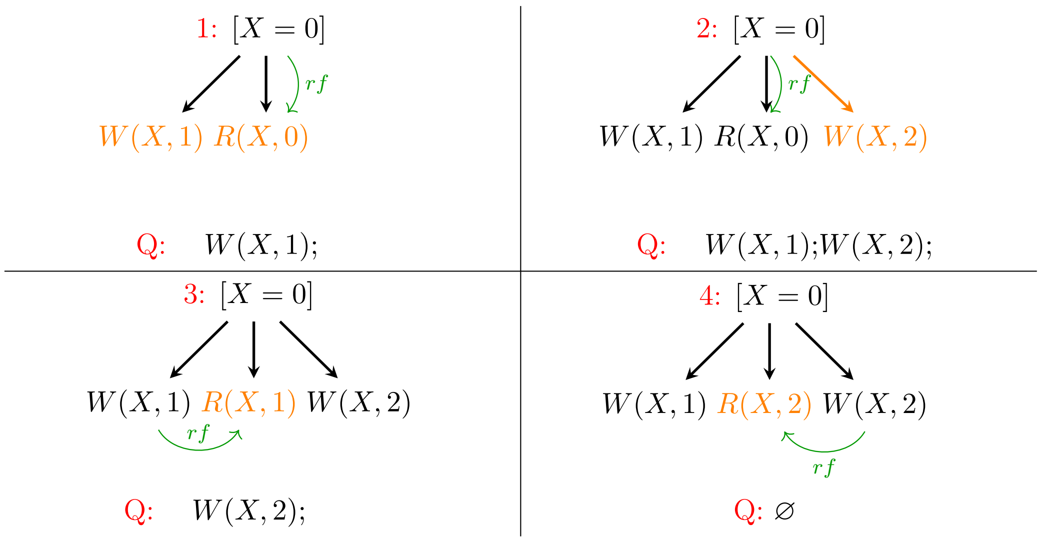


Figure 8: Step-by-Step Example of GenMC building an Execution Graph. Each stage is numbered (1-4), the new events or changes are highlighted with orange and the exploration queue is continuously updated.

Step-by-Step Example To illustrate how the verification algorithm works in practice, we now examine a simple three-thread example operating on a single shared variable x, as shown in Figure 8:

1. Initially, only the write $W(x,0)$ is present in the graph. Thread T1 issues $W(x,1)$, and then Thread T2 performs $R(x)$. Under the default rf edge, T2's read observes $W(x,0)$. Because $W(x,1)$ is also a legal source for that read, GenMC enqueues the pair $\langle R(x), W(x,1)\rangle$ as a forward revisit.

2. Next, Thread T3 performs $W(x,2)$. Since this new write can also satisfy T2's earlier read, GenMC enqueues $\langle R(x), W(x,2)\rangle$ as a backward revisit.

3. GenMC then dequeues the first revisit, $\langle R(x), W(x,1)\rangle$. It updates T2's rf edge to point to $W(x,1)$ and re-executes all events that follow T2's read under this modified rf. No additional revisits arise from this schedule, so $\langle R(x), W(x,1)\rangle$ is removed from the queue.

4. Finally, GenMC dequeues $\langle R(x), W(x,2)\rangle$, switches T2's rf to $W(x,2)$, and re-executes the suffix once more. At this point, the work queue is empty, and all three possible read-from choices $W(x,0)$, $W(x,1)$, and $W(x,2)$ have been explored exactly once.

This example demonstrates how GenMC systematically explores every possible rf choice without revisiting the same execution twice. By enqueuing only the read–write pairs that represent alternative observations, GenMC bounds memory usage and avoids unnecessary graph duplication.

4.3 Adapted GenMC

Building on the stateless execution model earlier described, Henkes extended GenMC to support high-level verification of concurrent data structures [25]. This adaptation introduced function-level reasoning on top of the original instruction-level semantics of GenMC. By lifting verification to the level of function calls, Henkes's tool made it possible to express and check behavioural properties like mutual exclusion and exclusive communication patterns between threads. These properties are particularly relevant for verifying the correctness of data structures such as mutexes, stacks, and queues under weak memory models, whilst unlocking the possibility for future improvements such as SMR Verification.

Function-Level Semantics. The key insight in Henkes' approach is composing instruction-level relations, such as `reads-from` (rf) and `program order` (po), to define abstract function-level relations. These include `communication order` (com), `synchronization order` (so), and `local happens-before` (lhb). To construct these, the checker tracks the start and end of function calls using synthetic marker instructions. It then builds execution graphs in which each function call is treated as a distinct node. Table 2 summarizes these function-level relations alongside the original po and rf, illustrating how com, so, and lhb extend the base instruction-level semantics.

Relation	Meaning
po	**Program order** preserves the order in which events appear in the same thread's source code.
rf	**Reads-from** links each load (read) event to the specific store (write) event whose value it observes, drawn as a green arrow.
com	**Communication order** connects function-call events that exchange data or hand off ownership. It links a call that *publishes* a result to a call that *consumes* it.
so	**Synchronization order** is the union of po and com restricted to function-call events. It captures the cross-thread happens-before relationships induced by communication. In practice, any path consisting of po and a com edge contributes to so.
lhb	**Local happens-before** captures causal order between function-call events via po or so; it is the transitive closure of po ∪ so.

Table 2: Extended set of execution-graph relations after introducing function-level semantics.

Verification Capabilities This adaptation is particularly interesting for our context due to its correctness guarantees and its flexibility. Henkes demonstrates that, with minimal overhead, the tool can accommodate an extensive range of concurrent implementations, from basic mutexes to complex dynamic queues, without any additional modifications to the verification logic. These data-structure specifications, such as mutual exclusion or exclusive communication, are defined directly in terms of the function-level relations (com, so, lhb), allowing high-level properties to be verified. Our tool will leverage these formal specifications extensively when verifying each data structure under weak memory, ensuring that correctness is enforced through the same relation framework. This generality, coupled with the extensibility of the function-level abstraction, positions the tool as a versatile backend for verifying not just classic safety properties but also more domain-specific attributes. For example, the ability to retain function boundaries through inlining is crucial for analyses that depend on whole-function reasoning, such as SMR, which we explore later in this chapter. Integrating Henkes' adaptation into our pipeline provides us with a verification framework that is easily expandable and sufficiently robust to support the verification of intricate behaviours across various concurrency scenarios. For further details on the specifications, the reader is referred to Henkes' thesis [25].

```
<0, 1> threadW:
    (1, 1): Rna (param[0], 4) [(5, 6)] main.c:53
    (1, 2): Wrlx (x[4], 46) main.c:57
    push3  (1, 3): F_CALL
    push3 (1, 4):  Rna (free_lists[4][0], 0) [(5, 5)] my_stack.c:35
<0, 2> threadW:
<0, 3> threadR:
<0, 4> threadR:
<0, 5> threadRW:
    (5, 1): Rna (param[4], 4) [(0, 48)] main.c:80
    (5, 2): Wrlx (x[4], 46) main.c:84
    push2  (5, 3): F_CALL
    push2  (5, 4): Rna (free_lists[4][0], 17) [(0, 32)] my_stack.c:35
    push2 (5, 5):  Wna (free_lists[4][0], 0) my_stack.c:38
    push2  (5, 6): Wna (param[0], 4) my_stack.c:93
```

Listing 1: GenMC output trace for the Treiber-stack implementation which detects a non-atomic race. The highlighted events (1, 4) and (5, 5) are conflicting.

GenMC Trace & Function-Level Execution Graph Example To illustrate how Casper's adaptation detects weak-memory conflicts at the function level, consider the Treiber-stack implementation [15]. In Listing 1, GenMC's execution trace for this implementation is shown, with the two conflicting events $(1, 4)$ and $(5, 5)$ highlighted in yellow. These highlighted lines reveal that the read in **push3** (thread 1, event $(1, 4)$) observes a write performed by **push2** (thread 5, event $(5, 5)$), thereby exposing a non-atomic race on **free_lists[4][0]**. In particular, this conflict arises because event $(5, 5)$ is a non-atomic write (Wna) while event $(1, 4)$ is a non-atomic read (Rna) on the same memory location without any synchronisation.

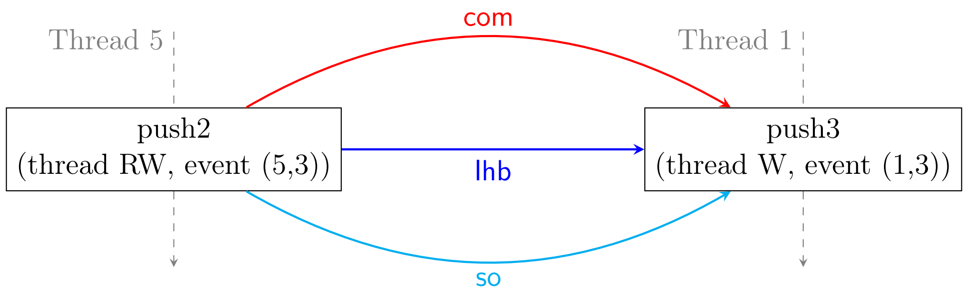


Figure 9: Function-level Execution Graph depicting the GenMC trace of Listing 1. Each node represents a function-call event. The grey dotted arrows represent the threads the functions were executed on, red arrows are com, light-blue arrows are so, and deep-blue arrows are lhb, all flowing from push2 to push3.

Figure 9 translates this trace into a function-level execution graph: each node corresponds to a `F_CALL` event (**push2** and **push3**), and the com edge from **push2** to **push3** directly reflects the function-level read–from relationship in the trace. The so arrow follows because the communication enforces a happens-before ordering between these function calls to ensure correct functionality, and the lhb edge is simply the transitive closure of $\text{po} \cup \text{so}$ at the function level. The data structure was being checked against the weak-stack specification in terms of the previously mentioned relations, and no issues were discovered up to the point of this non-atomic race in terms of respecting the data-structure-specific rules defined by Henkes [25].

4.4 Formalising SMR Correctness with Automata

The correctness of SMR techniques can be expressed as a property over execution traces: a memory address may only be reclaimed once it is provably unreachable and unprotected by any thread [18]. Verifying this property under relaxed memory models requires a formal representation that can track the reclamation-relevant lifecycle of each memory address.

To this end, Wolff adopts a finite-state automaton formalism that models the admissible state transitions of memory addresses within SMR protocols [18]. Each address is associated with an instance of the automaton, and transitions are triggered dynamically during program execution in response to key operations such as pointer protection, retirement, and reclamation. This approach enables compositional and scalable verification by separating data structure logic from SMR safety enforcement.

Automaton Model To express the correctness criteria for Safe Memory Reclamation (SMR), each dynamically allocated memory address is tracked using a finite-state automaton. The automaton serves as a monitor: it consumes a stream of SMR-relevant events and rejects only those traces where memory is not reclaimed safely. Each address is associated with a separate automaton instance that records which threads have protected, retired, or attempted to free the address over time.

Formally, an SMR automaton is a labeled transition system $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of abstract states representing the lifecycle of a memory address (e.g., Allocated, Retired, Protected$_\text{i}$).
- Σ is the event alphabet, consisting of SMR actions: $\mathsf{Protect}_i$, $\mathsf{Unprotect}_i$ and Retire.
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation.
- $q_0 \in Q$ is the initial state, typically Allocated.
- $F \subseteq Q$ is the set of accepting (safe) states, often defined implicitly.

Event Semantics The SMR automaton events are generated by instrumentation inserted at relevant program points:

- $\mathsf{Protect}_i(n)$ — issued when hazard pointer i stores n.
- $\mathsf{Unprotect}_i(n)$ — issued when hazard pointer i clears or overwrites store.
- $\mathsf{Retire}(n)$ — issued when a node n is removed from a shared data structure.

These events form a trace $\tau \in \Sigma^*$ per address, per thread, and $\mathcal{A}$ accepts τ iff the SMR usage of the address is correct. Any trace that leads to an illegal operation (e.g., a retire operation while a thread still protects the node) is rejected.

Example of SMR Automaton Trace To illustrate how the SMR automaton in Figure 10 enforces safety under weak memory, consider a single address n and this sequence of SMR events:

1. From the initial state Allocated $= s_9$, a $\mathsf{Protect}_0(n)$ operation drives the automaton to Protected$_0$ $= s_{10}$.
2. Next, an $\mathsf{Unprotect}_0(n)$ moves the automaton from s_{10} to Unprotected $= s_{11}$.
3. Then, a $\mathsf{Protect}_1(n)$ transitions from s_{11} into Protected$_1$ $= s_{13}$.

At this point, n is still unretired but protected by HP1. Under sequential consistency, no further $\mathsf{Protect}_0$ should occur until n is unprotected by HP1 or has been retired. However, under a weak memory model, a second $\mathsf{Protect}_0(n)$ call (from a later function invocation) might be reordered to execute before the $\mathsf{Unprotect}_1$. Since the automaton has no transition from s_{13} on input $\mathsf{Protect}_0$, attempting that operation in state s_{13} is undefined. Consequently, the automaton immediately rejects the trace, signaling an unsafe pattern where HP0 tries to protect an address held by HP1.

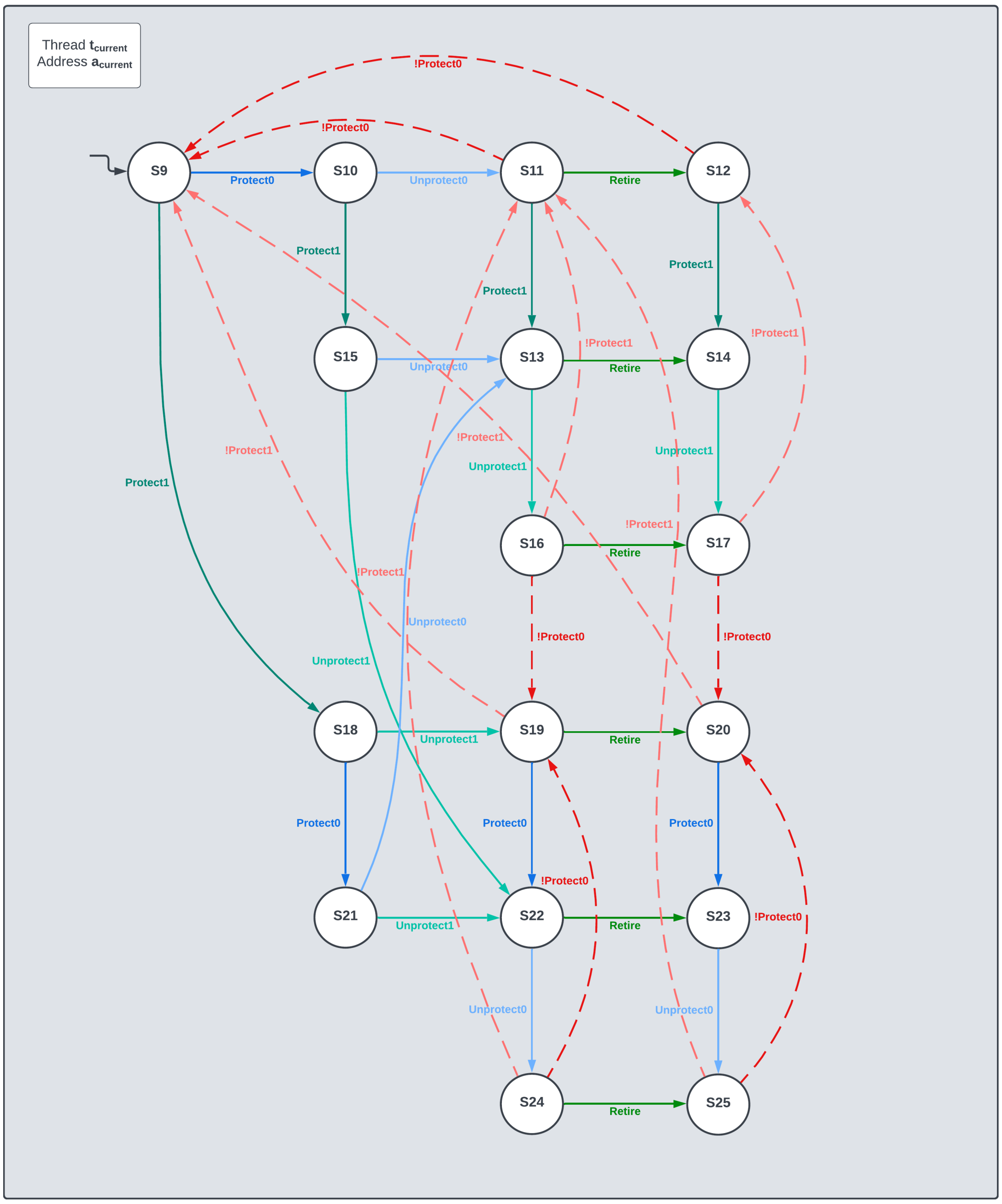


Figure 10: SMR Verification Automaton for two Hazard Pointers implementations designed by Wolff [18]. The automaton models the state evolution of a memory address in one thread. Each state has a state of accepted transitions `Protect_i`, `Unprotect_i` and `Retire` marked through labeled arrows of different colours. Each state can also have `Protect_i!`, marked with dotted lines, as an accepted transition which signals that the memory address can be protected again with the respective HP.

4.5 Further Extending GenMC: Integrating SMR Automata Checks (RQ1)

To verify SMR correctness under weak memory models, we extend **GenMC** with support for **SMR automata checks**. This integration enables the model checker to enforce memory safety properties over dynamically reclaimed nodes by observing protection and reclamation operations.

Extending Function-Level Semantics Our approach builds on the function-level abstraction introduced by Henkes, which lifts GenMC's semantics from instruction-level events to structured function invocations [25]. This abstraction is crucial for SMR verification: hazard pointer operations such as `protect`, `unprotect`, and `reclaim` typically occur within larger functions. Tracking these operations at the function level allows us to directly interpret program behaviour as automaton transitions, using the SMR model introduced earlier.

Each dynamically allocated memory address is associated with an instance of the SMR automaton. As the program executes, GenMC monitors SMR-relevant function calls and generates corresponding events ($\texttt{Protect}_\texttt{i}$, $\texttt{Unprotect}_\texttt{i}$, `Retire`). These events are grouped by memory address and by thread. For each address, we reconstruct a per-thread trace and evaluate it against the automaton, ensuring that all transitions are legal and reflect safe memory usage.

Implementation Sketch To integrate SMR automaton verification into GenMC, we proceed in three main phases: identifying SMR events at the function level, grouping those events by memory address, and then simulating each address's event sequence against the automaton from Figure 10. Below we explain each step, referring back to the SMR automaton.

1. **Collecting Function-Level SMR Events.** To enable precise event capture, we extended GenMC's inlining and function-tagging infrastructure by introducing a dedicated set of SMR-aware function identifiers (e.g., `FN_Protect0`, `FN_Unprotect0`, `FN_Protect1`, `FN_Unprotect1`, `FN_Retire`). During execution-graph construction, we inline and tag all wrapper functions that perform SMR operations. Each time GenMC schedules a call to one of these tagged wrappers, it records the function entry and exit as a single SMR event in the trace. This ensures that every operation appears exactly once in the event list.

2. **Grouping by Address.** Once all SMR-tagged calls have been recorded for a particular GenMC execution, we partition the list of event records by addr. Each unique address n then has its own sequence of SMR events. This grouping corresponds to *instantiating* one automaton $\mathcal{A}_n$ per address n. Because the SMR safety property must hold independently for each reclaimed node, we verify each address in isolation.

3. **Simulating the Automaton.** For each address n, we use its sorted event records to obtain a trace of SMR actions. We then simulate the finite-state automaton $\mathcal{A} = (Q, \Sigma, \delta, s_9, F)$ from Figure 10, where $s_9 = \textsc{Allocated}$ is the initial state. Concretely:

   (a) Initialize the current state $q := s_9$.

   (b) For each σ event, check if $\delta(q, \sigma)$ is defined:

   - If defined, update $q := \delta(q, \sigma)$. For example, if $\sigma = \mathsf{Protect}_0$ and $q = s_9$, then q becomes $s_{10} = \textsc{Protected}_0$.
   - If $\delta(q, \sigma)$ is undefined (e.g., trying $\mathsf{Protect}_0$ from $s_{13} = \textsc{Protected}_1$), the automaton rejects immediately. We record the violation and report the execution graph and the trace of automaton states our address has been in.

   If all events are consumed without hitting an undefined transition, $\mathcal{A}_n$ accepts, meaning that address n was used safely under SMR.

Putting these pieces together yields Algorithm 2, with comments indicating how each line corresponds to the steps above:

Algorithm 2: SMR Verification Algorithm. Each address is validated against the transition rules of the automaton depicted in Figure 10.

```
Function checkSMRTraces
    // 1. Collect SMR Events
    functionCalls = tag_smr_events();
    // 2. Group function calls by address (value) passed as first argument
    foreach f in functionCalls do
        val = f.getVal(0);
        grouped[val].push_back(f);
    foreach (addr, calls) in grouped do
        addrs = {};
        foreach f in calls do
            foreach label in f.labels do
                addr = getReadValue(readAt(label + 1));
                addrs.insert(addr);
        // 3. Simulate each distinct address against automaton
        foreach a in addrs do
            state = initial_state;
            foreach f in calls do
                if f.accesses(a) then
                    event = f.getEvent();
                    if transition(state, event) is invalid then
                        reportViolation(addr, state, event);
                        break;
                    state = transition(state, event);
```

Soundness Proof Sketch We formally establish the soundness of our SMR automaton-based verification approach integrated within GenMC. Soundness guarantees that any trace rejected by our checker represents a violation of the SMR correctness criteria as defined in Section 4.4.

Claim. (Soundness) Let τ be an execution trace generated by GenMC and let $\mathcal{A}$ be the SMR automaton as defined in Figure 10. If the integrated GenMC–SMR checker rejects τ, then τ contains an SMR safety violation.

Proof. We prove soundness by contradiction. Assume the checker rejects a trace τ, but τ is actually SMR-safe. Rejection occurs if and only if the automaton encounters an event e for which the transition from the current state q is undefined.

Formally, the automaton is defined as $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where:

$$\delta : Q \times \Sigma \rightharpoonup Q$$

is a partial transition function encoding precisely the allowed SMR transitions.

A rejection happens if, given a state $q \in Q$ and event $e \in \Sigma$, we have:

$$\delta(q, e) \text{ is undefined.}$$

Since the automaton construction in Figure 10 strictly encodes permissible event orderings, every undefined transition represents explicitly disallowed SMR behavior (e.g., attempting to protect a node already protected by a different hazard pointer without proper unprotection, or retiring a node still under protection).

Given our assumption that trace τ is SMR-safe, all events in τ must match defined transitions in δ, by the construction of $\mathcal{A}$. This directly contradicts our assumption that a rejection (undefined transition) occurred.

This contradiction arises precisely because our instrumentation, implemented through GenMC's function-level semantics, accurately reflects SMR operations, and the automaton explicitly encodes correctness constraints.

Thus, our initial assumption is false. Therefore, if the automaton checker rejects trace τ, trace τ must contain an SMR usage violation, proving the soundness of our checker. □

4.6 Evaluation (RQ1)

This extension directly answers our first research question (**RQ1**): *How can stateless model checking be adapted to verify against a specification concurrent data structures that employ SMR techniques under weak memory models?* By integrating automaton-based SMR reasoning into GenMC's stateless checking loop, we demonstrate that it is feasible to verify SMR safety across relaxed memory executions. In this section, we evaluate the effectiveness of this integration across multiple concurrent data structures. We show that our enhanced model checker can verify memory safety in known-correct implementations and reproduce well-known SMR bugs, providing strong evidence for the applicability of our approach.

Validating Correct Implementations We first consider the Michael and Scott dynamic queue, an efficient lock-free queue that employs two hazard pointers per thread to ensure memory safety [10]. This implementation has been rigorously studied and is widely recognized as correct under weak memory models when hazard pointers are applied properly [31]. Our enhanced GenMC verifies the correctness of this queue `without reporting any violations`.

```
No errors were detected.
Number of complete executions explored: 28902
Number of blocked executions seen: 759830
Total wall-clock time: 413.42s
```

Listing 2: GenMC verification output for the Michael and Scott queue showing no SMR violations.

Listing 2 shows a successful verification run. The absence of violations across extensive search-space exploration provides compelling evidence of the tool's effectiveness in confirming SMR correctness.

Catching Well-Known Bugs Next, we evaluate our tool on a hazard-pointer-augmented variant of the Harris linked list, as introduced previously in Section 2.5. Although hazard pointers are intended to ensure memory safety, the original Harris design permits traversal into logically deleted nodes, as depicted in Figure 7. This behaviour leads to subtle use-after-free errors in the presence of concurrent unlinking and reclamation. Our enhanced GenMC model checker successfully reproduces this known issue, capturing violations caused by improper synchronisation of reclamation and traversal under weak memory conditions. Listing 3 provides the shortened output from the failing run:

```
<0, 3> threadR:
        (3, 11): Rna      (succ, 0x70)    [(0, 15)]
        (3, 26): THREAD_END

<0, 5> threadRW:
        (5, 14): F_CALL    protect15
        (5, 15): Racq      (, 0x70)        [(2, 21)]
        (5, 16): Rna       (hp_next, 0x2000000000000040)   [(0, 15)]
        (5, 17): HP_PROTECT
        (5, 18): F_Ret
            (unprotect and retire 0x70)

Hp: Found one or multiple modelviolations
State 20
```

Listing 3: Shortened GenMC output showing the SMR calls and use-after-free on 0x70.

Analysis of the Detected SMR Violation The erroneous execution from Listing 3 uncovered by **GenMC** using the SMR automaton in the Harris List can be further analysed with the help of Figure 11. GenMC explores an interleaving where a logically deleted node is reclaimed too early due to an imprecise hazard pointer scan. Specifically, writer thread T5 physically removes the marked node at address 0x70, freeing it after confirming that no other thread currently protects them. However, reader thread T3, which is mid-traversal, is protecting `H` (its current node) and 0x50 (the node immediately after), but not the following node 0x70, which it will dereference next.

As a result, although the hazard pointer scan appears safe to the writer, the reader's next dereference accesses 0x70, a node that has already been freed. GenMC's automaton-based SMR check detects this invalid access and reports a violation precisely at *state 20* after having done the following transitions: $\mathsf{Protect}_1(0x70)$, $\mathsf{Unprotect}_1(0x70)$, $\mathsf{Retire}(0x70)$, and not being able to complete another $\mathsf{Protect}_1(0x70)$ according to the automaton specification in Figure 10. This confirms a concrete `use-after-free` scenario in the execution trace.

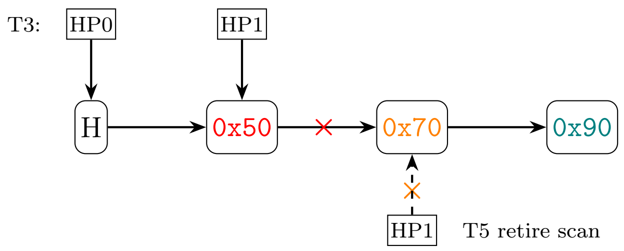


Figure 11: Simplified interleaving: reader T3 needs 0x70 for traversal, but writer T5 freed 0x70.

Summary (RQ1) These results demonstrate our improved GenMC's ability to correctly use SMR and identify errors in concurrent data structures under weak memory models. By integrating automaton-based reasoning into a stateless model checker, we validate the safety of correct implementations, such as the Michael and Scott queue, while also uncovering bugs in hazard-pointer-based designs, as exemplified by the Harris linked list [10, 11]. This confirms that stateless model checking, enhanced with domain-specific automaton reasoning, offers strong verification capabilities for SMR techniques. In doing so, we provide a definitive, positive response to **RQ1** and make our integration an essential backend for verifying correctness under various specifications and memory models.

5 Synthesis & Verification Pipeline

To generate and verify concurrent data structures, we developed a comprehensive pipeline that coordinates program synthesis, functionality testing, concurrency verification, and feedback-driven refinement. This process is depicted in Figure 1 (first introduced in Chapter 1). The process begins with *program synthesis*, in which an LLM produces candidate implementations from incomplete specifications. Each candidate is then subjected to two types of verification. First, *functionality testing* checks basic correctness properties by running a suite of unit tests. Second, *concurrency verification* analyses the candidate using a specialised model checker (GenMC), ensuring correctness in the face of weaker memory ordering behaviours and unsafe memory reclamations. If either step uncovers a deficiency, such as a failed unit test or an incorrect execution graph, the pipeline's *feedback integration* translates these diagnostic outcomes into guidance for the LLM, prompting a revised candidate that addresses the discovered errors. The iteration continues until no functionality or concurrency flaws remain, or until a configurable limit on refinement steps is reached.

5.1 Program Synthesis (RQ2, RQ3)

Program synthesis refers to the automated generation of source code that satisfies a given specification, often expressed as a combination of natural language descriptions, user-provided test cases, or formal constraints. This field has been characterised by symbolic and deductive methods designed to prune vast search spaces or to systematically construct programmes that are correct-by-construction [23, 24]. However, the recent breakthrough of LLMs trained on massive corpora of both natural language and code data has dramatically altered the landscape of program synthesis. Models like Codex and CodeGen have showcased striking capabilities in generating working solutions for relatively complex tasks, sometimes rivalling the performance of human programmers in certain controlled settings [6, 32]. The following paragraph provides an overview of key insights in the current literature on LLM-based program synthesis and is followed by our approach to selecting an optimal prompt design and locally hosted model combination for low-level C program synthesis.

LLMs for Program Synthesis: A Brief Survey Although neural models have been investigated for code generation and comprehension for several years, it was the advent of large-scale transformers trained on massive corpora that unlocked the potential of fully automated code-writing systems [33]. By framing code generation as a next-token prediction task, similar to standard language modelling, these systems learn both syntactic patterns (e.g., indentation, variable naming) and higher-level semantics (e.g., how to perform certain computations). Key contributors to this include:

- **Codex**, a GPT-based model further fine-tuned on a large Python code corpus, demonstrating the ability to generate functionally correct code for short prompts [6].
- **CodeGen**, trained on a blended dataset of natural language and code from various programming languages, achieving competitive performance on zero-shot Python tasks [32].
- **GPT-Neo**, open-source LLMs trained on The Pile, which includes a notable fraction of GitHub data, also show nontrivial code-writing capacities [34, 35].

Empirical evidence suggests that an LLM's capacity for program synthesis arises naturally from large-scale training, as the model is exposed to interleavings of natural language comments and code [34]. Consequently, even zero-shot prompts can produce coherent and often correct solutions for a variety of typical tasks, ranging from string manipulation to basic data structures [32].

Prompting Strategies and Effective Code Generation (RQ2) In this section, we investigate **RQ2** by discussing a range of prompting strategies that may help to improve zero-shot C code generation for concurrent data structures. It is recommended to take into account several design techniques since the correctness of the produced code can depend on the clarity and structure of the prompts. Although minimal prompts occasionally suffice, more structured techniques have emerged in the literature:

- **Few-shot prompting:** Including a small number of examples, such as input–output mappings or function signature snippets, within the prompt can help the model better grasp domain-specific conventions and expected formats [36]. In zero-shot scenarios, this method can give the model extra context.
- **High-Level Instruction-based prompts:** Providing explicit instructions regarding coding style or performance considerations (*e.g., "Implement a lock-free queue with hazard pointers"*) can help reduce the search space and minimise suboptimal outputs [37]. These instructions may not always produce correct code, but they guide the model toward appropriate implementation patterns.
- **Iterative refinement:** In this approach, code is generated in several stages and intermediate outputs are subjected to verification. If errors are detected, the prompt is adjusted and resubmitted, therefore simulating a back-and-forth dialogue until the final version reaches correctness [32]. This approach allows us to progressively fix mistakes that could arise in a single-shot generation even if it can cause more latency.
- **Fill-in-the-middle (FIM):** By marking specific segments in an existing code skeleton (e.g., using tokens like `<|fim_prefix|>`), the model is directed to focus on filling a particular *hole* in a larger code structure. Prior studies suggest that this can yield more accurate results for isolated code fragments compared to normal generation, though its effectiveness may vary across model sizes and domains [38].
- **System–User–Assistant roles:** Treating code generation as a multi-party conversation, where a `system` message defines high-level objectives (*e.g., "You are a concurrent programming assistant"*), a `user` message gives the problem statement, and an `assistant` responds with the generated code. Although this architecture can help to provide clearer direction and feedback, it may also need more prompt engineering to properly control role transitions [39].

In practice, combining more strategies may yield more consistent results than relying on a single technique. For instance, combining few-shot examples with high-level instructions and an iterative refinement process can help counter the limitations of each method. As outlined in Algorithm 3, our pipeline uses elements from all five prompting techniques to promote correct code generation. Later, Section 6.1 empirically supports that this combination of prompting strategies provides a reasonable baseline across different models and concurrent data structures, thereby also addressing RQ2: *What prompting techniques for LLMs should researchers use to generate functionally correct C code?*

Importance of Model Size and Training Regime (RQ3) We now investigate the primary factors known to affect zero-shot code generation quality in order to subsequently address **RQ3** in Section 6.1. A recurring theme in recent literature is that both *model size* and *training approach* have a strong influence on code-generation effectiveness, and larger parameter counts tend to yield better code quality [6]. For instance, smaller GPT-Neo variants trained on partial code corpora show substantially lower performance compared to multi-billion-parameter models such as CodeGen [34, 32].

In addition, whether a model is used in a zero-shot capacity or further refined on more specialised code data can cause significant changes in pass rates for functional correctness. Fine-tuning on domain-specific or tightly curated data aligns the model distribution to produce code that more closely matches a desired style or domain constraints. Empirical studies reflect:

- **Zero-shot models** (e.g., a code-trained LLM like CodeLlama [40]) can still generate workable solutions without additional fine-tuning, even for relatively challenging tasks. Nonetheless, more domain-heavy situations (like concurrency under weak memory models in C) can pose difficulties if the model's training data did not explicitly emphasise those patterns.
- **Specialised fine-tuning** (e.g., supervised or reinforcement learning–based [6]) often achieves higher correctness scores, primarily because it exposes the model to code distributions and correctness requirements that directly match the target usage.

In our context, we aim to evaluate a code-generation pipeline for generating new concurrent data structures. While specialised training might boost performance in concurrency scenarios, we opt **not to fine-tune** the model for this narrower objective. Our rationale is twofold: first, we plan to use well-known data structures as an evaluation set to assess the pipeline's capabilities without relying on memorised solutions from highly specialised data; and second, we envision a tool that generalises to novel data structures beyond those explicitly seen during training.

Additionally, we choose smaller, locally hosted models because our hardware constraints (32 GB of RAM) are insufficient for the largest-scale language models, and because we favour an offline approach over reliance on cloud-based APIs. In an early feasibility experiment, we observed that certain online services seemed to reference external data structures during inference, confounding any assessment of their intrinsic generation capabilities. By maintaining local deployments, we mitigate the influence of external resources, thus relying solely on a model's inherent aptitude for synthesising concurrency-related code. These decisions do, however, come with certain drawbacks, which are discussed in detail in the Limitations paragraph of Chapter 7.

Algorithm 3: Original Prompt for generating the `pop` function for the Treiber-stack implementation.

```
Function pop
    s : mystack_t*
<|fim_prefix|> pointer oldTop, newTop, next;
node_t *node;
bool success;
int val;
while true do
    <|fim_suffix|>
val = node->value;
// Reclaim the used slot
reclaim(get_ptr(oldTop));
return val;
<|fim_middle|>
// High-Level Instructions:
// - Implement a thread-safe pop operation for a lock-free stack.
// - Handle the edge case where the stack is empty.
// - Ensure atomic operations correctly update both the pointer and the counter to
    maintain consistency.
```

Experimental Setup

Step 1: Original Generate Prompt We begin by presenting the model with a partial C code snippet, as depicted in Algorithm 3, containing a clearly delimited gap, indicated using **Fill-in-the-middle (FIM)** tokens: `<|fim_prefix|>`, `<|fim_suffix|>`, and `<|fim_middle|>`. Alongside this code context, we provide concise **high-level instructions** outlining the required behaviour, while leaving flexibility for the model to determine the specific implementation details.

Notably, we intentionally avoid specifying detailed atomic memory orderings (e.g., acquire, relaxed, release) in the provided instructions. The reasoning behind this decision is that explicit memory concurrency primitives are typically familiar only to concurrency experts, whereas our goal is to make this synthesis tool accessible to a broader audience of programmers, including those without specialised expertise in concurrent memory semantics. By abstracting away these low-level concurrency details, we aim to leverage the model's implicit understanding, thereby facilitating usability without compromising correctness.

Step 2: Iterative Refinement Using Conversational Roles Should the initial generation fail to meet predefined structural checks, we proceed with an iterative, conversation-based refinement phase. In this stage, we leverage the **system–user–assistant** roles to enable structured, dynamic interactions and feedback. The process begins by taking a snapshot of the current conversation, which includes the initial prompt and the model's response. From there, we issue a new **system** message that employs **few-shot prompting** to provide clear guidance on the desired output format and structure.

Next, we invoke the conversational API using the accumulated conversation history. This history includes the few-shot system message, the most recent user message reiterating the code context and the specific gap to be filled, and the previous assistant message containing the candidate code snippet. Upon receiving a new candidate, we perform a series of automated checks. We extract the candidate code from the model's response, verify that it is enclosed within a proper `"c..."` code block, confirm that no placeholder symbols such as ??? remain, and ensure that the code conforms to the predefined output template through structural validation. This final check guarantees that no pre-existing code outside the intended gap has been modified.

If any of these checks fail, we automatically generate a new user message providing targeted feedback, such as requesting the removal of placeholders or adherence to the formatting template. This refinement loop continues until a valid code completion is produced or a predefined iteration limit is reached.

Evaluation Metrics To systematically evaluate different models, we make use of the previously described experimental setup to generate candidate solutions, adapting prompts only to include the specialised tokens required by each specific model. To ensure practicality and usability, we set a maximum acceptable threshold for conversational iterations at 10 iterations.

This threshold was chosen primarily based on efficiency and usability considerations. Allowing unlimited refinement iterations would significantly increase synthesis time and reduce the practicality of the tool in interactive or time-sensitive settings. In our experience, the 10-iteration cap represents a reasonable balance between resource usage, responsiveness, and the likelihood of success within a bounded search space. That said, the optimal iteration limit may vary depending on the complexity of the target program, the model's generalisation ability, and the prompt design. Further empirical investigation is needed to determine how these factors interact in different synthesis scenarios [41].

Consequently, only candidate solutions requiring fewer than or equal to 10 iterations to achieve structural correctness are considered for evaluation. For these solutions, we compute the Final BLEU Score using the SacreBLEU implementation, which quantifies lexical and structural similarity against a canonical reference implementation [42]. BLEU has become a widely adopted metric in the program synthesis and code generation literature, serving as one of the main methods for evaluating surface-level similarity and syntactic alignment [6, 43]. In our context, BLEU serves as the primary quantitative measure for comparing the performance of different models. A higher BLEU score indicates that the model not only produces compilable and syntactically valid code, but also closely follows idiomatic implementation patterns used in low-level concurrent C programs.

The BLEU score is calculated as follows:

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \tag{1}$$

where:

- p_n denotes the precision of n-grams up to order N,
- w_n is a typically uniform weight factor,
- $BP = \min(1, e^{(1-r/c)})$ is the brevity penalty,
- r and c represent reference and candidate lengths, respectively.

Therefore, to comprehensively assess model performance in our pipeline, we report **four metrics**:

1. **BLEU score**: Measuring lexical similarity to reference implementations.
2. **Compile pass rate**: The proportion of final outputs that compile successfully.
3. **Iterations to correctness**: The average number of refinement steps required to produce a valid output within the iteration limit.
4. **Verified solutions**: The number of candidates that pass concurrency verification.

These metrics, averaged over multiple runs, provide a well-rounded view of each model's capabilities in terms of lexical quality, syntactic validity, convergence efficiency, and correctness under weak memory.

5.2 Functionality Testing

Generative LLMs trained to translate high-level instructions or pseudocode into code can produce syntactically valid programs of substantial length yet still fail to execute the intended computation, especially when a significant intermediate state is involved. Recent work on synthesising programs from pseudocode demonstrates that coupling natural language guidance with input–output test cases, by searching over multiple candidates until one passes all tests, greatly improves functional correctness [44].

Motivated by these findings, we introduce a dedicated functionality testing stage in our pipeline: each structurally valid candidate is executed within an isolated Docker container against a predefined suite of unit tests. Any solution that fails is immediately returned to the LLM along with the captured error messages for automated refinement. This early filtering mechanism prevents fundamentally incorrect implementations from reaching the subsequent concurrency verification phase, thereby enhancing the efficiency of our synthesis workflow.

Technical Details Once a candidate solution passes all structural checks, the synthesised code is inserted into the data-structure template by filling the designated code hole. The complete source is then compiled and executed inside an isolated Docker container. We run a suite of unit tests that exercise the core operations of the data structure (e.g., push/pop for a stack) in a one-threaded environment. This step focuses exclusively on basic functional correctness. Any implementation that fails a unit test is considered a non-viable candidate and removed from further consideration.

Iterative Feedback and Context Management When a candidate solution fails one or more unit tests (for example, due to incorrect return values or runtime errors), the error diagnostics produced by the test are collected and sent back to the LLM. To streamline refinement, we clear almost all prior context, retaining only the latest structurally valid code and the specific error messages. This ensures the model's next attempt addresses only the current functional defects, rather than revisiting earlier generation details that are no longer relevant.

5.3 Concurrency Verification (RQ4)

To address **RQ4**, we integrate concurrency verification into our pipeline by invoking the enhanced GenMC described in Chapter 4. While unit tests (Section 5.2) validate single-threaded behaviour, covering only trivial execution scenarios, they cannot reveal problems arising from concurrent interleavings under weak-memory models.

Our pipeline performs a unified verification pass using the adapted GenMC backend. Each candidate implementation, together with its driver programme, is analysed directly by the stateless model checker. This process consists of:

- **Weak-memory interleaving checks**: We exhaustively explore all RC11-consistent executions and validate them against the data-structure specifications defined by Henkes, expressed in terms of `po` and `com` relations [25].
- **SMR hazard pointer checks**: Through our hazard-pointer automaton (Section 4.4), we detect memory reclamation issues such as *use-after-free.*

Whenever GenMC identifies a violation, whether a broken control-flow invariant (e.g., FIFO (first-in, first-out) / LIFO (last-in, first-out) ordering) or an SMR fault, it produces a minimal execution-graph counterexample. This diagnostic is then used as structured feedback to guide the next synthesis iteration.

Technical Details After passing unit tests, each candidate is submitted to GenMC for concurrency verification. To maintain consistency with the functionality-testing stage, we execute GenMC within the same Docker container used for unit testing. GenMC systematically explores every weak-memory interleaving of the candidate under RC11, checking each execution against the function-level specification. Simultaneously, SMR checks ensure that memory reclamation follows the hazard-pointer rules. Any detected violation produces a minimal counterexample trace, which we convert into structured feedback for prompt refinement.

Iterative Feedback and Context Management The feedback loop follows the same pattern as in the functionality-testing stage. Whenever GenMC emits a violation, we include its execution-graph counterexample output in the next prompt. We reset earlier context to retain only the latest valid C source code and the GenMC diagnostics. By focusing the LLM's attention on the specific concurrency or SMR issue, we streamline the refinement process.

Evaluation Metrics To assess how the inclusion of GenMC feedback alters the performance of our synthesis pipeline, we track two key measures: the **verified success rate**, defined as the proportion of runs that yield a GenMC-verified implementation within ten iterations, and **iterations to correctness**, which denotes the average number of generate–verify cycles required to produce a verified candidate. These metrics are shown and interpreted in Section 6.2.

We conduct a side-by-side comparison of two distinct pipeline configurations to demonstrate the advantages obtained by incorporating structured feedback. In the **baseline (no feedback)** configuration, each candidate is regenerated without using any GenMC counterexample information. In the **feedback-driven** configuration, GenMC counterexamples are injected into the LLM prompt, enabling targeted fixes based on precise verification results.

An increase in the verified success rate, alongside a possible reduction in the average number of iterations for the feedback-driven configuration, provides clear evidence that embedding GenMC's diagnostic information into the synthesis loop materially improves both convergence speed and overall correctness, thereby directly addressing **RQ4**.

5.4 Pipeline Evaluation

This section evaluates the end-to-end synthesis–verification loop and demonstrates how our approach integrates LLM-based generation with GenMC verification to produce correct concurrent data structures under weak memory. We begin by presenting absolute performance metrics for each benchmark and then compare against PSKETCH to quantify performance improvements.

We base our evaluation on an extended version of the GenMC benchmark suite originally proposed by Henkes [25]. That suite includes driver programmes for validating concurrent stacks, queues and mutexes. We have further extended it with additional driver programmes that exercise SMR operations, covering data structures not previously verified, such as the hazard-pointer-based Harris linked list [11]. All benchmarks are compiled with identical optimisation flags and executed in a controlled environment on a 32 GB, 6-core workstation with an i7-9750H processor with 12 virtual threads.

As in Chapter 4, each implementation is checked against Henkes's function-level data-structure specifications, defined in terms of the `po` and `com` relations. For full details on those specifications, the reader is referred to Henkes' thesis [25].

Absolute Results For each benchmark, we measure three key metrics over multiple independent runs: the **success rate**, defined as the proportion of synthesis runs that produce a fully GenMC-verified implementation within a ten-iteration limit; the **time to convergence**, which is the wall-clock time required to synthesise a verified solution, measured in seconds and averaged over the runs that succeed; and the **iterations to correctness**, representing the average number of generate–verify cycles needed before a candidate passes all GenMC checks. These absolute results provide a clear picture of how the pipeline behaves in isolation.

Comparison with Prior Work To contextualise our pipeline's performance, we compare it against the synthesis tools from the literature that operate under sequential consistency and encounter smaller solution spaces. Specifically, we contrast against **PSKETCH**, which uses counterexample-guided inductive synthesis (CEGIS) to complete partial programs [23].

For the comparison, we match our benchmark complexity (e.g., while loop within a dequq function) to templates of comparable complexity as described in the original papers. This enables a fair assessment of how well our feedback-guided, weak-memory-capable pipeline performs relative to solver-based, SC-oriented baselines.

6 Results

This chapter presents the empirical evaluation of the synthesis–verification pipeline described in Chapter 5. Its purpose is to quantify the pipeline's ability to generate low-level concurrent C data structures that are both functionally correct and free from weak-memory and SMR violations. All experiments are conducted on benchmark programs drawn from the Henkes suite, extended with additional SMR-dependent implementations [25]. We employ the evaluation metrics earlier introduced, namely *BLEU* for assessing structural similarity, *iterations-to-correctness* and *time* for measuring refinement effort, and *success rate* to capture overall convergence.

The chapter is structured around the remaining research questions. **RQ1** is addressed via the GenMC extensions in Chapter 4, and **RQ2** through the prompt design analysis in Section 5.1. Section 6.1 answers **RQ3** by evaluating how effectively LLMs can synthesise correct concurrent implementations in a zero-shot setting, guiding our model selection. Section 6.2 investigates **RQ4**, measuring the effect of structured verifier feedback on convergence. Finally, Section 6.3 provides an end-to-end evaluation of the complete synthesis–verification loop to answer the overarching research question.

6.1 LLM Synthesis Accuracy (RQ3)

This section answers **RQ3**: *How effectively can LLMs synthesise low-level concurrent C code in a zero-shot setting?* Our goal is to assess LLMs' suitability for integration into a synthesis-verification pipeline and to identify a strong default model. Since concurrent data structures require precise handling of atomic operations and memory orderings, understanding an LLM's raw ability to produce syntactically correct C code without any task-specific training is critical.

We evaluate three publicly available LLMs using a shared natural language prompt and identical decoding settings with **Ollama**. Each model is tasked with synthesising candidate implementations for one benchmark instance in the Henkes suite, the Treiber-stack, without any task-specific examples or fine-tuning [15]. The generation process includes a lightweight feedback loop in which outputs are checked for structural template conformance and compilation validity; candidates failing these checks are rejected, and feedback continues to be provided until a valid candidate is found or a maximum of **10** iterations per candidate is reached. A detailed description of this module's setup can be found in Section 5.1.

We measure performance across four dimensions: (i) **BLEU score**, quantifying lexical similarity to a canonical implementation; (ii) **Compile pass rate**, the proportion of final outputs that compile successfully; (iii) **Iterations to correctness**, the average number of attempts required to generate a compilable and structurally valid output; and (iv) **Verified solutions**, the number of candidates that pass formal verification using GENMC.

Model	**BLEU**	**Compile Pass Rate**	**Iterations**	**Verified Solutions**
qwen2.5-coder:7b	84.44	100%	3.5	2
codellama:7b-instruct	78.94	60%	5.3	0
deepseek-coder-v2	78.59	10%	8.0	0

Table 3: Zero-shot synthesis performance on the Treiber-stack benchmark. For each model, we report the average BLEU score, compile pass rate, average iterations to generate a compilable and structurally valid output, and the number of candidates that passed GENMC verification, computed over ten runs.

Summary LLMs exhibit varying levels of effectiveness when synthesising low-level concurrent C code in a zero-shot setting, directly addressing **RQ3**. While all models can produce structurally plausible code, only Qwen2.5-Coder consistently generates compilable candidates and produces verified solutions. These results also support our findings for **RQ2** regarding prompt design, demonstrating that even without fine-tuning, careful prompting enables reliable synthesis. Based on its superior compile reliability, lexical similarity, and verification success, we choose Qwen2.5-Coder as the synthesis component in our pipeline [45].

6.2 Verifier Feedback Impact (RQ4)

This section answers **RQ4**: *How does incorporating structured verifier feedback into LLM prompts affect convergence to correct concurrent data structure implementations?* Our aim is to determine whether embedding GenMC counterexample information into each prompt leads to faster convergence and higher final correctness in the synthesis–verification loop.

We compare two pipeline configurations, both of which perform structural and functional checks followed by GenMC verification at each iteration. In the *baseline (no feedback)* configuration, when a candidate fails concurrency verification, the next candidate is generated without any information from the counterexample. In the *feedback-driven* configuration, the minimal execution-graph counterexample produced by GenMC (as described in Chapter 4) is injected into the LLM prompt before generating the next candidate. This added diagnostic context helps the model understand exactly which interleaving or SMR rule was violated, guiding it to correct the specific conflict rather than just retrying. By focusing each revision on concrete errors identified by GenMC, we aim to escape faulty implementations.

Each configuration is tested on the Treiber-stack benchmark using the same fixed prompt, identical decoding parameters, and an unchanged runtime environment. We perform ten independent runs for each setup, measuring both the proportion of runs in which a fully verified solution is found within ten iterations (verified success rate) and, for those runs that do succeed, the average number of generate–verify cycles required to reach a GenMC-verified implementation (iterations to correctness).

Configuration	**Verified Success Rate**	**Iterations to Correctness**
Baseline (no feedback)	30%	2.3
With GenMC Feedback	70%	3.9

Table 4: Effect of GenMC counterexample feedback on the Treiber-stack synthesis process. Verified success rate indicates the percentage of runs that produced a verified implementation within ten iterations, while iterations to correctness shows the average number of generate–verify cycles among successful runs.

Summary Incorporating structured verifier feedback into LLM prompts substantially improves the model's ability to converge to verified concurrent implementations. The feedback-enabled configuration achieves a **70%** success rate within 10 iterations, more than **double** the baseline's **30%**. Although the average iteration count is higher (3.9 vs. 2.3), this increase reflects the model's ability to make progress on harder problems, specifically, resolving concurrency and functional correctness issues that arise after compilation succeeds. In contrast, the baseline's lower iteration count is often due to two distinct behaviours: either it gets *lucky* and generates a correct implementation immediately after passing compilation checks, or it becomes stuck, repeatedly generating variants that fail verification without meaningful improvement.

In a follow-up **stress test** where both configurations generated a solution containing a concurrency error and allowed unlimited iterations, the **baseline failed to converge after 100+ attempts and 900+ seconds**, while the **feedback-enabled** version synthesised a correct and verified solution in just **6 iterations** and **54.97 seconds**. These findings confirm that structured GenMC feedback not only increases convergence success but also enables the model to escape faulty implementations that static prompting alone cannot overcome, providing an affirmative answer to **RQ4**.

6.3 End-to-End Synthesis Performance

This section provides a comprehensive evaluation of the full synthesis-verification loop and answers the main research question: *How can Large Language Models and advanced verification tools integrate effectively to synthesise concurrent data structure implementations targeting weak memory models?* We evaluate our tool's efficiency and correctness in synthesising full implementations, compare its performance to the state of the art, and report results across a diverse set of data structures and memory models.

Algorithm 4: The Dequeue Implementation synthesised against PSKETCH

```
Function dequeue
    q : queue_t*, retVal : unsigned int*
int success = 0;
pointer head, tail, next;
while !success do
    head = atomic_load_explicit(&q->head, acquire);
    tail = atomic_load_explicit(&q->tail, relaxed);
    next = atomic_load_explicit(&q->nodes[get_ptr(head)].next, acquire);
    if atomic_load_explicit(&q->head, relaxed) == head then
        if get_ptr(head) == get_ptr(tail) then
            assert(get_ptr(next) != POISON_IDX);
            if get_ptr(next) == 0 then
                return false;
            atomic_compare_exchange_strong_explicit(

                &q->tail, &tail,

                MAKE_POINTER(get_ptr(next), get_count(tail) + 1),

                release, release);
        else
            *retVal = q->nodes[get_ptr(next)].value;
            success = atomic_compare_exchange_strong_explicit(

                &q->head, &head,

                MAKE_POINTER(get_ptr(next), get_count(head) + 1),

                release, release);
            __VERIFIER_assume(success);
reclaim(get_ptr(head));
return true;
```

Comparison with Prior Work To contextualise our tool's performance, we compare it with **PSKETCH**, a solver-based synthesis tool for concurrent data structures. In their evaluation, synthesising a `deque` function with a non-trivial `while` loop under sequential consistency took **147.07 seconds** [23]. In contrast, our tool synthesised a more complex `deque` variant (shown in Algorithm 4), which includes weaker memory accesses in an average of **39.46 seconds** over 10 attempts. Despite differences in the experimental setup, this indicates a significant speedup over the state of the art, even on more challenging inputs, and highlights the effectiveness of combining LLM synthesis with verifier-guided feedback. Furthermore, because our benchmark includes realistic weak-memory and SMR verification requirements, this comparison underscores the practical advantages of our pipeline in handling real-world concurrency complexities.

Absolute Results To assess the general effectiveness of our pipeline, we evaluate its ability to synthesise complete function implementations across a range of concurrent data structure implementations. Each benchmark is tested under multiple data structure models, as defined by Henkes and further extended with SMR where applicable [25]. For each task, we record the verification success rate over a maximum of ten iterations, the total synthesis time, and the average number of iterations required for convergence. These metrics collectively illustrate how quickly and reliably the pipeline produces verified implementations when operating in isolation.

Implementation	**Model**	**Success Rate**	**Time (s)**	**Iterations**
mutex	mutex	100%	12.12	5.5
alt-mutex	mutex	100%	13.52	4.5
spinlock	mutex	60%	7.73	3.8
ttaslock	mutex	90%	4.09	2.0
ttaslock-opt	mutex	60%	6.82	2.5
ticketlock	mutex	100%	8.51	3.3
twalock	mutex	100%	7.44	2.5

Table 5: End-to-end synthesis results for the `lock` (or `acquire`) function across various mutex implementations. For each implementation, we report the underlying data structure model, the percentage of runs that produced a verified solution within ten iterations, the average wall-clock time to convergence (in seconds), and the mean number of generate–verify iterations

Mutex Results A mutex (mutual exclusion object) ensures that only one thread can hold the lock at a time, preventing concurrent threads from entering critical sections simultaneously. The simplest version, labelled *mutex*, uses a basic lock/unlock protocol; *alt-mutex* is a variant with minor optimisations in acquisition order. A *spinlock* busy-waits until the lock becomes available, while a test-and-test-and-set lock (*ttaslock*) first checks a flag before attempting an atomic test-and-set, reducing memory traffic; *ttaslock-opt* further minimises contention by optimising cache usage. A *ticketlock* grants access in FIFO order using ticket counters, ensuring fairness, and *twalock* (test-and-wait-and-acquire) uses a two-phase check to reduce spinning overhead.

Table 5 shows that all variants achieve rapid synthesis: *mutex*, *alt-mutex*, *ticketlock* and *twalock* each reach a **100%** success rate, converging in around 7 to 14 seconds. *Spinlock* and *ttaslock-opt*, despite their simplicity, still achieve **60%** percent success within very short runtimes (under 8 seconds), demonstrating that a wide range of locking schemes can be synthesised efficiently by our pipeline. Synthesising each variation highlights the generality of our tool: from basic lock/unlock idioms to more sophisticated fairness and contention-reduction strategies, all can be handled within our tool without any additional modifications to the verification logic.

Implementation	**Model**	**Success Rate**	**Time (s)**	**Iterations**
ms-queue	weak-q	80%	39.46	3.6
ms-queue-d	weak-q & smr	50%	42.24	2.6
ms-queue-d	hw-q & smr	70%	18.25	2.3
qu	weak-q	60%	7.50	1.7
qu	hw-q	60%	10.52	2.7
qu-opt	hw-q	50%	8.54	2.0

Table 6: End-to-end synthesis results for the `deque` function across various queue implementations and data structure models. For each implementation and model pair, we report the success rate (percentage of runs that produced a GenMC-verified solution within ten iterations), the average wall-clock time to convergence (in seconds), and the mean number of generate–verify iterations, all averaged over multiple independent runs.

Queue Results A concurrent queue allows multiple threads to enqueue and dequeue items safely without locks, using atomic operations and specific memory-order guarantees. The *ms-queue* implementation refers to the Michael–Scott non-blocking queue, which uses two atomic pointers (head and tail) and requires careful ordering to maintain FIFO behaviour under weak memory [10]. The *ms-queue-d* variants extend this design with Safe SMR support: under the *weak-q* model, the SMR automaton enforces hazard-pointer rules, while under *hw-q* (hardware-queue) the model reflects memory guarantees present on actual hardware. The *qu* implementation is a simpler queue that uses weaker atomic primitives, and *qu-opt* is an optimized version that further reduces atomic overhead through caching and fewer memory barriers.

Table 6 shows good performance, though with somewhat reduced success rates and longer runtimes, reflecting the increased complexity of the **deque** function compared to the mutex **lock or acquire** functions. The ms-queue achieved a notable success rate of **80%**, while the integration of Safe Memory Reclamation (SMR) lowered the success rates slightly, as seen with ms-queue-d (**50%** to **70%**). The qu and qu-opt implementations also demonstrated consistent performance. Despite these slightly lower success rates and longer synthesis times compared to the mutex implementations, the results remain impressive, confirming the tool's capability to handle more complex concurrent data structures effectively.

Implementation	**Model**	**Success Rate**	**Time (s)**	**Iterations**
dq	weak-s	100%	25.21	3.2
dq-opt	weak-s	100%	17.66	2.4
stc	weak-s	50%	20.50	2.4
stc-opt	weak-s	40%	29.29	4.0
treiber-stack	weak-s	70%	18.32	3.9
treiber-stack-d	weak-s & smr	50%	20.77	4.4
treiber-stack-d	c-s & smr	40%	20.30	4.0

Table 7: End-to-end synthesis results for the `pop` function across various stack implementations and data structure models. For each implementation and model pair, we report the success rate (percentage of runs yielding a GenMC-verified solution within ten iterations), the average wall-clock time to convergence (in seconds), and the mean number of generate–verify iterations, all averaged over multiple independent runs.

Stack Results A concurrent stack allows multiple threads to push and pop items in a last-in, first-out (LIFO) order without locks, using atomic operations to maintain consistency under weak memory. The *dq* implementation represents a basic deque-based stack, while *dq-opt* is an optimized version that reduces atomic overhead through caching and fewer memory barriers. The *stc* implementation employs a split-cache design, separating push and pop pointers to reduce contention; *stc-opt* further refines this approach by optimising memory barriers and pointer updates. The *treiber-stack* is the classic Treiber lock-free stack, which relies on an atomic compare-and-swap on the head pointer; *treiber-stack-d* extends this design with Safe Memory Reclamation (SMR) to safely reclaim nodes. Under the *weak-s* model, hazard pointers are enforced solely by the SMR automaton, whereas under the *c-s* model stronger memory orders of real hardware are assumed alongside SMR.

Table 7 reinforces the positive pattern of results, with strong success rates and efficient convergence for the dq (**100%**) and dq-opt (**100%**) implementations. Notably, for the dq implementation, previously identified by Henkes as containing a concurrency error, we successfully synthesised a version incorporating operations with stronger concurrency rules that passes the model checking stage of our pipeline [25]. Although this does not necessarily imply an improvement of the data structure itself, it showcases the tool's versatility and potential for future work. Other stack implementations presented lower success rates but still demonstrate solid performance considering their complexity and the use of SMR.

Summary We successfully synthesised **20** implementation and model combinations, demonstrating fast convergence and strong success rates in generating correct concurrent data structure implementations under weak memory models. Compared to prior solver-based approaches, specifically PSKETCH, our synthesis-verification loop significantly outperforms it, achieving faster convergence even on more complex structures. These findings **affirmatively answer** the primary research question: LLMs, when guided by structured feedback from advanced verification tools like GENMC, can be effectively integrated into a feedback-driven synthesis loop to efficiently generate verified concurrent data structure implementations for weak memory models.

7 Conclusion

This work addressed the significant challenge of synthesising correct concurrent data structures under weak memory models, an area traditionally affected by subtle concurrency errors and by the limitations of exhaustive search-based approaches. Recognising that concurrent data structures greatly benefit in performance from using **SMR** techniques, we developed our synthesis and verification tool with a forward-looking perspective to support future data structures with SMR integrated. A significant innovation was our extension of GenMC to incorporate automaton-based verification of **Hazard Pointers**. Consequently, our primary contribution is integrating **LLMs** into an advanced synthesis and verification pipeline, specifically tailored for concurrent data structures operating under **weak memory semantics**. By combining the generative capabilities of LLMs with the specialised verification strengths of an enhanced **GenMC**, we established an iterative refinement loop capable of efficiently synthesising verified low-level concurrent C code. Empirical evaluation demonstrated substantial improvements in synthesis speed and the ability to generate a large range of well-known data structure implementations compared to existing state-of-the-art tools such as **PSKETCH**, highlighting the scalability and practicality of our approach for automated concurrent data structure design.

Limitations Nevertheless, our approach is not without limitations. First, the synthesis quality is connected to the limitations, tendencies, and **training artefacts** of the underlying LLM. Since models like Qwen2.5-Coder are trained on general-purpose code corpora, they may lack sufficient exposure to specialised patterns in concurrent C programming, particularly those involving subtle memory orderings and synchronisation primitives. This can result in low-quality generations or the inability to synthesise novel solutions when the model's training data does not reflect the desired concurrency behaviour. Second, there exists a delicate balance in **prompt engineering**: excessive specificity can overly constrain the model, limiting creativity, while insufficient guidance may lead to incorrect outputs. Moreover, the success of our iterative refinement loop is sensitive to the feedback integration from GenMC, where ineffective prompt injection can lead to stagnation in the refinement process. Third, the **computational constraints** of our setup led us to rely on smaller locally hosted models. This choice helped mitigate the risk of **data leakage** from benchmark implementations the model may have seen during pre-training, but it also restricted our ability to explore the synthesis capabilities of larger models. Finally, while our pipeline successfully synthesised a variety of data structure implementations, its ability to generalise to complex novel data structures remains an open question.

Future Work Our work can be extended by synthesising novel data structure implementations, while also exploring its application in automatically repairing weak-memory bugs in existing concurrent code. This transition from **generation to repair** would validate the pipeline's utility and flexibility. Additionally, experimenting with larger LLMs and specialised fine-tuned variants could unlock more sophisticated concurrency patterns and, if used as **judges**, replace the formal verifier to accelerate the feedback loop. Another important direction is refining how GenMC feedback is injected into the prompts: exploring semantically enriched representations of **execution graph** counterexamples may help LLMs identify and resolve concurrency violations more effectively. Expanding the verification backend to support additional memory reclamation techniques, such as Epoch-Based Reclamation (**EBR**), could further extend the pipeline's applicability to a broader class of concurrent systems. Looking ahead, these directions collectively represent a compelling roadmap for advancing synthesis research in the context of weak memory concurrency, where traditional tools struggle with scalability. By pushing the boundaries of automated generation, our approach opens a path toward more efficient and accessible design of concurrent data structures under realistic memory models.

References

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[2] 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.

[3] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings 22*, pages 391–407. Springer, 2009.

[4] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying arm concurrency: multicopy-atomic axiomatic and operational models for armv8. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, 2017.

[5] Maged M Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 21–30, 2002.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[7] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Ryan Tsang, Najmeh Nazari, Han Wang, Houman Homayoun, et al. Large language models for code analysis: Do {LLMs} really do their job? In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 829–846, 2024.

[8] Michalis Kokologiannakis and Viktor Vafeiadis. Genmc: A model checker for weak memory models. In *International Conference on Computer Aided Verification*, pages 427–440. Springer, 2021.

[9] Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

[10] Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82, 2002.

[11] Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*, pages 300–314. Springer, 2001.

[12] Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers*, 100(9):690–691, 1979.

[13] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in c/c++ 11. *ACM SIGPLAN Notices*, 52(6):618–632, 2017.

[14] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

[15] R Kent Treiber et al. *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research . . . , 1986.

[16] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[17] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. In *2010 13th IEEE international symposium on object/component/service-oriented real-time distributed computing*, pages 185–192. IEEE, 2010.

[18] Sebastian Wolff. *Verifying Non-blocking Data Structures with Manual Memory Management.* PhD thesis, Dissertation, Braunschweig, Technische Universität Braunschweig, 2021, 2021.

[19] Trevor Alexander Brown. Reclaiming memory for lock-free data structures: There has to be a better way. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 261–270, 2015.

[20] Nachshon Cohen. Every data structure deserves lock-free memory reclamation. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–24, 2018.

[21] Ruslan Nikolaev and Binoy Ravindran. Universal wait-free memory reclamation. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 130–143, 2020.

[22] MD Arovi and Ruslan Nikolaev. Fixing non-blocking data structures for better compatibility with memory reclamation schemes. *arXiv preprint arXiv:2504.06254*, 2025.

[23] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. Sketching concurrent data structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 136–148, 2008.

[24] Sarat Chandra Varanasi, Neeraj Mittal, and Gopal Gupta. Generating concurrent programs from sequential data structure knowledge using answer set programming. *arXiv preprint arXiv:2109.08298*, 2021.

[25] C. L. W. Henkes. Verifying weak memory concurrent data structure implementations. Master's thesis, Delft University of Technology, Delft, The Netherlands, 2024.

[26] Weiyu Luo and Brian Demsky. C11tester: a race detector for c/c++ atomics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 630–646, 2021.

[27] Koushik Sen. Effective random testing of concurrent programs. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 323–332, 2007.

[28] 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.

[29] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. *ACM SIGPLAN Notices*, 49(1):373–384, 2014.

[30] Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. *ACM SIGPLAN Notices*, 50(6):165–174, 2015.

[31] Maged M Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[32] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

[33] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[34] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

[35] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

[36] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55:1 – 35, 2021.

[37] Laria Reynolds and Kyle McDonell. Prompt programming for large language models: Beyond the few-shot paradigm. In *Extended abstracts of the 2021 CHI conference on human factors in computing systems*, pages 1–7, 2021.

[38] Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022.

[39] Aobo Kong, Shiwan Zhao, Hao Chen, Qicheng Li, Yong Qin, Ruiqi Sun, Xin Zhou, Enzhi Wang, and Xiaohang Dong. Better zero-shot reasoning with role-play prompting. *arXiv preprint arXiv:2308.07702*, 2023.

[40] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[41] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*, 2023.

[42] Matt Post. A call for clarity in reporting bleu scores. *arXiv preprint arXiv:1804.08771*, 2018.

[43] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[44] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32, 2019.

[45] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.