Identifying Design Flaws in a Lock-Free Task Pool with TLA+

Vasil Dyadov, Alexander Kogtenkov[0000−0003−4873−8306], and Ilya Shchepetkov[0009−0008−5630−1896]

Kaspersky Lab, Moscow, Russia
vasil.s.d@gmail.com, kwaxer@mail.ru, ilya.shchepetkov@17451k.space

Abstract. Lock-free data structures are notoriously difficult to design and implement correctly. The absence of critical sections protected by synchronization primitives makes reasoning much harder, often leading to almost unreproducible bugs and subtle errors. All of these can be systematically addressed using formal methods. We use TLA+, a state-based formal method especially suitable for working with concurrency, to verify SALSA, a scalable and low synchronization NUMA-aware algorithm for producer-consumer pools. The task pool was considered as part of a redesign phase of inter-process communication in a microkernel-based operating system under development. During this work, we uncovered numerous issues with the algorithm, from small typos in the pseudo-code to significant design flaws. The work was completed in two person-months and resulted in a decision to abandon this task pool in favor of a more traditional lock-free queue. This potentially saved us many more months of development and debugging and advanced formal verification use within the company.

Keywords: formal verification · lock-free queue · concurrency · TLA+.

1 Introduction

High-assurance systems achieve their goals with appropriate design and technology principles. In our case of a secure operating system KasperskyOS, the microkernel design reduces the attack surface compared to monolithic kernels, and lock-free structures for the central mechanism, inter-process communication (IPC), guarantee desired performance. Among the research questions are the pros and cons of different algorithms for message queues, including SALSA [12], a scalable and low synchronization NUMA-aware algorithm for producer-consumer pools.

SALSA is a lock-free algorithm that relies on hardware atomic instructions to keep its operations thread safe. Compared to more traditional lock-based algorithms with critical sections, lock-free ones guarantee progress (no deadlocks and stalling) and enable higher performance in high-contention scenarios [8,3]. However, they are significantly harder to design and reason about [11,4], which often leads to subtle bugs. As a preventive measure to avoid these issues, we employed formal methods.

This paper presents the process and results of a formal verification of SALSA with TLA+ [20]. Contributions of our work include the following:

- A detailed TLA+ specification that accurately captures the concurrent behavior of the SALSA task pool.
- A brief review of the risks associated with formalization and their possible mitigation.
- A collection of invariants that should be preserved at any time during the algorithm execution.
- Demonstration of the suitability of the TLA+ formalism for verification of lock-free queues in the industrial setting.
- Discovery of significant flaws in the algorithm that were not previously publicly described.

The paper is organized as follows: the reasoning behind choosing TLA+ (section 2), an overview of SALSA (section 3), the formal specification (section 4), analysis of the results and the discovered issues (section 5), discussion of the findings (section 6), the related work (section 7), and the conclusion (section 8).

2 Choosing TLA+

The following requirements influenced the choice of the verification tool: the verification project should remain practical and achieve results in a time frame of no more than two months; the produced artifacts should be understandable by developers without a steep learning curve. The latter one facilitates the ownership transfer of the produced specification and its future maintenance and improvement by the developer team.

To reduce risk, we excluded methods that we had no previous experience with. This left us with the Isabelle/HOL proof assistant [24], Alloy [17], Event-B [1], and TLA+ [20]. Isabelle/HOL contains layers of specialized logic and tooling that differ significantly from standard software engineering workflows, making it very hard to introduce developers into. Alloy is a great choice for structural modeling and has a sufficiently simple language, but it struggles with very long execution traces that arise due to possible interleavings within concurrent algorithms. The Event-B language is not tailored towards the specification of concurrent executions expressed as a sequential algorithm, though it can be done using control variables. This would be unintuitive for the developers; and the process of proving the correctness of the model in this case would probably require the use of refinement [2,1], which is yet another potential barrier.

TLA+ is a high-level formal specification language designed to model and to verify complex systems, especially distributed and concurrent ones [21]. Using it, a verification engineer can model threads, simulate all possible interleavings between them, quickly iterate, and debug issues by using the available tools and inspecting reported reports or error traces. On top of it, the PlusCal language [22], which can be automatically transpiled into raw TLA+, greatly simplifies the

specification of concurrent executions and is more easily understandable by developers, because it looks like a pseudo-code.

There are different ways to find errors in TLA+ specifications or to prove their absence. The primary tool is the TLC model checker [29]. It can exhaustively explore all reachable states of a specification to find a state where invariants or temporal properties are violated. If such is found, TLC outputs an error trace that shows a shortest path from an initial state to the error one. Other tools include Apalache [19], a bounded symbolic model checker, capable of exploring infinite state space but usually no longer than 6–12 steps from an initial state. And finally, TLAPS [9] is used to write and to mechanically verify formal proofs of a system's correctness.

PlusCal is best used for specifying shared-memory concurrent algorithms and is reportedly more approachable by engineers, since it feels closer to a traditional programming language rather than to a sophisticated math [23]. The language introduces labels to define an atomic "step" of an algorithm:

- Everything between two labels is executed as a single atomic action.
- Context switches (interleaving) can occur only at labels.
- TLC checks invariants and explores different states only at labels.

3 Overview of SALSA

SALSA is a non-blocking **producer-consumer** task pool that offers high throughput and supports the writing of policies to achieve load balancing and cache locality. Load balancing is further improved through **stealing**, which allows one consumer to take **tasks** that were originally assigned to another. To make this process efficient, tasks are grouped into larger units of fixed size called **chunks**. Stealing an entire chunk significantly reduces overhead compared to stealing individual tasks one by one.

To support stealing, chunks store information about their **owner**, who is one of the consumers. Each chunk may be encapsulated within a **node** containing an integer index field (**idx**); its purpose is to track the last taken task in that chunk. With the index, the owner can claim tasks sequentially without interfering with other consumers. Stealing a chunk involves creating a new encapsulating node with separate index, which protects from possible reading collisions between the original and new owners.

Initially, a chunk does not contain any tasks. Each producer chooses a consumer to work with, chooses a chunk owned by that consumer, adds new tasks to it until it is full, and then repeats the process. Consumers retrieve tasks and replace the corresponding chunk entries with the **TAKEN** entry.

Each consumer has its own task pool represented by several **chunk lists** and a single **chunk pool** (see Fig. 1). Contrary to the name, each chunk list is a linked list of nodes, which may or may not point to chunks. Consumers can freely retrieve tasks from all their local chunk lists.

When working with a particular consumer task pool, each producer can insert tasks only to a single chunk list fully reserved for its use. This eliminates synchronization between multiple producers.

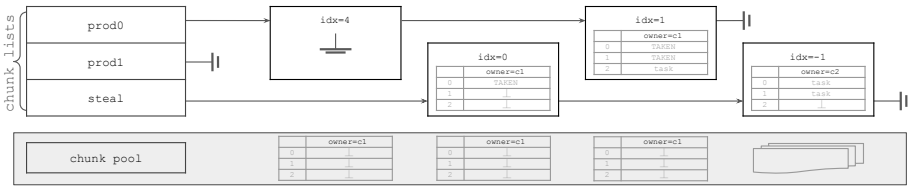

Fig. 1. Architecture of a representative task pool for an individual consumer. Each consumer maintains a dedicated chunk list for every producer in the system, an additional list for stealing tasks, and a pool of free chunks. Each entry in the chunk lists includes an **owner** field to identify the consumer that currently has the right to take tasks from this chunk.

Chunk pools are collections of spare chunks that facilitate efficient memory usage. Fully consumed chunks are stored in the chunk pool by a consumer and are retrieved by a producer when a new chunk is required for new tasks.

The stealing process is the most complicated part of the whole algorithm, as it tries to strike a balance between correctness of synchronization and efficiency by design.

4 Specification in TLA+

Working with TLA+ includes the following logical steps:

1. Formalization of the algorithm.
2. Definition of the properties to be checked.
3. Configuration and evaluation by the tool.
4. Analysis of the results; implementation of any changes if necessary.

This section describes the first two steps, and the next section discusses the evaluation and results.

We chose PlusCal to verify SALSA and specified the algorithm as closely as possible, describing the slight deviations and the reasoning behind them in the comments. The translation from informal pseudo-code to PlusCal was fairly straightforward. The difficult part was to set the correct atomicity zones using labels (see the example in Fig. 2) and come up with invariants. Too granular placement of the labels can lead to redundant interleaving checks: for example, several writings to local process variables can be safely grouped together under common labels, since they can not interfere with the execution of other processes.

We check the following categories of properties in lock-free algorithms:

Pseudocode

```
1: Function checkLast(Node n):
2:   if(n.idx + 1 = CHUNK_SIZE) then
3:     n.c ← ⊥;
4:     return chunk to chunkPool;
5:     currentNode ← ⊥;
```

Simplified PlusCal

```
 1: procedure checkLast(n)
 2: variable chunk = NULL;
 3: if nodes[n].idx + 1 = CHUNK_SIZE then
 4:   chunk := nodes[n].c;
 5:   nodes[n].c := NULL;
 6:   if chunk /= NULL then
 7:     scPools[self].chunkPool := Append(
 8:       scPools[self].chunkPool, chunk);
 9:   end if;
10:   scPools[self].currentNodeId := NULL;
11: end if;
```

Fig. 2. Translation example for the `checkLast()` procedure from algorithmic pseudocode to PlusCal. The procedure is executed by a consumer; all variables, except for the procedure argument `Node n`, are global. Colors denote atomicity zones defined via PlusCal labels (absent in the example). Some syntax details are omitted for clarity.

- *Linearizability*, a strong correctness condition that helps to reason about the system [15].
- *Lock-freedom* confirms that there is always a thread that makes progress.
- *Absence of ABA problems* or correctness of proposed mitigations.
- *Functional correctness* guarantees that the algorithm never produces a wrong result or enters an invalid state.
- *Structural correctness* covers invariants of the underlying data structures, including type correctness.

Orthogonally to the above, we add an additional layer—*memory models*—to take into account specific hardware behavior with memory access. There is a recent report on the formalization of a weak ARM memory model in TLA+ [28]. Internally, we employ ARM and x86-TSO memory models with TLA+ specifications.

Shortly after starting adding invariants and checking the specification of SALSA with TLC, we started receiving issue reports with error traces. After analysis, we found that some of the initial invariants turned out to be too strict and the reports were actually false positives, so we removed them from the specification. However, we categorized other reports as true positives, representing critical issues (see section 5) in the SALSA algorithm. We had been attempting to fix them and ultimately failed; we believe that it may be impossible to do so without fundamentally affecting the reported performance characteristics. Also, having failed to prove existing invariants, we stopped adding new ones. In the end, our specification contains only invariants that describe functional and structural correctness, as well as the absence of runtime errors.

The TLA+ specification of SALSA is available online under MIT license [1]. It includes 850 lines of invariants and PlusCal code; the PlusCal part translates into about 1,250 lines of TLA+.

[1] https://codeberg.org/17451k/salsa-spec

5 Evaluation of the model and discovered issues

Model checkers usually report one error at a time because they halt upon encountering an inconsistency. Stopping early prevents a cascade of misleading, secondary errors caused by the initial violation, reducing false positives. It makes it difficult to find several issues without first fixing the existing ones, but we did our best to manually check the error traces and to confirm that the occurrence of each reported error does not depend on another.

With the above in mind, we found the following problems in the published version of the algorithm:

1. **Task loss, use-after-free.** The algorithm does not explain in detail what happens to a chunk when it is returned to a chunk pool. We assume that it is re-initialized, since one of the purposes of its existence is memory reuse, and other possible interpretations lead to even more significant errors. However, verification shows that re-initialization leads to a broken state in several places and, most importantly, task loss. Example of the latter:
   - The system runs 2 concurrent consumers, `C1` and `C2`.
   - `C1` initiates `takeTask(Node n)` function and increases the index `n.idx++` to tell the world that it is going to take a task from `n`.
   - `C2` steals the chunk and finishes taking the rest of the tasks from it. `C2` returns an empty chunk to a chunk pool, where it is reinitialized.
   - Except that it was not empty: consumption of one task by `C1` was not finished. This results in the use-after-free error down the line.
2. **Index out of bounds.** This happens when a `Consumer` takes a task from a chunk that it just stole:
   - Assuming that the stolen chunk is full except for one last task, this last task will be successfully consumed at the end of the `steal()` function (lines `106-111` of the algorithm [12]).
   - The algorithm here has a mistake: the `checkLast()` call occurs before the node index increases, and thus the call fails to detect that the chunk is now fully consumed.
   - This leads to a runtime error in `takeTask()`, when the `Consumer` tries to access a task at index `n.idx+1`, which just became out of bounds.
3. **Unexpected value.** When there are no available tasks anywhere and only a single chunk is present, two consumers may end up stealing this chunk from each other continuously in a loop. During this process, one consumer may try to insert the node into its `STEAL` chunk list that already contains the copy of this node, causing duplicates. It is unlikely to cause any safety issues, but leads to excess memory or CPU usage required to handle this behavior.
4. **Invalid state.** It is possible to execute the "take last task" part of the algorithm twice. This leads to two consumers having the same chunk in their chunk pools, which in turn results in producers adding tasks to the same chunk concurrently without any synchronization.
   - Assume that there is an almost fully consumed chunk, and only two tasks are available.

- `C2` starts the stealing process, but stops in the middle.
- `C1` notices that its chunk is stolen, takes the *last task*, and goes away. Note that during this procedure, the `currentNode` variable becomes empty, but the node itself is still in the chunk list of `C1`.
- `C1` executes the second part of `consume()` function (see lines 68–71). It selects the same node with the stolen task and calls `takeTask()`.
- `C1` notices for the second time that the chunk is stolen and again takes one *last task* from it. If the chunk is fully consumed, `C1` returns it to its own chunk pool.
- `C2` finishes the stealing process, notices that one task was consumed by `C1` and adjusts the index in its own node to reflect it.
- `C2` tries to take the next task and receives `TAKEN` instead. As it should not be possible at this stage, there are no checks to detect it, so `takeTask()` returns `TAKEN` as a valid task to the consumer.
- `C2` finishes working with the chunk and returns it to its own chunk pool, which combined with previous actions of `C1` registers the same chunk in two different chunk pools, causing a potential conflict between producers.

All the issues we found on the following configuration: single producer; two consumers; no more than two chunks, two nodes, and two tasks per chunk. On a 16 core Ryzen 5950x CPU, each TLC execution took only a couple of seconds until an invariant violation. Most of the error traces were around one hundred steps of execution. Since we did not reach checking of temporal properties, RAM usage was minimal.

6 Discussion

We identified the following risks that may affect the validity of the findings:

1. Misinterpretation of the SALSA paper and the pseudocode.
2. Possible divergence between the pseudocode and the specification.
3. Mistakes during error trace analysis, wrong conclusion.
4. Wrong configuration of the specification.
5. Issues that are known, documented, or fixed.

To mitigate the first three risks, the specification and the error traces were independently audited by a separate person from our team; no issues affecting the results were found. As far as we can tell, all the discovered errors arise when consumers try to steal chunks from each other via `steal()` function and are located within inter-consumer synchronization code. Producers do not interact with each other at all, and it seems that there are no bugs in the logic of production and consumption of tasks without stealing We attempted to confirm this observation by turning off stealing in our model. Such a configuration successfully passes all our invariants, as incomplete as they may be. However, the simplified algorithm is hardly useful and does not guarantee the declared performance properties.

Although all reported issues were found on a specific set of the specification parameters, we extended our analysis to other possible configurations that were larger in scope. These configurations were subject to state-space explosion, making exhaustive model checking computationally expensive. Nevertheless, additional checks replicated the same failure patterns, suggesting that the identified issues are not artifacts of a restricted scope and that the chosen smaller configuration can represent the algorithm's general behavior.

We attempted to find other publications or reports that mention SALSA; none discussed its use in production or potential issues. An extended technical report [13] by the same authors describes SALSA with slight deviations in the pseudocode. We tried to formalize it as well, but quickly encountered similar issues, and decided to focus solely on the conference version.

Unfortunately, neither regular nor extended description provides a link to the source code used in benchmarks, so we were unable to compare the specification with the implementation.

A thorough discussion with the SALSA authors would be beneficial to further validate and explore our findings.

7 Related Work

There are quite a few works related to the formal verification of lock-free algorithms using interactive theorem provers. Carbonneaux et al. applied the Rocq prover (formerly named Coq [5]) and the implementation of concurrent separation logic called Iris [18] to verify two queue data structures used for inter-process communication in their microkernel [7]. The same techniques were also applied to verify a concurrent queue from Meta's Folly library [27]. Doherty et al. describe a semi-automated verification of an optimized version of Michael and Scott's lock-free FIFO queue via forward and backward simulations [10]. Tofan et al. presented KIV [25] proofs that cover memory safety, ABA prevention, and preservation of linearizability and lock-freedom for a lock-free stack with safe memory reclamation [26]. The formally proved properties are quite similar to what we check (section 4). But while theorem provers provide general and unbounded correctness guarantees, model checkers, such as TLC used in our work, focus on automatic bug detection in finite instances and bounded configurations, freeing developers from manual construction of mathematical proofs.

TLA+ also has an extensive history of usage for specification and verification of non-blocking algorithms. Hackett et al. describe a methodology for validating concurrent implementations against high-level TLA+ specifications [14]. It is evaluated on a state-of-the-art lock-free data structure from the research community and a lock-free queue featuring aggressive performance optimizations. Their approach is capable of finding known bugs injected into the systems under test and helped discover two previously unknown bugs. Hurault and Quéinnec used TLA+ and TLAPS to prove the correctness and completeness of a wait-free concurrent algorithm used to rename processes [16].

Amazon Web Services (AWS) has integrated formal methods [23] into the engineering workflow to verify the correctness of their most critical distributed systems, including lock-free data structures. They reported significant benefits of using TLA+, most notably the ability to discover subtle high-severity bugs and to verify aggressive performance optimizations. According to the publication, engineers were able to learn TLA+ from scratch and to get useful results in two to three weeks. Recently though, their focus shifted from TLA+ to P [6]: a high-level state machine based programming language also capable to formally model and validate distributed systems.

8 Conclusion

It is almost inevitable to encounter minor inaccuracies and errors while reading descriptions of lock-free algorithms, though this can be generalized to any other system as well. In articles describing queues, it is customary to provide a simple textual proof of their correctness. In our experience, such proofs are incomplete and insufficient, yet almost no one performs formal verification of their submissions. This is particularly dangerous because any inaccuracy, let alone a design flaw, can lead to serious consequences when using a queue in practice. Formal methods enable finding and fixing such problems at an early stage of development, where the cost of error is not yet prohibitive.

In this work, we applied the TLA+ formal specification language to evaluate the SALSA algorithm within the context of a microkernel redesign. A key factor was the use of the TLC model checker, which provided a fully automated verification approach that helped us finish the work in just two person-months. The analysis exposed several issues, including design flaws that undermined the algorithm's reliability in concurrent environments. This early detection of flaws likely preempted months of costly debugging and potential system failures during later development stages.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010). https://doi.org/10.1017/CBO9781139195881
2. Abrial, J.R., Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. Fundamenta Informaticae **77**(1–2), 1–28 (Jan 2007)
3. Alistarh, D., Censor-Hillel, K., Shavit, N.: Are Lock-Free Concurrent Algorithms Practically Wait-Free? Journal of the ACM (JACM) **63**(4) (Sep 2016). https://doi.org/10.1145/2903136
4. Ben-David, N., Blelloch, G.E., Wei, Y.: Lock-Free Locks Revisited. In: Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. p. 278–293. PPoPP '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3503221.3508433
5. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Springer (2004). https://doi.org/10.1007/978-3-662-07964-5

6. Brooker, M., Desai, A.: Systems Correctness Practices at Amazon Web Services. Communications of the ACM **68**(6), 38–42 (Jun 2025). https://doi.org/10.1145/3729175
7. Carbonneaux, Q., Zilberstein, N., Klee, C., O'Hearn, P.W., Zappa Nardelli, F.: Applying Formal Verification to Microkernel IPC at Meta. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 116–129. ACM, Philadelphia PA USA (Jan 2022). https://doi.org/10.1145/3497775.3503681
8. Cong, G., Bader, D.: Lock-Free Parallel Algorithms: An Experimental Study. In: Bougé, L., Prasanna, V.K. (eds.) High Performance Computing - HiPC 2004. pp. 516–527. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30474-6_54
9. Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., Vanzetto, H.: TLA+ Proofs. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods. pp. 147–154. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_14
10. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal Verification of a Practical Lock-Free Queue Algorithm. In: Formal Techniques for Networked and Distributed Systems – FORTE 2004. pp. 97–114. Springer Berlin Heidelberg (2004). https://doi.org/10.1007/978-3-540-30232-2_7
11. Fraser, K.: Practical Lock-Freedom. Tech. rep., University of Cambridge, Computer Laboratory (Feb 2004). https://doi.org/10.48456/tr-579
12. Gidron, E., Keidar, I., Perelman, D., Perez, Y.: SALSA: Scalable and Low Synchronization NUMA-aware Algorithm for Producer-Consumer Pools. In: Proceedings of the 24th ACM symposium on Parallelism in algorithms and architectures - SPAA '12. p. 151. ACM Press, Pittsburgh, Pennsylvania, USA (2012). https://doi.org/10.1145/2312005.2312035
13. Gidron, E., Keidar, I., Perelman, D., Perez, Y.: SALSA: Scalable and Low Synchronization NUMA-aware Algorithm for Producer-Consumer Pools. Tech. rep., Technion (2012), https://ece.technion.ac.il/wp-content/uploads/2021/01/publication_807-1.pdf
14. Hackett, F., Wrench, E., Macko, P., Davis, A.J.J., Wei, Y., Beschastnikh, I.: Trace Validation of Unmodified Concurrent Systems with OmniLink (2026), https://arxiv.org/abs/2601.11836
15. Herlihy, M.P., Wing, J.M.: Linearizability: a Correctness Condition for Concurrent Objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (Jul 1990). https://doi.org/10.1145/78969.78972
16. Hurault, A., Quéinnec, P.: Proving a Non-blocking Algorithm for Process Renaming with TLA+. In: Tests and Proofs. pp. 147–166. Springer International Publishing (2019). https://doi.org/10.1007/978-3-030-31157-5_10
17. Jackson, D.: Alloy: a Lightweight Object Modelling Notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (Apr 2002). https://doi.org/10.1145/505145.505149
18. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming **28** (2018). https://doi.org/10.1017/S0956796818000151
19. Konnov, I., Kukovec, J., Tran, T.H.: TLA+ Model Checking Made Symbolic. Proc. ACM Program. Lang. **3**(OOPSLA) (Oct 2019). https://doi.org/10.1145/3360549

20. Lamport, L.: The Temporal Logic of Actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (May 1994). https://doi.org/10.1145/177492.177726
21. Lamport, L.: Specifying Concurrent Systems with TLA+. Calculational System Design pp. 183–247 (April 1999), https://www.microsoft.com/en-us/research/publication/specifying-concurrent-systems-tla/
22. Lamport, L.: The PlusCal Algorithm Language. In: Theoretical Aspects of Computing - ICTAC 2009. pp. 36–60. Springer Berlin Heidelberg (2009). https://doi.org/10.1007/978-3-642-03466-4_2
23. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services Uses Formal Methods. Commun. ACM **58**(4), 66–73 (Mar 2015). https://doi.org/10.1145/2699417
24. Nipkow, T., Klein, G.: Concrete Semantics. Springer Cham (2014). https://doi.org/10.1007/978-3-319-10542-0
25. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured Specifications and Interactive Proofs with KIV, pp. 13–39. Springer Netherlands (1998). https://doi.org/10.1007/978-94-017-0435-9_1
26. Tofan, B., Schellhorn, G., Reif, W.: Formal Verification of a Lock-Free Stack with Hazard Pointers. In: Theoretical Aspects of Computing – ICTAC 2011. pp. 239–255. Springer Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-23283-1_16
27. Vindum, S.F., Frumin, D., Birkedal, L.: Mechanized Verification of a Fine-Grained Concurrent Queue from Meta's Folly Library. In: Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 100–115. New York, NY, USA (2022). https://doi.org/10.1145/3497775.3503689
28. Xiao, L., Hou, Z., Zhu, H., He, M., Qin, S.: Specifying and Verifying Programs Over the MCA ARMv8 Architecture with TLA+. Journal of Circuits, Systems and Computers **35**(01) (2026). https://doi.org/10.1142/S0218126625300089
29. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA+ Specifications. In: Correct Hardware Design and Verification Methods. pp. 54–66. Springer Berlin Heidelberg (1999). https://doi.org/https://doi.org/10.1007/3-540-48153-2_6