DOI: 10.15514/ISPRAS-2026-38(6)-25



Using TLA+ and the Tsay-Li Framework for Building Transactional Memory Models

E.D. Pervunetskikh, ORCID: 0009-0006-9576-3116 <erdmpervunetskikh@edu.hse.ru>
HSE University,
11, Pokrovsky Blvd, Moscow, 109028, Russia.

Abstract. This paper adapts the Tsay-Li framework, designed for non-blocking binary search trees, to software transactional memory (STM) models, which simplify the non-blocking implementation of complex concurrent algorithms. Using TLA+ (language for temporal logic of actions) specifications and verification with the TLC model checker, we demonstrate how formal methods prove the correctness of universal algorithms built from specialized components. This approach addresses the challenge of developing other specialized frameworks by enabling refining rollbacks from the state of the general STM. The results show that lock-based and non-blocking algorithms share more structural similarities than their source code suggests. This insight justifies the creation of universal constructions for achieving lock-free properties.

Keywords: concurrent data structures; temporal logic; formal verification; non-blocking algorithms; binary search trees; transactional memory.

For citation: Pervunetskikh E.D. Using TLA+ and the Tsay-Li Framework for Building Transactional Memory Models. Trudy ISP RAN/Proc. ISP RAS, 2026, vol. 38, issue 6, part 2, pp. 57-66.
DOI: 10.15514/ISPRAS-2026-38(6)-25.

Acknowledgements. This article is an output of a research project (HSE-BR-2025-024) implemented as part of the Basic Research Program at HSE University.

Использование TLA+ и методологии Цая-Ли для построения моделей транзакционной памяти

Е.Д. Первунецких, ORCID: 0009-0006-9576-3116 <erdmpervunetskikh@edu.hse.ru>
Национальный исследовательский университет «Высшая школа экономики»,
Россия, 109028, г. Москва, Покровский бульвар, д. 11.

Аннотация. В работе адаптирована методология Цая-Ли, предложенная для неблокирующих бинарных деревьев поиска, к моделям программной транзакционной памяти, которая упрощает неблокирующую реализацию сложных конкурентных алгоритмов. На примере спецификаций на языке темпоральной логики действий TLA+ и верификации с помощью инструмента проверки моделей TLC доказывается корректность универсального подхода. Полученные результаты показывают, что блокирующие и неблокирующие алгоритмы имеют больше структурных сходств, чем их исходный код. Этот вывод обосновывает создание универсальных конструкций для достижения неблокируемости.

Ключевые слова: параллельные структуры данных; темпоральная логика; формальная верификация; неблокирующие алгоритмы; двоичные деревья поиска; транзакционная память.

Для цитирования: Первунецких Е.Д. Использование TLA+ и методологии Цая-Ли для построения моделей транзакционной памяти. Труды ИСП РАН, 2026, том 38, вып. 6, часть 2, стр. 57–66 (на английском языке). DOI: 10.15514/ISPRAS-2026-38(6)-25.

Благодарности: Исследование осуществлено в рамках Программы фундаментальных исследований НИУ ВШЭ (HSE-BR-2025-024).

1. Introduction

In recent decades, parallel approaches have been actively developed to increase system throughput with minimal overhead. An example is the lock-free framework for binary search trees by Tsai and Lee [1], which was later adapted by Natarajan and Mittal [2] using a descriptor-based approach for structural data consistency.

This paper proposes a generalization of these methods to construct a Software Transactional Memory (STM) framework [3]. The use of STM enables the implementation of non-blocking operations, which enhances performance and provides additional benefits, such as support for nested transactions. The underlying mechanism relies on the concurrent execution of independent transactions in local memory copies, followed by either committing the changes or rolling them back in the event of conflicts. The previously mentioned approach represents a non-blocking implementation achieved through a helper mechanism for busy threads.

The primary objective of this paper is to demonstrate, using formal specifications in TLA+ (the language of Temporal Logic of Actions) [4], that the resulting general-purpose algorithm retains its correctness when generalizing the specialized approach. TLA+ describes a system as a state machine to verify safety and liveness invariants by exploring the state space. Specifications are structured around an initial state ($Init$) and a next-state predicate ($Next$). Definitions are specified using the $\triangleq$ symbol, transitions are denoted by a prime symbol ($value'$), liveness properties are represented by the $\rightsquigarrow$ (leads to) operator, and function updates are defined via the $EXCEPT$ construct. $num'=[numEXCEPT![3]=123]$ for example, updates the third element of the function num to the number 123.

2. Related Works

The Software Transactional Memory (STM) protocol described in this paper aligns with the approach proposed by N. Shavit and D. Touitou [3], which was subsequently optimized by Guerraoui et al. [5]. The primary objective of this work is to formally verify this protocol and demonstrate its derivability. In contrast to those studies, where the algorithm is presented via textual

descriptions, we propose using the TLA+ language for mathematical validation. This language is the most convenient for describing algorithms and it declares of any temporal system.

Alternative STM protocols exist (such as those by Ennals [6] and Harris [7]) that employ locks when operational conflicts arise (either through direct memory writes with a rollback log or by acquiring locks during the commit phase). The objective of the lock-free approach is to entirely eliminate blocking operations, thereby ensuring the independent execution of threads.

Furthermore, the proposed baseline mechanism does not inherently support nested transactions [9]. To implement them, it is necessary to generalize the concept of a memory cell, allowing nodes to store references to operation descriptors – a task that will be addressed throughout this paper. In the context of nested transactions, recursive thread helping can lead to an interleaving where a thread attempts to help itself on the exact same set of nodes. To resolve this issue, references to other descriptors within the local states of current descriptors can be utilized. Despite the implementation complexity, the safe descriptor reuse mechanism guarantees the linearizability [9] of the algorithm, as concurrent references will not be processed simultaneously with standard ones within a single phase of operation execution.

3. Advanced Tsay-Li framework

The work of Tsai and Lee [1] aimed to show than even complex data structures (such as binary search trees) with lock-free properties maintain linearizability. Simpler structures, such as linked lists, do not require such approaches; for them, non-blocking execution is achieved through logical deletion (as seen in the work of Heller et al. [10]). The objective of the non-blocking approach is to guarantee that at least one thread will successfully complete a data structure method infinitely often. These methods must be presented as atomic operations that preserve all partial execution orders. To achieve this, a window-based approach was proposed.

In this context, a window (or "frame" / subtree) is defined as a region of the tree that originates at a specific vertex and is bounded from below by a set of terminal vertices. The size of the window corresponds to the number of elements it contains and includes all edges of the original tree connecting these nodes. The window is selected in such a manner that its complete atomic replacement preserves the internal invariants of the data structure.

The Tsai-Lee method partitions the nodes accessed by each operation into these windows. For instance, to insert a node into the tree, the algorithm clones the window originating at the root node. The target operation is then executed on these nodes. If a concurrent modification is detected, the operation restarts. Otherwise, as the process progresses deeper into the tree, a new window is allocated. Such a window always overlaps with the previous one, thereby ensuring global visibility of the executed operations (Fig. 1).

This implementation was originally designed for the DEC Alpha architecture [11], which supports Load-Linked/Store-Conditional instructions. However, in modern architectures, the atomic compare-and-swap (CAS) operation has become dominant instead. Due to its theoretical universality and integration with helper mechanisms, it allows for the construction of any complex lock-free algorithms, which remains an active area of research [12].

To address the consistency violation that occurs when a concurrent thread performs a CAS on an internal node of a window and becomes isolated from the active topology, Natarajan et al. [13] introduced operation descriptors. Before executing an operation, a thread attaches a reference to its descriptor to all involved nodes; if another thread accesses a marked node, it is required to help the original thread complete its execution, recursively if necessary. This helping mechanism, similar to the multi-word CAS by Ellen et al. [14], guarantees both system-wide progress and strict linearizability under concurrent modifications.

Let us formalize this window-based update in TLA+ to ensure its correctness. The TreeCAS module is modeled around a minimal tree consisting of a root and its child leaf. This structure is sufficient for correctness verification, since the critical race condition that needs to be evaluated involves a

larger window being overlapped by a smaller one. Adding more nodes does not disrupt the operation, as confirmed by model checking runs.

The state of the model is primarily defined by the following variables (Fig. 2):

1) *nodeStruct* and *structLinks* map nodes to their internal structures and target child nodes;
2) *nodeDesc* tracks descriptor references (or *NULL*) within memory cells;
3) *descState* manages the descriptor lifecycle (*IDLE*, *ACTIVE*, *COMMITTING*, *SUCCESS*, *ABORTED*);
4) *descPending* and *descFrame* collectively track unacquired nodes and assigned window boundaries for each descriptor.

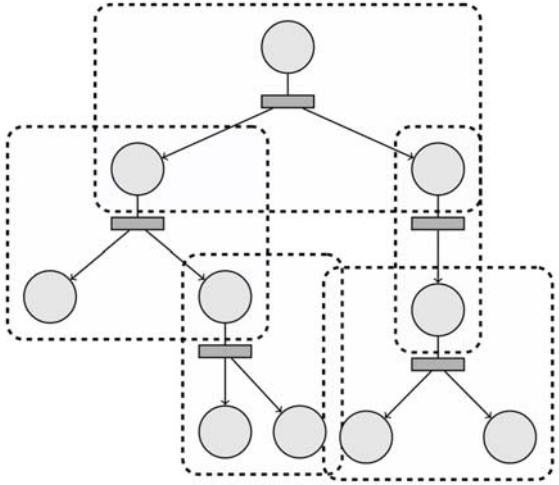


Fig. 1. Decomposition of a binary tree into always overlapping frames.

```
Совместное продвижение дескриптора, если он блокирует наш путь
    Cooperatively advance for descriptor if it blocks our path
Help(d, n) ≜ let otherD ≜ nodeDesc[n]in
  ∧ descState[d] = "ACTIVE"
  ∧ n ∈ descPending[d]      мы активно пытаемся захватить узлы
                              we are actively trying to acquire nodes
  ∧ nodeDesc[n] ≠ NULL      Целевой узел должен быть сейчас
                              The target node must be currently
  ∧ nodeDesc[n] ≠ d         заблокирован другим дескриптором otherD
                              locked by a different descriptor otherD
  ∧ if descState[otherD] ∈ {"ACTIVE", "COMMITTING"}
    then Если конкурент находится в процессе выполнения
      CommitStep(otherD)  If the competitor is in progress
    else if descState[otherD] ∈ {"SUCCESS", "ABORTED"}
    then Помогаем ему разблокировать узлы и освободить память
      ReleaseDescriptor(otherD)  Help it unlock nodes and free memory
    else Если otherD находится в состоянии IDLE, ничего не делаем
      unchanged vars  If otherD is IDLE, do nothing
```

Fig. 2. The helping phase for other threads to guarantee non-blocking operations.

In the algorithm, the concepts of an execution thread and an operation descriptor are not separated, which is equivalent to using coroutines in practice. When constructing formal specifications, the Help(d, n) procedure is strictly necessary only in a real-world implementation: omitting it violates the liveness property but does not affect safety invariants (Fig. 3).

$$GlobalProgress \triangleq$$
$$\exists\, d \in Descs : descState[d] = \text{“ACTIVE”} \rightsquigarrow$$
$$\exists\, u \in Descs : descState[u] = \text{“SUCCESS”}$$

Fig. 3. Liveness property guaranteeing active thread completion.

To verify linearizability, we enforced the preservation of the tree topology invariant, ensuring that every node remains continuously reachable from the root and is never erroneously obscured by a concurrent transaction.

Another aspect of the specification is the CAS acquisition phase. If the linked structure of a target node has been modified, our transaction must be aborted. The core block for this mechanism is presented in Fig. 4.

$$\begin{aligned}
&\wedge \text{ if } nodeStruct[n] \neq descExpectedStruct[d][n] \\
&\text{then } \text{Базовая структура узла изменилась с момента} \\
&\text{создания дескриптора. Прерываем транзакцию.} \\
&\quad \text{The underlying node structure has changed} \\
&\text{since the descriptor creation. Abort the transaction.} \\
&\wedge descState' = [descState \text{ except } ![d] = \text{“ABORTED”}] \\
&\wedge \text{unchanged } vars \setminus \{\dots\} \\
&\text{ else if } nodeDesc[n] = NULL \\
&\text{ then } ExecuteAcquire(d, n) \ \text{CAS SUCCESS} \\
&\text{ else unchanged } vars \ \text{CAS FAIL: Узел занят.} \\
&\quad \text{The node is occupied.}
\end{aligned}$$

Fig. 4. Logic of the CAS-based node acquisition phase.

The commit phase, executed step-by-step, preserves the invariant of structural consistency, as any access to a modified node requires interaction with the descriptor, thereby involving that thread in the execution of the next commit step.

4. Adaptation to Transactional Memory

In the original model, searching for a target node requires a top-down tree traversal, during which a thread must track all active transactions along its path. Ensuring the visibility of updates across all intersecting operations is a mandatory requirement. For arbitrary subsets without a common synchronization point, this visibility is not guaranteed, as a shared vertex for such subsets may not exist.

The subsequent description of the generalized mechanism in Section 4.1 is not without its limitations, which necessitates decoupling the operational principles of reading and writing threads– a task that will be addressed in Section 4.2.

4.1 Introduction of the Generalization

Regarding the problem outlined above, a global consensus on the state of the window is required. This limitation justifies the use of indirect edge addressing to ensure a consistent view of data during concurrent operations.

We store the data of all memory cells accessed by the current operation within the descriptor. This guarantees that any modifications to nodes and edges are performed strictly inside the descriptor.

The slots of the primary data structure now hold references exclusively, as their values are no longer independent and are not updated atomically during operations performed on them.

This two-phase approach – logical commitment followed by physical propagation of changes – mirrors the multi-word compare-and-swap (MCAS) semantics proposed by Harris et al. [15]. This ensures the linearizability of updates across multiple memory locations via a single shared descriptor. This mechanism closely approaches the STM model, demonstrating that lock-free tree primitives can be transformed into a universal construction. Furthermore, this approach provides disjoint-access parallelism, which offers a distinct advantage over blocking STMs.

Based on this descriptor-centric architecture, each transaction is partitioned into three atomic phases that ensure safe state transitions within the window:

1) Acquisition Phase. The thread attempts to install its descriptor D into each target memory cell r_j using a $CAS(r_j, old_j, D)$ operation. If a cell is already occupied by another descriptor D', the thread invokes the recursive helping procedure. To prevent deadlocks, descriptors are acquired strictly in ascending order of their addresses.
2) Decision Phase. The point of logical linearizability. Upon successfully acquiring all target cells, the thread attempts to perform an atomic CAS on the status field of the descriptor, transitioning it from the *Active* state to either *Committed* or *Aborted*.
3) Release Phase. The thread restores the state of each node to a concrete value. If the transaction status is *Committed*, the cell is updated with the value new_j; otherwise, it reverts to old_j. This refuse descriptor-based ownership and makes the modifications visible to the entire system.

The described approach aligns with composable transactional memory models [8]. However, to guarantee a linearizable view, any process accessing a cell must first resolve the pending descriptor. This dependency transforms a standard read into a "heavyweight" operation, which introduces unnecessary overhead and reduces performance in read-heavy scenarios.

4.2 Lightweight Read Model Strategy

Transitioning to a lightweight read model requires more granular tracking of concurrent dependencies, resembling the Read-Copy Update [16] mechanism in terms of monitoring critical sections. We employ an announcement array where each reader thread registers itself before commencing an operation. A writer thread is constrained to scan this array during an atomic replacement and forcibly abort readers operating on those nodes. This validation must occur strictly before the replacement is committed; otherwise, the reader will complete incorrectly.

Since a write operation that aborts concurrent readers cannot be rolled back itself, the list of values is guaranteed to be consistent with the entire data structure. A reader thread first registers in the array, then reads the current value either from the memory cells themselves or from their associated descriptors, and unregisters upon successful completion. The safety of this optimization is ensured by a memory barrier [17], which establishes a strict happens-before [18] relationship between the reader's registration and the writer's scan. Fig. 5 illustrates two possible execution interleavings of the system: visibility from the writer's perspective and awareness from the reader's perspective.

5. Formal Methodology

To verify the MCAS algorithm, a TLA+ specification was developed for the TLC model checker. The memory state is monitored via the nodeValue and nodeDesc variables. Resources and metrics are managed by the freeWDescs, freeRDescs, and successCount variables, while linearizability is validated through the absNodeValue refinement mapping. To constrain the state space, a state constraint condition of successCount ≤ 3 is introduced. The liveness property (GlobalProgress) is verified similarly to the TreeCAS module.

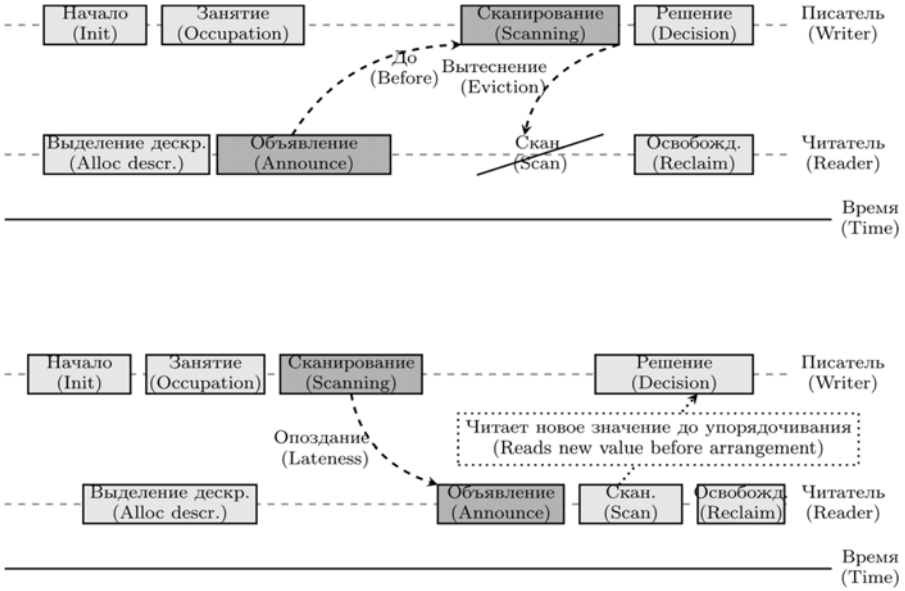


Fig. 5. Two global execution interleavings for the reader and writer processes.

Linearizability is verified using two safety invariants:

1) *AtomicyInvariant* ensures consistency between the physical state (*nodeValue*) and the abstract state (*absNodeValue*). If a cell is free, the values match. If it is occupied by an active descriptor, the value corresponds to the abstract state in the *ACTIVE* or *ABORTED* states. In the *SCANNING* or *SUCCESS* states (after the linearizability point), the physical memory may store *wDescOldValue*.
2) *ReaderAtomicy* verifies the data snapshot consistency for a reader in the *READING* state. It guarantees that the local snapshot (*rDescView*) for all target nodes is linearizable and matches the current global abstract state (*absNodeValue*).

6. Methodology Evaluation

The verification was conducted using exhaustive state space exploration with intentional fault injection to validate the specification itself. The primary challenge was the step-by-step implementation of reader eviction within the writer, which violated global atomicity. Attempting to resolve this through generalized verification by introducing abstract mappings (rAbsNodeValue) for each reader led to a violation of the foundational safety criterion (ReaderAtomicy). This proved that the eviction process must be atomic.

A configuration consisting of two writer threads and one reader thread (as readers do not interact with one another) is sufficient to verify phase overlaps and prevents an exponential growth of the state space. Process symmetry and the transitivity of conflict relations (which follows from the recursive nature of helping) guarantee that for any number of active writers, their pairwise interactions are isomorphic. Since the AcquireNode and Help operations execute in a strict queue, adding a third writer does not generate unique paths in the state transition graph.

7. Conclusion

In this paper, the Tsai-Lee framework for tree structures was adapted to a modern Software Transactional Memory model. The specification written in TLA+ confirmed that the proposed lightweight read protocol utilizing an announcement array and memory barriers preserves strict linearizability while significantly reducing the overhead associated with cooperative helping. The source code of the implementation is available in [19].

Список литературы / References

[1]. Tsay J.J., Li H.C. Lock-free concurrent tree structures for multiprocessor systems (extended abstract). In Proc. of the 1994 International Conference on Parallel and Distributed Systems, 1994. pp. 544-549. DOI: 10.1109/ICPADS.1994.590370.

[2]. Natarajan A., Mittal N. Fast concurrent lock-free binary search trees. ACM SIGPLAN Notices, 2014, vol. 49, no. 8, pp. 317-328. DOI: 10.1145/2555243.2555256.

[3]. Shavit N., Touitou D. Software transactional memory. In Proc. of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC ’95), 1995. pp. 204-213. DOI: 10.1145/224964.224987.

[4]. Lamport L. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, 2002. 382 p.

[5]. Guerraoui R., Kogan A., Marathe V.J., Zablotchi I. Efficient multi-word compare and swap. In Proc. of the 34th International Symposium on Distributed Computing (DISC 2020), 2020. pp. 4:1-4:18. DOI: 10.4230/LIPIcs.DISC.2020.4.

[6]. Ennals R. Software transactional memory should not be lock free. Intel Research Cambridge, Tech. Rep. IRC-TR-06-052, 2006.

[7]. Harris T., Fraser K. Language support for lightweight transactions. In Proc. of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA ’03), 2003. pp. 322-335. DOI: 10.1145/949305.949340.

[8]. Harris T., Marlow S., Peyton-Jones S., Herlihy M. Composable memory transactions. In Proc. of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’05), 2005. pp. 48-60. DOI: 10.1145/1065944.1065952.

[9]. Arbel-Raviv M., Brown T. Reuse, don't Recycle: Transforming Lock-free Algorithms that Throw Away Descriptors. 31st International Symposium on Distributed Computing (DISC), 2017, vol. 91, pp. 4:1-4:16. DOI: 10.4230/LIPIcs.DISC.2017.4.

[10]. Heller S., Herlihy M., Luchangco V., Moir M., Scherer III W.N., Shavit N. A lazy concurrent list-based set algorithm. In Principles of Distributed Systems (OPODIS 2005), 2005. pp. 3-16. DOI: 10.1007/11795490_3.

[11]. Sites R.L., McLellan E. The Alpha AXP Architecture and 21064 Processor. IEEE Micro, 1993, vol. 13, no. 3, pp. 36-47. DOI: 10.1109/40.216747.

[12]. Alistaih A., Jayanti P. Generalized Compare-and-Swap and Space-Efficient Universal Constructions for the Infinite-Arrival Model. Proceedings of the 2026 ACM Symposium on Principles of Distributed Computing (PODC), 2026, pp. 467-477. DOI: 10.1145/3796701.3815968.

[13]. Natarajan A., Savoie L.H., Mittal N. Concurrent wait-free red black trees. In Stabilization, Safety, and Security of Distributed Systems (SSS 2013), LNCS, vol. 8226, 2013. pp. 45–60. DOI: 10.1007/978-3-319-03089-0_4.

[14]. Ellen F., Fatourou P., Ruppert E., van Breugel F. Non-blocking binary search trees. In Proc. of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC ’10), LNCS vol. 2508, 2010. pp. 131–140. DOI: 10.1145/1835698.1835736.

[15]. Harris T.L., Fraser K., Pratt I.A. A practical multi-word compare-and-swap operation. In Distributed Computing (DISC 2002), 2002. pp. 265-279. DOI: 10.1007/3-540-36108-1_18.

[16]. McKenney P.E., Walpole J. What is RCU, Fundamentally? Available at: https://lwn.net/Articles/262464, accessed 12.07.2026.

[17]. McKenney P.E. Memory barriers: a hardware view for software hackers. Available at: http://rdrop.com/~paulmck/scalability/paper/whymb.2010.07.23a.pdf, accessed 12.07.2026.

[18]. Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978, vol. 21, no. 7, pp. 558–565. DOI: 10.1145/359545.359563.

[19]. Formal TLA+ specification of a version of the STM algorithm: MCAS.tla file. Available at: https://github.com/Yerk08/MCAS-formal-specification, accessed 12.07.2026.

Информация об авторах / Information about authors

Еремей Дмитриевич ПЕРВУНЕЦКИХ – студент, обучающийся по направлению «Программная инженерия» Факультета Компьютерных Наук с 2025 года. Сфера научных интересов: формальная верификация распределенных систем, параллельные структуры данных.

Eremey Dmitrievich PERVUNETSKIKH – Software Engineering student at the Faculty of Computer Science since 2025. Research interests: formal verification of distributed systems and concurrent data structures.