Assessing Formal Verification in SPARK
A Case-Study Evaluation of Formal Verification Tooling

Dinu Blanovschi[1]
Supervisors: Dr. Benedikt Ahrens[1]**, Kobe Wullaert**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
July 1, 2025

Name of the student: Dinu Blanovschi

Final project course: CSE3000 Research Project

Thesis committee: Dr. Benedikt Ahrens, Kobe Wullaert, Dr. Maliheh Izadi

Assessing Formal Verification in SPARK
A Case-Study Evaluation of Formal Verification Tooling

DINU BLANOVSCHI, Delft University of Technology, The Netherlands

Abstract: Formal verification promises stronger correctness guarantees than conventional testing, yet it is often perceived as too costly or specialised for everyday software development. This thesis investigates whether SPARK — Ada's provable subset — can deliver industrial-strength verification for representative algorithms and concurrent systems. Two separate questions guided the work: how well does SPARK work with (RQ1) sequential algorithms; and (RQ2) concurrent systems.

We implemented and formally proved insertion sort, quick-sort, and a generic statically-allocated hash-map to SPARK's highest assurance level, achieving full functional correctness, as well as two concurrent case studies: a Publisher / Subscriber channel and a novel IO multi-reactor pattern runtime task scheduler, where we combined SPARK with TLA$^+$ model checking to capture liveness and safety properties that SPARK cannot express directly. Across all sequential examples, proof overhead averaged ratios of 6–10 lines of specification, while for the concurrent case-studies, the TLA$^+$ models averaged ratios of 1.7 lines, per every line of executable code.

The study shows that (i) SPARK is practical for non-trivial sequential algorithms, unless they make use of the heap; (ii) concurrency still requires external formalisms, but the combination remains tractable; and (iii) careful specification design, not solver performance, is the dominant cost driver. These findings confirm that correctness-by-construction is attainable within undergraduate project scope and provide quantitative benchmarks for future work.

Keywords: formal verification; Ada; SPARK; TLA$^+$; functional correctness

1 Introduction

The most widely used practice to prove software correctness is through testing, which, while effective in general, does not inspire full confidence in the system. Approaches that emerged as a result include, but are not limited to, property-based testing [15, 19], fuzzing [9, 44], model-checking [18], and formal verification [36]. Given that there are cases where software failures could prove catastrophic, such as in the avionics [51], defense, military, and automotive industries [64], stricter guarantees of correctness are not only recommended, but necessary [10, 34, 41, 48]. Formal verification is one such approach to verification that can inspire great confidence in the correctness of the system [34].

One such language for critical software that has built-in support for formal verification is SPARK, the provable subset of Ada [56]. SPARK has multiple levels to which programs can be checked, according to the requirements [59]. `GNATprove`[1] is the tool responsible for actually formally verifying SPARK code [38]. It has made formal verification accessible for SPARK programs, yet challenges remain, particularly around concurrency and scalability to large codebases.

The goal of this project is to understand the correctness-by-construction approach to software engineering [25], as well as its limitations, with a specific focus on SPARK. To achieve this, we implemented several formally verified case studies: an implementation of insertion sort, a quicksort implementation [2], a generic HashMap implementation [1], all verified to SPARK's highest level of assurance (full functional correctness [59]), as well as a bounded publish / subscribe channel implementation backed by a circular buffer, and a novel IO-optimized work-stealing multi-threaded task scheduler, first implemented in SPARK, and then modeled in TLA$^+$ separately (and checked with TLC), to get a better understanding of how SPARK is able to reason about concurrency.

The research (sub-)questions that guided this work are as follows:

- RQ1 How can SPARK be used to formally verify sequential algorithms, and implementations of data structures? And if it cannot, why not, and what other tools can be used for this purpose?
- RQ2 How can SPARK be used to formally verify concurrent systems? And if it cannot, why not, and what other tools can be used for this purpose?

This paper is structured as follows. Section 2 offers the theoretical foundations, as well as an introduction to how the tools used actually work. Section 3 follows with an in-depth description of the properties we want to check, contracts, and interfaces, of each of the case studies implemented, as well as the prior background. Section 4 describes the high-level approach to the implementations (and the scheduler in Section 5 as it deserves its own section). Finally in Section 6 we present and discuss our implementation results, and in Section 7 we conclude with answers to the research questions. Section 8

[1] https://docs.adacore.com/spark2014-docs/html/ug/en/gnatprove.html

Author's Contact Information: Dinu Blanovschi, EEMCS, Delft University of Technology, Delft, The Netherlands.

reports on responsible research. Code is included in this paper in order to show the constructs available in SPARK / TLA$^+$ and briefly explain the high-level proofs for the case studies. For the full specifications / implementations, readers are referred to the code repository (see Appendix B). We assume some prior knowledge of Ada, and as such we will not be explaining Ada's syntax, but only the parts relevant for formal verification. learn.adacore.com is a great resource to pick up Ada in order to be able to understand the code in this paper [56].

2 Theoretical foundations

This section covers the background, starting with what formal verification is in subsection 2.1. subsection 2.2 introduces the inner workings of SPARK, and then the theory behind TLA$^+$ and TLC is described in subsection 2.3.

2.1 Formal verification and why it matters

Formal verification is an approach that attempts to prove the correctness of software through formal means. As it is typically implemented today, the programmer is asked to annotate their source code, indicating all of the *contracts* that specific interfaces uphold, and letting a tool extract these contracts and verify that the actual implementation matches the specification derived from the contracts, through the use of formal methods, the basis of which is Floyd-Hoare logic [4, 5], or one of its many extensions, such as Owicki-Gries logic [8, 52] or separation logic [23].

Formal verification has a few similarities to property-based testing (PBT) [15, 19]. Formal verification and PBT are both about proving functional properties of the code. With a formal proof that some property will hold for all the possible outputs of a given function, there will no longer be a need to check that property through PBT. The main difference between them is their approach: formal verification is static while PBT is dynamic. This difference between them gives rise to a few trade-offs, although they can be used complementarily to prove different properties. Table 1 summarizes the trade-offs between PBT and formal verification.

Table 1. Trade-offs between PBT and Formal verification

Criterion	PBT	Formal verification
Up-front investment	small	big
Debugging effort	debug counterexample	counterexamples rarely found
Performance exploring entire input state-space	infeasible	fast
Assurance provided	medium	highest

Software certification standards such as ISO26262-1:2018 and EN50716:2023 either highly recommend or mandate formal verification in safety-critical software [41, 48], and DO-178C:2012 even allows formal verification to replace some of the other forms of testing [34, 38]. Since the inception of its predecessor, DO-178B in 1992 [10], no aircraft-related fatality has been attributed to DO-178B/DO-178C-certified software [38], which speaks volumes about their reliability. There was an exception to this, namely the Boeing 737-MAX crashes in 2018 and 2019, due to assigning a wrong Design Assurance Level (DAL) to the Maneuvering Characteristics Augmentation System (MCAS), and therefore not certifying it to the level required [49]. If Boeing would have attempted to certify the software at the proper DAL, they would have identified the problems long before the crashes happened [49]. This goes to show what could happen when protocols are not followed properly.

The formal specification step is typically the bottleneck in the process of developing new software, but once it is finished, the software will be *extremely* reliable [25, 33]. Despite the bottleneck of formalization in general, programming and specification in SPARK are still up to 3 times faster than the industry average in codebases aiming for certifications, with far fewer defects [25, 32].

Given the critical importance of software reliability in domains such as avionics and defense, and SPARK's widespread use in these domains, ensuring algorithmic correctness in SPARK programs is both necessary and impactful.

2.2 A brief explanation of *how* it all works in SPARK

`GNATprove` is the tool responsible for formal verification in SPARK. It does 3 things in order to prove program correctness: flow analysis (such as uses of uninitialized variables), proving program integrity (by statically proving the absence of runtime errors, such as buffer overflows, integer arithmetic overflows), but the one of most importance for this project is proving functional correctness. In order to achieve this, it generates a lower-level version of the code, which is intended to be

consumed by Why3, a platform for deductive reasoning about programs [31, 35]. The verification condition (VC) generation is then offloaded to Why3, which converts the program into a set of VCs, which are then turned into a set of goals in Satisfiability Modulo Theory (SMT) and then discharges them to SMT solvers.

The SMT solvers then have to come up with proofs for the individual goals; a proof implies that the respective contract holds in the source code. In case such a proof cannot be found after exploring a predetermined number of nodes, or after a timeout, `GNATprove` is also able to use Why3 for finding a counterexample to the current rule, and then presents it to the programmer. A counterexample implies that the contract does not hold, and as such, further specification is required. In cases where the SMT solvers cannot find a proof, nor come up with a counterexample, it typically means that either more contracts are needed, or that the goals are really hard to prove, which is a known limitation of `GNATprove` [58].

The supported SMT solvers include `Z3` [26], `cvc5` [46], `colibri` [43], and `Alt-Ergo` [40], but by default `GNATprove` only uses `cvc5` [61]. As such, the logical foundations on which SPARK is based on are First-Order logic and Floyd-Hoare logic [4, 5].

Ada provides the following constructs for concurrent code, as part of the language [56]: tasks, analogous to threads from other languages, entry methods, aka rendezvous points, which allow for message passing between different tasks without the use of mutable shared state, protected objects, which allow for simpler and more efficient synchronization between tasks than entry methods, and semaphore objects. These constructs shaped some of the design decisions behind SPARK, and as such, SPARK tooling is partially able to reason about uses of Ada's concurrency features, but has limitations in reasoning about other forms of mutable state in concurrent contexts, such as lock-free atomics.

Tasking profiles in Ada, such as Ravenscar and Jorvik, are sets of restrictions on how to write concurrent code; Ravenscar is intended for high-integrity software with hard safety requirements, whereas Jorvik is a newer profile that lifts some restrictions from Ravenscar, intended for real-time software, but not necessarily in the safety-critical category [63].

In the context of concurrency, a safety property can be thought of as "nothing bad ever happens," while a liveness property can be interpreted as "something good eventually happens." Ravenscar's restrictions allow SPARK to prove some safety properties, but liveness properties require a different approach altogether.

The specific safety properties that SPARK can guarantee include the absence of data races, mutual exclusion with protected objects, and under the Ravenscar tasking profile, also the absence of deadlocks (partially). SPARK by itself is unable to reason about liveness nor the other safety properties.

2.3 A short introduction to TLA$^+$ and model-checking

Here we introduce TLA$^+$ [11, 22, 27, 69], its model checker TLC [18], and how we can take advantage of them to verify our concurrent systems.

TLA$^+$ is a specific form of logic, in which we can express first-order logic, as well as set theory and temporal logic. We are interested in temporal logic because we need it to express and verify safety and liveness properties that SPARK cannot express by itself.

Formally put, a model is a 3-tuple: $\langle S, I, T \rangle$, where S is the set of all possible states, I is the set of initial states ($I \subseteq S$), and T is the set of possible transitions in the system ($T \subseteq S \times S$).

A model checker is a tool that checks models, by exhaustively analyzing all possible states and transitions in the modelled system, in order to prove certain properties of this system. To model-check a TLA$^+$ model with TLC, the I set has to be finite, but S and T can be infinite. If they are infinite, then the model-checker either stops when all the transitions it can take have been explored already, or it is stopped manually.

Manna and Ashcroft have shown that a concurrent program can be described in terms of a non-deterministic sequential program [6]. For example, running two threads in parallel, where the first thread runs statements $[S1; S2]$ and the second runs statements $[S3; S4]$, is the same as non-deterministically choosing between any of the following possible behaviors, each of which corresponding to a sequential program:

$$[S1; S2] \ \| \ [S3; S4] = \text{CHOOSE} \begin{cases} [S1; S2; S3; S4] \\ [S1; S3; S2; S4] \\ [S1; S3; S4; S2] \\ [S3; S4; S1; S2] \\ [S3; S1; S4; S2] \\ [S3; S1; S2; S4] \end{cases}$$

This process is called task-interleaving. To prove a concurrent system holds a certain property, we should prove that said property holds for all the possible task-interleavings. This is fundamentally what TLC does when model-checking a concurrent TLA$^+$ specification: with a bit of modelling, statements are essentially transitions in T.

The number of possible behaviors grows quite quickly: with only 2 threads, each doing n and m atomic assignments respectively, the number of task-interleavings is $(m+n)!/(m! \cdot n!)$, so Ashcroft in 1975 published another paper in which he tries to optimize this: he realized that he can keep track of the current state storing the next instruction that each thread would run, and then check if any two threads accessed the same state in the next instruction; otherwise the order in which they ran wasn't relevant, as the results would be the same [7]. With this insight, model checkers can now perform faster, as they would have to check much fewer states. The way this is implemented in TLC is through keeping a set of hashes (called fingerprints) for the already visited states, and whenever multiple different transitions lead to the same state, that state is explored just once rather than multiple times [18].

Another insight is that we can apply Hoare logic to concurrent algorithms. This is what Owicki explored in her thesis [8]. She noticed that Hoare logic can be used to verify the individual threads, and then reason that the way they interact with the task-interleaving approach does not violate their assertions. This is called Owicki-Gries logic.

TLA$^+$ is a fundamentally different logic. It is based on first-order logic, set theory, and TLA. TLA (temporal logic of actions) itself is a logic that introduces, among other things, the following *temporal* operators [11]:

- Always: written as $\Box F$, it means that temporal formula F will always hold.
- Eventually: written as $\Diamond F$, it means that temporal formula F will hold at some point in the future. Formally it is defined as $\Diamond F \triangleq \neg\Box\neg F$ [11].

Those can be combined to give "eventually always": $\Diamond\Box F$. This can be understood as F will be true at some point, and then it will stay true forever.

As an alternative to TLC, TLAPS (TLA$^+$ Proof System) is a formal verification tool for TLA$^+$ specifications [30]. As of now, it does not support the full set of features of TLA$^+$, and is still in active development [72], so a possible future extension would be formally verifying the concurrent systems in TLAPS, once all features are supported.

3 Problem Description

In order to understand how SPARK can be used for formal verification, a couple of case studies have been implemented. The goal was to achieve full functional correctness in all of the sequential algorithms and liveness and safety in the concurrent ones. As such, this section will cover the specific properties that the implemented case studies abide by. The goal of the case studies is then to implement the algorithms / structures / systems in such a way that the properties we describe here hold. An algorithm / system will be considered correct only if it conforms to the properties described here.

The problems have been ordered in increasing order of complexity, to be able to introduce the key constructs learned along the way one step at a time.

Given that SPARK is unable to reason about liveness, to properly reason about the concurrent systems, once we had a working system in SPARK, we created models in TLA$^+$ [11, 22], and then checked them with TLC [18].

3.1 Sorting algorithms

Sorting algorithms are algorithms that, as the name implies, sort a list of items. They are used in many more complex algorithms and as such have great utility.

A sorting algorithm is correct if it modifies the list such that, the sorted list, e.g. the new version of the list, after invoking the sorting algorithm, is (1) a permutation of the input, which (2) is sorted.

As such, the two post-conditions that are enough to fully prove the correctness of a sorting algorithm are:

- `Weak_Sorted(A)`: `for all I in A'First+1..A'Last => A(i-1) <= A(i)`, where `A'First` and `A'Last` refer to the first and last indices of the array `A`, respectively.
- `Multiset_Unchanged(A, A'Old)`: Keeps the old values, or more formally put, is a permutation of the old array. Expanding the definition: `for all X in Integer => Occ(A, X) = Occ(A'Old, X)`, where `Occ(A, V)` refers to the number of occurrences of `V` in array `A`, and `A'Old` is SPARK's way to refer to the old value of parameter `A` before the call.

Inspiration for the specifications and proofs came from the spark-by-example examples on sorting [42, 54].

The chosen sorting algorithms were insertion sort and quick-sort [2]. Insertion sort was previously proven in spark-by-example [42, 54]. They have two implementations for insertion sort, one of them does the insertion in an unoptimized way, whereas the other does it the optimized way. Spark-by-example, at the time of writing, only has proofs for the unoptimized version, with a "Version of `Insertion_Sort` using Rotate (not proved)" comment on the optimized version (using rotate) [54]. We have formally proven the optimized version. There was no implementation or proof for quick-sort in spark-by-example.

3.2 Hash map

A hash map is a key-value data store. That is, it maps specific keys to specific values, and the values can later be retrieved from the map by their specific key.

Ada already has hash maps defined in the standard library [56], as well as structures in SPARK for exposing those interfaces with contracts that allow for formal verification of the client code, but not of the implementations themselves [53, 57]. It doesn't have a standard hash map with a fully formally verified implementation. This case study is about formally verifying a bounded, statically allocated hash map. SPARK is able to reason about static allocations, but has very limited support for dynamic allocations. Recent work on SPARK has introduced aliasing analysis, allowing `GNATprove` to formally verify some heap structures, although full separation logic is still out of reach, due to poor support in SMT solvers [45].

For the hash map, the following contracts have been created for each of the functions / procedures:

3.2.1 Checking if a hash map contains a key. This function doesn't have any pre- or post-conditions, as it only returns a boolean informing us of whether the given key is in the map or not.

```
function Contains_Key (HM : Hash_Map; Key : K) return Boolean;
```

This brings up a crucial insight about SPARK: this function can be used to specify that the hash map will have a certain key in the pre- and post-conditions of `Insert` and `Delete_Key`, even though we don't know what will actually happen when we run this function.

3.2.2 Fetching a value. This function returns the value associated with the given key in the given hash map. Of course, the map has to have the given key, and so that is the pre-condition. There is no post-condition, as we cannot guarantee anything about the contents of the returned value at this point, but similarly to `Contains_Key`, this function can be used as a post-condition for the other procedures, even though the implementation details are hidden.

Given the generic nature of our implementation, we cannot always be sure that `null` is a valid value, and as such, the key *must* exist in the map.

```
function Get_Value (HM : Hash_Map; Key : K) return V
with Pre => Contains_Key (HM, Key);
```

3.2.3 Insertion. For insertion, the pre-condition is that the bucket that the key will go in is not full, if the key doesn't exist already in the bucket. If the key does exist, then we will just replace the value. In either case, in the end, the hash map will contain the key, and the value will be equal to the given value.

```
procedure Insert (HM : in out Hash_Map; Key : K; Value : V)
   with Pre  => (if not Contains_Key (HM, Key) then not Bucket_Key_Is_Full (HM, Key)),
        Post => Contains_Key (HM, Key) and then Get_Value (HM, Key) = Value;
```

When we verify the implementation, we should show that the implementation makes the post-condition true, i.e. `Contains_Key (HM, Key) and then Get_Value (HM, Key) = Value`, but, when verifying client code using the function, the client code doesn't need to know anything about how either `Contains_Key` or `Get_Value` are implemented, it only needs to know that calling those functions after calling `Insert` will have the results we put as post-condition; this preserves encapsulation.

3.2.4 Deletion. Deleting a key is somewhat similar, the key has to exist the in map before we call this method, and so that is the pre-condition. The post-condition says that the key will not be in the map after the procedure finishes.

```
procedure Delete_Key (HM : in out Hash_Map; Key : K)
with Pre  => Contains_Key (HM, Key), Post => not Contains_Key (HM, Key);
```

3.2.5 Additional contracts. There are additionally the following boiler-plate contracts omitted from the definitions above for brevity, but still required for a fully correct implementation of hash maps:

- `All_Buckets_Have_Unique_Keys (HM)`: a type invariant, which encodes the property that there are no duplicate keys in the hash map.
- `No_Changes_Other_Than_To_Key(A, B, K)`: Describing the contract that if key `Key` is in `B` and it is different from `K`, then the key `Key` is also in `A`, and furthermore, `Get_Value(A, Key) = Get_Value(B, Key)`.

The following conjunction makes sure that, in the case that we are modifying the hash map, (1) none of the keys previously in the map disappear, and (2) none of the new keys in the map are different from `Key`, or to put it differently, no other new keys appear in the map:

```
No_Changes_Other_Than_To_Key(HM, HM'Old, Key) and then No_Changes_Other_Than_To_Key(HM'Old, HM, Key)
```

3.3 Publish/Subscribe channel

A publish/subscribe channel, or pub/sub for short, is a channel in which multiple tasks can publish messages, and multiple tasks can subscribe and receive messages from it. This is conceptually similar to the "Observer pattern" in object-oriented programming [12], though they are often implemented very differently.

We have implemented a bounded publish / subscribe channel, that is, a channel which might drop some of its messages, in order to save space, in case subscribers lagged behind too much.

Given its bounded nature, the liveness property of all messages being delivered to all subscribers is impossible to prove, unless the tasks that interact with the channel synchronize in some other way. We can however, prove that all messages will be either delivered or missed, eventually, for each of the subscribers, provided there is at least one fair process that keeps polling each subscriber (as in, the process will not fail or be starved indefinitely).

This is expressed in TLA$^+$ as follows:

```
variables
    SeenVersions = [x \in ReceiverIds |-> [m \in MsgRange |-> FALSE]];
      \* maps subscribers to messages that they saw
    MissedVersions = [x \in ReceiverIds |-> [m \in MsgRange |-> FALSE]];
      \* maps subscribers to messages that they missed

\A r \in ReceiverIds:           \* for all subscribers
\A m \in MsgRange:              \* for all messages
<>(                             \* eventually
    [](SeenVersions[r][m])      \* the message will always be delivered to this subscriber
    \/ [](MissedVersions[r][m]) \* or, the message will always be missed by this subscriber
)
```

This makes sure that no message will ever be able to switch between delivered and missed after they have been either delivered, or missed.

3.4 IO-optimized task scheduler

Work stealing is an approach to task scheduling, in which each thread has a queue of tasks (units of computation). Other threads, when they do not have any work to perform, will "steal" some work from one of the other busy threads [16]. This approach is different from work-sharing, where the scheduler runs separately and dispatches threads to different processors, whereas in work-stealing, the processors themselves take the initiative to take on work from other threads, and as such, there is no need for a dispatcher thread. Work stealing has been formally proven to achieve optimal performance for the restricted class of fully-strict computations (i.e. tasks only ever synchronize with their direct parent) [14, 21, 24, 50], and even for some computations that are not fully-strict [29], with experimental results showing that it is also more energy-efficient [37].

Several libraries / runtimes have been implemented for multi-threaded work-stealing task schedulers. We have identified 2 different categories: compute-optimized and IO-optimized task schedulers, each with their own strengths and weaknesses. Examples of Compute-optimized work-stealing task scheduler libraries / runtimes include Cilk in C [13], Rayon in Rust [39, 55], TPL for .NET [28], a fork/join framework in Java [20], and the Parasail LWT scheduler for Ada 2022, in Ada [47, 71], while examples of IO-optimized work-stealing task schedulers include the goroutine runtime in Golang (specifically netpoller [70]) [65], the multi-threaded scheduler in Tokio in Rust [73], and Akka in Java/Scala [60]. The critical component in an IO-optimized scheduler is an *IO reactor*: an event loop software design pattern in which we can register events, and at the next iteration of the event loop, we check whether any events we are interested in have happened (such as, we have some data we can read on a socket), and then demultiplexes them, invoking the appropriate handlers.

Given the already existing Parasail LWT compute-optimized work-stealing task scheduler written for Ada 2022, in Ada [47, 71], this case study was about implementing a similar, asynchronous IO-optimized work-stealing task scheduler in Ada. As such, the ultimate goal for this project was to offer a solution to the C10k problem, fully implemented in Ada. A solution to the C10k problem is a system that can handle large amounts of concurrent sockets, with the name standing for 10.000 connections [17]. It doesn't need to do Input/Output very fast (which would be through-put), but it must be able to handle a lot of connections with minimal latency [17]. As such, it needs help from the scheduler and an IO reactor, to be able to schedule individual tasks *only* when they have some I/O available.

The general properties of this system we would like to check are:

(S1) Liveness / safety, no starvation: once a task has some IO available, or the timer it was waiting on expires, and is marked as *ready* as a result, it will run eventually (weak fairness, unbounded, needs to be checked as a liveness property) or within a bounded amount of time (can be checked as a safety property, e.g. never at any point in the program's execution is there a task that hasn't been polled within this time; this is called bounded fairness or, specifically in the case of our system, *size-parametric fairness*, as in, the bound depends on the size of the system).
(S2) Safety: Tasks do not run while they indicated that they are waiting on a timer or some IO.
(S3) Safety: Once a task has finished, it doesn't get scheduled again.
(S4) Safety: No tasks are dropped before they finish, raise an error, or are cancelled.

4 Implementation

4.1 Insertion sort

The approach in insertion sort is to build a sorted array, one item at a time. To achieve this, it starts from a sorted array containing the first element, which is already sorted. At each step, it takes the next element, and inserts it at the right position in the sorted list. As such, the following code listing gives a high-level overview of the algorithm:

```
-- if A'Length < 2 then return
-- if A(A'First) > A(A'First + 1) then swap(A, A'First, A'First + 1)
for I in A'First + 1 .. A'Last - 1 loop
   pragma Loop_Invariant (Weak_Sorted (A (A'First .. I)));
   pragma Loop_Invariant (Multiset_Unchanged (A, A'Old));
   ipt := insert_point (A, I, A (I + 1)); -- find position to insert the next element
   if ipt < I + 1 then
      move_forward (A, ipt, I + 1);       -- rotate A(ipt..I+1)
   end if;
end loop;
```

The loop invariants mentioned above are enough to fully prove the correctness of the sorting algorithm.

The first invariant says that on each iteration of the for loop the array is sorted up until index `I`. `move_forward` has as a pre-condition that the array is sorted up until index `I`, and as a post-condition that it is sorted up until index `I + 1`. `insert_point` does not modify the array. To prove the invariant, `GNATprove` uses induction: it first proves that it holds before the loop (the base case, $p(0)$), and then proves that the body of the loop, during an arbitrary iteration, makes the invariant for the next loop iteration hold (induction step, $p(k) \rightarrow p(k+1)$). If it can show both of them, then the loop invariant will hold after the loop, and so we managed to prove that we sorted the entire array. The base case is proven due to the swap in the **`if`** right above the loop. The induction step is proven by the post-condition of `move_forward`.

The second invariant is much easier to explain. `move_forward` has as a post-condition that `Multiset_Unchanged(A, A'Old)`. So in our main loop, the induction step is trivial to prove, and in the base case we can show that it also holds, as the array is either unchanged or we swapped the first 2 elements (and swap has the same `Multiset_Unchanged(A, A'Old)` postcondition).

4.2 Quicksort

The quick-sort approach is to partition the array into a sub-array of elements that are smaller than the pivot and a sub-array of elements that are larger than the pivot, and then recursively calling the sorting algorithm on these sub-arrays [2].

As such, the algorithm was modeled roughly as follows:

```
pivot := A (A'First);
i := A'First + 1;
j := A'Last;

while i < j loop
   pragma Loop_Invariant (Multiset_Unchanged (A, A'Old));
   pragma Loop_Invariant (A (A'First) = A'Old (A'First));
   pragma Loop_Invariant (for all P in A'First .. i - 1 => A (P) <= pivot);
   pragma Loop_Invariant (for all P in j + 1 .. A'Last => A (P) > pivot);

   -- increment i until A(i) > pivot
   -- decrement j until A(j) <= pivot
   -- swap if i < j
end loop;

-- if A(i) > pivot then decrement i     -- (make sure A(i) <= pivot)
-- swap(A, i, A'First)                  -- (put pivot in right place)
-- sort(A(A'First..i-1))                -- (sort left part)
-- sort(A(i+1..A'Last))                 -- (sort right part)
```

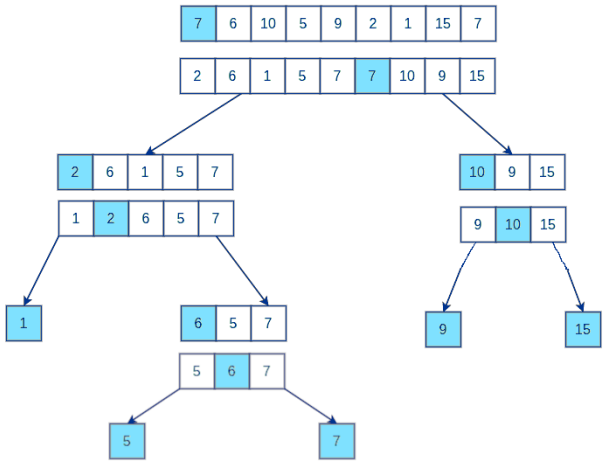

Fig. 1. Quick-sort call graph example: the blue cells are the pivots, and at each level, it shows the array before and after partitioning.

Figure 1 displays the recursive calls that this procedure makes, for an example array. Similarly to the insertion sort, we have the `Multiset_Unchanged` property as a loop invariant for the partitioning algorithm, preserved by the swap. We also have the loop invariants that to the left of `i`, there are only smaller or equal elements, and that to the right of `j`, there are only larger elements. We can prove these invariants by showing that before the swap on line 13, `A(i) > pivot and A(j) <= pivot`, and so after the swap, `A(i) <= pivot and A(j) > pivot`. We then continue the main loop in the partitioning part.

4.3 Hash Map

Flat hash maps are implemented as a set of *entries*, where each *entry* holds a single key-value pair [1]. They make dealing with hash collisions quite difficult, and so we would like to use separate chaining to handle hash collisions. With separate chaining, our hash map is a set of *buckets*, where each *bucket* is a list of *entries*.

When a key is inserted, it gets hashed, then we take the hash modulo the number of buckets, and then place the key and value in the corresponding bucket.

To retrieve the value corresponding to a key, we compute its hash, modulo the number of buckets, and look for it in the corresponding bucket.

And finally, to remove a key from the hash map, we hash the key, look it up in the corresponding bucket, and remove it from the list, by shifting all elements after it one slot forward, and finally setting the size of the bucket to the old size minus one.

We assume that our key-hashing function is pure, but there is no way to require that when instantiating a generic package in SPARK, which is how the hash map interface was designed, to make it work with all possible data types.

4.4 Publish / Subscribe channel

We have implemented a pub-sub channel, with 2 subscribers, and any number of publishers. More subscribers can easily be added, although due to how generic packages, `GNATprove` and the Jorvik profiles interact together, this couldn't be made a generic parameter. Future work could be figuring out how to make this more ergonomic, but from a formal verification point of view, 2 subscribers and 200 subscribers are not that different.

```
1  type Version is new Natural;
2  type Msg_Buffer_Index is new Natural range 1 .. Msg_Buffer_Size;
3  type Msg_Buffer_Array is array (Msg_Buffer_Index) of M;
4
5  protected type Pub_Sub_Channel is
6     procedure Publish (Message : M);
7     entry Subscribe_1 (Message : out M; Missed : out Natural);
8     entry Subscribe_2 (Message : out M; Missed : out Natural);
9  private
10    procedure Get_V
11      (Message : out M; Missed : out Natural; Sub_V : in out Version)
12    with Pre => Sub_V < Version'Last;
13    procedure Clear;
14
15    Msgs     : Msg_Buffer_Array := (others => Default_Message);
16    Subs_1_V : Version := 0;
17    Subs_2_V : Version := 0;
18    V        : Version := 0;
19 end Pub_Sub_Channel;
```

We will quickly go over the important elements:

- A `Version` refers to the version of the channel, e.g. the index of the last message put in the channel. The channel itself has a `Version` (line 18).
- The `Msgs` (line 15) is a circular buffer of messages: once it is full, we overwrite the first element, and then the second, and so on.
- Each subscriber has a `Version` (lines 16-17), which refers to the last message that the given subscriber has received.

- `Publish` will publish a message in the channel. It will increment the version of the channel, and put the message in the buffer, at the right position. Any task can call this procedure.
- Each subscriber has its own `Subscribe` entry method. An entry method allows for an entry guard, which makes sure that the code will not continue until the guard evaluates to true. The guards for the `Subscribe_1` and `Subscribe_2` methods are `Subs_1_V < V` and `Subs_2_V < V` respectively (defined in the body of the protected type). Internally, to share the logic, both functions call `Get_V(Message, Missed, Subs_1_V / Subs_2_V)`.

The implementation of `Get_V` can be described roughly as:

(1) Compute the next version from the given subscriber (`New_Sub_V := Sub_V + 1`).
(2) If it is larger than the size of the buffer, and the new version from step 1 is smaller than `V - Msg_Buffer_Size`, that means we missed some messages: Set `Missed` to `V - Msg_Buffer_Size + 1 - New_Sub_V`, and set `New_Sub_V` to `V - Msg_Buffer_Size + 1`.
(3) Get the message associated to version `New_Sub_V` in the message buffer, store it in output parameter `Message`.
(4) Set `Sub_V` to `New_Sub_V`.

Given that SPARK verifies that no two threads ever call functions on protected objects at the same time, and that we maintain the invariant that all subscribers have versions that are smaller than or equal to the version of the channel in sequential code, we can prove that they will hold for our concurrent code as well, no matter how different tasks interact with the publish / subscribe channel.

SPARK doesn't allow pre- and post-conditions involving the protected type's state variables, so proving correctness will have to happen outside of SPARK.

4.4.1 TLA⁺ for publish / subscribe channel. We make use of the following variables:

```
variables
    Q = [i \in QI |-> NullMessage]; \* the buffer of messages
    VC = 0;                         \* the version of the channel
    VS = [x \in ReceiverIds |-> 0]; \* the subscriber versions
```

We will present the code for receiving a message on a subscriber (`self`), which can be done atomically (within a single action; due to the channel being a protected type, and as such, implements mutual exclusion, this level of granularity is enough):

```
await VS[self] < VC;  \* wait for a message to be available
VS[self] := \* update the subscriber version
    IF VC > QSize /\ VS[self] < VC - QSize
    THEN VC - QSize + 1
    ELSE VS[self] + 1;
message := Q[((VS[self] - 1) % QSize) + 1]; \* get the message at the associated position in the buffer
```

The full TLA⁺ code is available in Appendix C. The full model exhaustively visits 871,132 states of which 386,111 are distinct.

5 IO-optimized asynchronous task scheduler

In subsection 5.1, we offer a short overview of the scheduler, then in subsection 5.2 we introduce the architecture of the scheduler, and then in subsection 5.3 we model this architecture in TLA⁺, to prove the properties we described in subsection 3.4.

5.1 High-level overview

Because Ada does not have language-level support for asynchronous programming yet, we defined our own version of a task, based on coroutines [3], similar to `Future` types in Rust, which are what Tokio fundamentally works with, Tokio being the asynchronous runtime we have used in part as a reference [73]. Due to this design choice, the scheduler is non-preemptive, and the scheduled computations have to collaborate with the scheduler in order to achieve peak performance; in practice this means that all futures should save their state, return control to the scheduler every so often, and will later continue when they are scheduled again.

A simplified interface containing the most important elements of the scheduler is presented here, for the full interface see Appendix D.

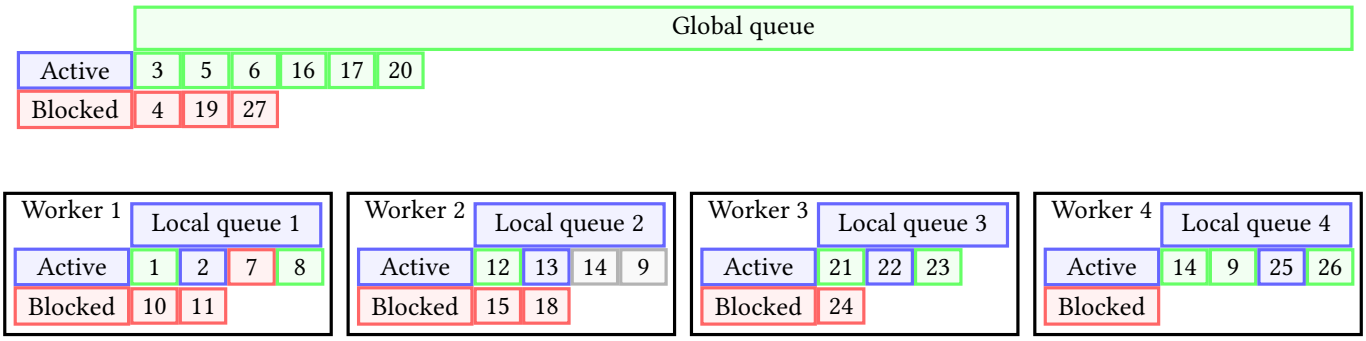

Fig. 2. High-level scheduler overview. Tasks can be either ready (green), running (blue), blocked (red), or dead (gray, only tasks 14 and 9 in Local queue 2 in the example above, as they were stolen by worker 4).

```
type Future is abstract tagged limited null record;
procedure Poll (F : in out Future; Finished : out Boolean) is abstract;
procedure Spawn (F : Future); -- Spawn a future in the runtime
procedure Spawn_RT (Root : Future); -- Spawns the runtime on the given future.
-- This procedure returns when this future completes, and all the sub-tasks it spawned also complete.
procedure Wake_In_Future (Time: Ada.Calendar.Time);
-- Tells the scheduler to not poll this Future until the specified time
procedure Wake_On_IO_Read (Socket : Socket_Access);
-- Tells the scheduler to not poll this Future until we have something to read from the given socket
procedure Wake_On_IO_Write (Socket : Socket_Access);
-- Tells the scheduler to not poll this Future until we can write to the given socket
function Get_Available_Data return Natural;
-- If the previous poll ended with Wake_On_IO_Read / Wake_On_IO_Write
-- then, this gives us the number of bytes we can read right now,
-- or the number of bytes we can write until the kernel will have to flush (and therefore block us)
```

5.2 Scheduler & IO multi-reactor architecture and implementation

Figure 2 depicts the scheduler architecture. The basic design pattern we rely on is an IO multi-reactor. Every worker has its own local queue, which consists of active tasks and blocked tasks (the tasks in the blocked queues are in the IO reactor as well). The IO reactor is an event loop-based software design pattern, which demultiplexes incoming requests / events and invokes the appropriate handlers. Modern, high-performance IO reactors make use of kernel APIs, such as `epoll` (Linux [66]), the more recent `io_uring` (Linux [67]), kqueue (BSDs, MacOS [68]), or IOCP (Windows [62]). Due to the complexity of integrating multiple APIs for different kernels into the reactors, we have decided to stick to kqueue-only. Possible future work could include extending the reactor to other kernel APIs.

The inspiration for the global queue was taken from the architecture of Tokio's multi-threaded scheduler [73]. Here we cover our architecture as well as implementation details.

5.2.1 Futures. A *future* is a basic computation. It is not blocking, but can be *polled* to advance it a single step. It is similar to coroutines [3].

5.2.2 Tasks. A *task* is an unit of work in the scheduler. It has an associated Future — the actual computation, — as well as some metadata, including its (1) state (currently running, ready, or blocked), (2) if it is blocked, what is it blocked on, and (3) if it was previously blocked on IO but when polled the kernel indicated that IO is ready, we also store how much data is available now.

5.2.3 IO reactor. Tasks can be blocked on (1) timers or (2) IO. In the case of IO, blocked tasks are handled with the use of an *IO reactor*. The IO reactors allow us to register "interests" in specific events from the kernel, and then when we receive those events, they tell us which specific tasks we should wake.

In our case, the IO reactors are implemented with `kqueue`, the kernel queues in BSD-based systems, through which we can subscribe to certain events from the kernel (such as notifications that some file descriptor – for example a socket – has data we can read); we get those notifications by actively polling (see kqueue(2) [68]). All the tasks that are blocked on IO have registered "interests" in the `kqueue` corresponding to the events they need in order to continue.

When we poll the `kqueue`, we unblock the tasks corresponding to the events we get from the kernel, and then move them from the blocked to the active queues, within the same queue structure (see subsubsection 5.2.5 and subsubsection 5.2.6). All registered events are one-shot within the `kqueue`, so they need to be registered again with a new `Wake_On_IO` operation, if the corresponding tasks need more IO.

5.2.4 Workers. A worker is an OS-level thread that executes tasks. Every worker has its own queue of tasks (subsubsection 5.2.5), as well as access to the global queue (interactions described in subsubsection 5.2.7) and the local queues for the other workers managed by the scheduler (for work stealing, see subsubsection 5.2.8). The *IO multi-reactor pattern* allows for horizontal scaling, by using multiple reactors instead of just one; in our design, every single queue has its own IO reactor (including the global queue).

5.2.5 Local task queues. Local task queues have 2 queues of tasks: Active and Blocked. Both have the same size: 256 task slots. As the names imply, the active queue contains ready tasks, and the blocked queue contains blocked tasks.

Task slots can additionally be "dead," as in, indicating that they were either finished or stolen by another worker.

The local queue's scheduling algorithm happens according to a clock-based schedule, where the "hand" of the clock moves through the list of active tasks, polling each one; for simplifying the implementation, the clock starts from the back of the queue and moves forward. Changes to task states are flagged immediately, but not processed until we finish a clock cycle. That is why we can see that task 7 is blocked in Figure 2, despite being in the list of active tasks: it was just polled, and indicated that it would block on either a timer or IO. When we finish the clock cycle, we "sweep" through the set of active tasks, remove completed / stolen tasks, move blocked tasks to the blocked queue, and if they were blocked on IO, we also add them to the interest list of the IO reactor.

Invariants:

(LQ1) Active queue: To the left of the clock hand, there are only ready tasks. (*safety*)
(LQ2) Blocked queue: Tasks in the blocked queue are always blocked. *(safety)*
(LQ3) Blocked queue: Tasks in the blocked queue, which are blocked on IO, always have interests registered in the corresponding IO reactor. *(safety)*

5.2.6 Global queue. The global queue is a queue that contains tasks that currently do not belong to any local worker (similarly to local queues, it is also made up of a queue of active tasks and a list of blocked tasks). Whenever tasks' active queues are empty, they first try to pull from the global queue (see subsubsection 5.2.7).

Invariants:

(GQ1) Active queue: Tasks in the active queue are always ready. (*safety*)
(GQ2) Blocked queue: Tasks in the blocked queue are always blocked. (*safety*)
(GQ3) Blocked queue: Tasks in the blocked queue always have corresponding registered interests in the global IO reactor. (*safety*)

5.2.7 Pushing to and pulling from the global queue. When a task's queue is full, and we attempt to push a new task (for example, if the currently running task spawns a new task), we try to "push" half of the tasks in the active queue to the global queue, to make sure that we have space for the new task. A similar operation is provided for the blocked queues.

When worker threads do not have any work to do (their active queue is empty), they try to "pull" tasks from the global queue first. When this happens, the global blocked queue gets polled first (so that all blocked tasks that are ready get moved into the global active queue). Then we pull either 128 tasks or the size of the global active queue, whichever is smaller. Unlike pushing, there is no operation to pull from the global queue of blocked tasks. "Unblocking" them happens when they are polled and we notice that they are no longer blocked, so we move them to the global active queue. This happens right before we try to pull tasks from it into the local worker queues.

There is another operation, called "*push-pull*," during which, we swap the entire active task queue of one of the local queues, with the same number of active tasks from the global queue, after some predetermined amount of clock cycles performed on the local queue (currently 10). The global queue also has its own *clock pointer*, specifically for this operation, so

that no tasks stay ready indefinitely in the global active queue. This operation is required to guarantee the size-parametric fairness.

5.2.8 Work-stealing. Work stealing happens when a worker has no ready tasks, and it cannot pull from the global queue because that has no ready tasks either. It then proceeds to look at the other workers' queues, and attempts to steal at most `Size / 2` *ready* tasks from one of them, specifically from the right side of the "clock hand," e.g. tasks that the victim worker already polled during its current clock cycle.

5.3 TLA$^+$ modelling

Here we describe the highlights of the TLA$^+$ specification of our scheduler design. It attempts to model all the details, and is an almost one-to-one mapping from the source Ada code. This gives us the highest possible assurance of our design, but due to the exhaustive search of the state space employed by TLC, exploring all possible behaviors becomes computationally unfeasible with >2 workers scheduling >6 futures.

We have first modeled the entire scheduler in a single TLA$^+$ specification, in order to check all properties except for the size-parametric fairness, which we will cover in the next paragraph. All of the other properties can be safely checked on a single-worker scheduler, given that the only interaction between multiple workers is work-stealing, but all the properties except for size-parametric fairness we have to check are preserved by it (253,286,771 states generated, of which 99,661,607 distinct, to the depth of 1172).

For the fairness, we simplified our specification to not model the kqueues explicitly, and a few other small optimizations. Another thing we had to implement was a Round-Robin token so that the specific worker processes synchronize, in order to prevent any from being starved indefinitely (we assume the OS-level scheduler handles this for us, so none of the worker threads are indefinitely starved); our size-parametric fairness property depends on this, as well as individual futures cooperating with the runtime, and individual poll operations completing within a small amount of time. This allows us to verify that all tasks will be scheduled within (Clock_Cycles_Until_Push_Pull + 1) × Num_Workers × Num_tasks ticks of being marked as ready (we define a tick as any of the workers polling a future). We have model-checked this for a model of our scheduler with a single worker thread, and it holds (checked 251,226,207 states, of which 71,774,420 distinct, stopped after diameter got to 68070; all possible interactions with the global queue preserve this property), and for a multi-worker scheduler (2 workers, same workload), we have let it run but it turned out to be computationally unfeasible (we stopped it after exploring 1,973,421,146 states, of which 443,743,574 distinct, up to a depth of 387), so we offer a proof here. The only interaction that is not checked in the single-worker scheduler is work-stealing, and given that we only ever steal tasks after they have run in the current clock cycle of the victim queue, the number of ticks until they would be scheduled again is always smaller than or equal to the number of ticks it would have taken on the single-worker version of the scheduler, and as such, we can extend our previous proof to multi-worker schedulers.

6 Implementation results and discussion

Table 2. Final line counts

Algorithm / System	Executable code	After verification (SPARK)	TLA$^+$ specification	Ratio
Insertion sort	100	838	-	8.3
Quick sort	70	966	-	13.8
Hash map	100	644	-	6.4
Pub/Sub channel	100	-	170	1.7
Scheduler	2487	-	4249 (full) / 3164 (fairness)	1.7 / 1.27

As we can see from Table 2, the full formal specifications usually have 6-10 times the lines of code of the original, non-verified versions, which implies that, on average, we need 6-10 lines of specifications for every single line of executable code in our sequential algorithms, and 1-2 lines of TLA+ specification for every line of executable code in our concurrent systems.

We have benchmarked our scheduler against tokio [73] on two different C10k scenarios [17] on CPU wall time, where in the scheduler-heavy one, our scheduler managed to outperform Tokio by a significant margin (Cohen's d = 1.71, p-value = 1.66×10^{-6}), whereas on the other, more IO-heavy benchmark tokio won by a very large margin (Cohen's d = 7.05, p-value

$= 1.1 \times 10^{-9}$). The result aligns with expectations: Mio—the IO reactor underlying Tokio—has benefited from a long production history, whereas the reactor presented in this work was implemented from the ground up within a four-week timeframe. Details about this comparison, along with the hardware and software specs can be found in Appendix E.

7 Conclusions

Formal verification in SPARK, complemented with TLA$^+$, can deliver industrial-level correctness guarantees on both sequential algorithms and realistic concurrent systems—within the scope, time, and resources of an undergraduate project. Across five case-studies we wrote 6800 lines of specification for 2800 lines of executable Ada, a $6 - 10\times$ overhead for sequential code and $\approx 1.7\times$ for the concurrent TLA$^+$ models. Performance measurements show that a formally verified IO-optimised scheduler can outperform the production-grade Tokio runtime on scheduler-heavy workloads (large effect size d = 1.71), while falling behind on IO-bound tasks (d = 7.05). These results demonstrate that correctness-by-construction need not preclude competitive performance.

7.1 Answers to the research questions

RQ1 Sequential verification. SPARK's prover fully verified insertion sort, quicksort and a generic static hash-map, exposing only minor SMT time-outs. Hence, SPARK is practical for non-trivial sequential algorithms—except where pervasive heap allocation is unavoidable.

RQ2 Concurrency. SPARK alone can establish absence of data races as well as selected safety properties, but cannot express liveness. Combining SPARK with TLA$^+$ allowed us to specify and model-check fairness and progress in a publish/subscribe channel and a work-stealing scheduler, showing the approach is tractable in practice.

7.2 Key lessons learned

Specification effort, not solver speed, is the main cost driver. Careful contract design reduced unprovable obligations far more effectively than switching SMT solvers or increasing solver timeouts.

The learning curve is real but finite. With solid programming experience, one can reach productive SPARK proficiency in weeks, not months.

Hybrid toolchains pay off. Off-loading liveness reasoning to TLA$^+$ avoided convoluted pseudo-concurrency encodings in SPARK contracts. High-fidelity models of real software systems in TLA$^+$ are still unfeasible to exhaustively explore, so a certain level of abstraction is required for TLC to finish within a reasonable amount of time.

7.3 Threats to validity

7.3.1 Model-checking scalability. TLC explored up to the depths of 68070 (1 worker) and 387 (2 workers) in the scheduler fairness model; larger parameter spaces may reveal unseen behaviours.

7.3.2 Hardware and OS bias. Benchmarks ran on a single macOS 13 system, using `kqueue`; results may differ on other kernels with different APIs, such as `epoll` or IOCP.

7.3.3 Assumption still left unproven in quick-sort. We have encountered a problem when trying to check the `Multiset_Unchanged` property on the sub-arrays after the recursive calls, despite that being the post-condition of sort, and thus already proven, the reason being SMT solver timeouts. We have left it as an assumption (with `pragma Assume(...)`). Because `Occ` is itself a recursive definition, the goal cannot be simplified any further. After running the automated provers for several hours without success, we conclude that an automated proof is probably unattainable.

7.4 Future work

Possible future work includes:

(1) Pub/Sub channel, Scheduler: Use TLAPS instead of TLC for actual formal verification of TLA$^+$ models, once it supports all required features.
(2) Scheduler: Implement `epoll`- and IOCP-based polling as well to bring it to Linux-based and Windows systems.
(3) Scheduler: Implement timeouts on `Wake_On_IO` operations.
(4) Scheduler: Optimize when there are no ready tasks, it does blocking kernel polls without a timeout, so it does not use the CPU at all.

8 Responsible Research

We considered the ethical implications of our research. Given that SPARK is widely used in high-integrity software, and Ada's motto is "when software *has* to work," using SPARK can greatly reduce the number of bugs in software. Depending on what that software does, it can have beneficial or detrimental (malicious actors writing bug free code) ethical implications. It is a tool, and therefore has to be used responsibly by its users. Nevertheless, we do not really foresee anyone using the specific case-studies presented in this paper for any malicious uses.

All source code, SPARK contracts, TLA$^+$ models and build scripts are archived in a public GitHub repository (Appendix B) under the permissive MIT license, enabling free reuse in academic and industrial settings. The full commit history is retained to document design iterations. Alire 2.1.0 (gprbuild 22.0.1, gnat_native 14.2.1), `GNATprove` 14.1.1, and TLC 2.15 were used; exact invocation flags can be found in the `alire.toml`, `.gpr` files and TLA$^+$ Toolbox models to guarantee bit-for-bit reproducibility. The TLC model logs, the benchmark source code and individual benchmark measurements, are present in the repository as well. The machine specs for the benchmark are provided in Appendix E.

No personal or sensitive data were processed; all inputs are synthetic.

8.1 ChatGPT prompts

- Rephrase the paragraph below to improve academic clarity and coherence.
- Identify and correct any grammar / spelling errors in the text provided.

A Acknowledgments

First and foremost, I would like to acknowledge my supervisor, Kobe Wullaert, for his invaluable help and guidance throughout all of the steps of the research project. I would also like to thank my responsible professor, Benedikt Ahrens, for his guidance and valuable feedback to early drafts. I would like to also thank my peer group, formed of Kaj Neumann, Jeroen Koelewijn, Mohammed Balfakeih, and Tejas Kochar, for their many, many answers to my questions, the very productive discussions and insights we have had over the course of the project, as well as early feedback on the first drafts on this paper. And of course, special thanks to my loyal rubber duck for patiently listening to countless hours of my debugging monologues.

B Repository

The full implementation of the algorithms and structures described in this paper can be found in the following repository:
https://github.com/dnbln/formalgorithms-code

C Publish / Subscribe channel TLA$^+$ model

```
------------------------------ MODULE pubsub ------------------------------
EXTENDS Naturals, Integers, TLC, Sequences

CONSTANT QSize
CONSTANT NumReceivers
CONSTANT NumMessages
CONSTANT NullMessage

ASSUME QSize >= 1
ASSUME NumMessages >= 1

QI == 1..QSize

QV == Nat
ASSUME NullMessage \notin QV
QVOpt == QV \union {NullMessage}
QT == [QI -> QVOpt]

SenderId == 0
```

```
ReceiverIds == 1..NumReceivers

MsgRange == 1..NumMessages
VersionType == 0..NumMessages

Processes == SenderId \union ReceiverIds

SeenOrMissedTy == [ReceiverIds -> [MsgRange -> BOOLEAN]]

EmptySeq == [i \in 1..0 |-> TRUE]

(* --algorithm pubsub
variables
    Q = [i \in QI |-> NullMessage];
    VC = 0;
    VS = [x \in ReceiverIds |-> 0];
    SentEnd = FALSE;
    SeenVersions = [x \in ReceiverIds |-> [m \in MsgRange |-> FALSE]];
    MissedVersions = [x \in ReceiverIds |-> [m \in MsgRange |-> FALSE]];

define
    SafetySeenOrMissed == \* safety
        \A r \in ReceiverIds:
            \A m \in MsgRange:
                SeenVersions[r][m] => ~(MissedVersions[r][m])
    EventuallySeenOrMissed == \* liveness
        \A r \in ReceiverIds:
            \A m \in MsgRange:
                <>([](SeenVersions[r][m]) \/ [](MissedVersions[r][m]))

    TypeInvariant == \* safety
        /\ Q \in QT
        /\ VC \in VersionType
        /\ SeenVersions \in SeenOrMissedTy
        /\ MissedVersions \in SeenOrMissedTy
end define;

fair+ process Sender \in {SenderId}
variables
    sent = 0;
begin
    SenderWhileLoop:
        while sent < NumMessages do
                VC := VC + 1;
                Q[((VC - 1) % QSize) + 1] := sent;
                sent := sent + 1;
        end while;
    SenderEnd:
        SentEnd := TRUE;
end process;

```

```
fair+ process Receiver \in ReceiverIds
begin
    RecvLoop:
        while ~ SentEnd \/ VS[self] < VC do
            await VS[self] < VC;
            MissedVersions[self] :=
                IF VC > QSize /\ VS[self] < VC - QSize
                THEN [x \in VS[self] + 1 .. VC - QSize |-> TRUE] @@ MissedVersions[self]
                ELSE MissedVersions[self];
            VS[self] :=
                IF VC > QSize /\ VS[self] < VC - QSize
                THEN VC - QSize + 1
                ELSE VS[self] + 1;
            SeenVersions[self][VS[self]] := TRUE;
        end while;
end process;

end algorithm; *)
\* PlusCal -> TLA+ Translation omitted
```

D Full scheduler interface

```
type Sched_Cx is limited private;
type Sched_Cx_Access is access all Sched_Cx;

type Future is abstract tagged limited null record;
procedure Poll
   (F : in out Future; Sched_Cx : Sched_Cx_Access; Finished : out Boolean)
is abstract;

type Future_Access is access all Future'Class;
procedure Spawn_RT (Root : Future_Access);

procedure Spawn (Sched_Cx : Sched_Cx_Access; F : Future_Access);

procedure Wake_In_Future
   (Sched_Cx : Sched_Cx_Access;
   Time     : Ada.Calendar.Time := Ada.Calendar.Clock);

procedure Cancel (Sched_Cx : Sched_Cx_Access);

function Get_Available_Data (Sched_Cx : Sched_Cx_Access) return Natural;
function IO_EOF (Sched_Cx : Sched_Cx_Access) return Boolean;

type Bytes is array (Positive range <>) of Interfaces.C.unsigned_char;

type File is private;
type File_Access is access all File;

function Open_Read (Path : String) return File_Access;
function Open_Write (Path : String) return File_Access;
```

```
30  function Open_Append (Path : String) return File_Access;
31
32  procedure Advise_Read_Sequencial (File : in out File_Access);
33
34  procedure Close (File : in out File_Access);
35
36  procedure Read
37     (File : File_Access; Buffer : in out Bytes; Count : out Natural)
38  with Pre => Buffer'Length >= Count;
39  -- Read Count bytes from File into Buffer.
40  -- Buffer'Length must be at least Count.
41  procedure Write (File : File_Access; Buffer : Bytes; Count : out Natural);
42
43  procedure Wake_On_IO
44     (Sched_Cx : Scheduler.Sched_Cx_Access; File : File_Access);
45
46  type Listener_Socket is limited private;
47  type Socket is limited private;
48
49  type Listener_Socket_Access is access all Listener_Socket;
50  type Socket_Access is access all Socket;
51
52  type Address_IPv4 is array (1 .. 4) of Interfaces.C.unsigned_char;
53  type Port is range 0 .. 65535;
54
55  function Connect_Socket
56    (Address : Address_IPv4; P : Port) return Listener_Socket_Access;
57  function Accept_Socket (LS : Listener_Socket_Access) return Socket_Access;
58  procedure Close_Listener (Socket : in out Listener_Socket_Access);
59  procedure Close (Socket : in out Socket_Access);
60
61  procedure Read
62    (Socket : in out Socket_Access;
63    Buffer : in out Bytes;
64    Count  : out Natural)
65  with Pre => Buffer'Length >= Count;
66  -- Read Count bytes from Socket into Buffer.
67  procedure Write
68    (Socket : in out Socket_Access; Buffer : Bytes; Count : out Natural);
69
70  procedure Mark_TCP_NoDelay (Socket : Socket_Access);
71
72  procedure Wake_On_Connection_Requested
73    (Sched_Cx : Scheduler.Sched_Cx_Access; Listener : Listener_Socket_Access);
74
75  procedure Wake_On_IO_Read
76    (Sched_Cx : Scheduler.Sched_Cx_Access; Socket : Socket_Access);
77
78  procedure Wake_On_IO_Write
79    (Sched_Cx : Scheduler.Sched_Cx_Access; Socket : Socket_Access);
```

E Scheduler comparison to tokio

We have benchmarked both our scheduler and the one in tokio [73], and Table 3 and Table 4 show the results. We observe that our scheduler outperforms Tokio's on CPU wall time on scheduler-heavy workloads (Table 3), while Tokio outperforms ours on IO-heavy workloads (fewer tasks, more IO / task, Table 4).

Implementation	n	μ	σ	95% CI	99% CI
Our scheduler	24	330.83ms	123.97ms	280.17ms - 381.50ms	264.25ms - 397.42ms
Tokio	24	731.04ms	306.97ms	605.59ms - 856.49ms	566.17ms - 895.92ms

Table 3. CPU wall time benchmark results: showing a Cohen's d coefficient of 1.71, and a p-value of 1.66×10^{-6} (10000 connections, 2 iterations of 1024 bytes)

Implementation	n	μ	σ	95% CI	99% CI
Our scheduler	10	148.90ms	2.13ms	147.51ms - 150.29ms	147.07ms - 150.73ms
Tokio	10	123.40ms	4.65ms	120.36ms - 126.44ms	119.41ms - 127.39ms

Table 4. CPU wall time benchmark results: showing a Cohen's d coefficient of 7.05, and a p-value of 1.1×10^{-9} (1000 connections, 10 iterations of 1024 bytes)

Machine specs for the benchmark: 2017 MacBook Pro, Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz, 16GB RAM, running MacOS Ventura 13.7.6.

Software versions:

```
$ alr exec -- gcc --version
gcc (GCC) 14.2.0
$ alr toolchain
gprbuild    22.0.1  Default
gnat_native 14.2.1  Default

tokio pinned to commit 99a03a502eb69309024406b07c85cabb965f53d4 (Jun 16, 2025)
$ cargo --version
cargo 1.89.0-nightly (056f5f4f3 2025-05-09)
$ rustc --version
rustc 1.89.0-nightly (ce7e97f73 2025-05-11)
```

References

[1] H. P. Luhn, "A new method of recording and searching information," *American Documentation*, vol. 4, no. 1, pp. 14–16, 1953, ISSN: 1936-6108. DOI: 10.1002/asi.5090040104. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/asi.5090040104 (visited on 05/08/2025).

[2] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, Jan. 1, 1962, ISSN: 0010-4620. DOI: 10.1093/comjnl/5.1.10. [Online]. Available: https://doi.org/10.1093/comjnl/5.1.10 (visited on 05/07/2025).

[3] M. E. Conway, "Design of a separable transition-diagram compiler," *Commun. ACM*, vol. 6, no. 7, pp. 396–408, Jul. 1, 1963, ISSN: 0001-0782. DOI: 10.1145/366663.366704. [Online]. Available: https://dl.acm.org/doi/10.1145/366663.366704 (visited on 05/29/2025).

[4] R. W. Floyd, "Assigning meanings to programs," in *Proceedings of Symposium on Applied Mathematics*, vol. 19, 1967, pp. 19–32.

[5] C. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969. DOI: 10.1145/363235.363259. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84945708698&doi=10.1145%2f363235.363259&partnerID=40&md5=456a0b6df7740cacf841e429d6c96a55.

[6] E. Ashcroft and Z. Manna, *Formalization of Properties of Parallel Programs* (Memo (Stanford artificial intelligence Laboratory)). Computer Science Department, Stanford University, 1970. [Online]. Available: https://books.google.nl/books?id=kdwsGwAACAAJ.

[7] E. A. Ashcroft, "Proving assertions about parallel programs," *Journal of Computer and System Sciences*, vol. 10, no. 1, pp. 110–135, Feb. 1, 1975, ISSN: 0022-0000. DOI: 10.1016/S0022-0000(75)80018-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000075800183 (visited on 05/29/2025).

[8] S. Owicki, "A consistent and complete deductive system for the verification of parallel programs," in *Proceedings of the eighth annual ACM symposium on Theory of computing*, ser. STOC '76, New York, NY, USA: Association for Computing Machinery, May 3, 1976, pp. 73–86, ISBN: 978-1-4503-7414-9. DOI: 10.1145/800113.803634. [Online]. Available: https://dl.acm.org/doi/10.1145/800113.803634 (visited on 05/09/2025).

[9] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/96267.96279. [Online]. Available: https://dl.acm.org/doi/10.1145/96267.96279 (visited on 04/26/2025).

[10] RTCA, Inc., *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, Standard Document, Published December 1992, 1992.

[11] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 872–923, May 1994, ISSN: 0164-0925, 1558-4593. DOI: 10.1145/177492.177726. [Online]. Available: https://dl.acm.org/doi/10.1145/177492.177726 (visited on 05/27/2025).

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995, 395 pp., ISBN: 978-0-201-63361-0.

[13] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, Aug. 25, 1996, ISSN: 0743-7315. DOI: 10.1006/jpdc.1996.0107. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731596901070 (visited on 05/28/2025).

[14] R. D. Blumofe and D. Papadopoulos, "The performance of work stealing in multiprogrammed environments (extended abstract)," in *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '98/PERFORMANCE '98, New York, NY, USA: Association for Computing Machinery, Jun. 1, 1998, pp. 266–267, ISBN: 978-0-89791-982-1. DOI: 10.1145/277851.277939. [Online]. Available: https://dl.acm.org/doi/10.1145/277851.277939 (visited on 05/29/2025).

[15] O. Goldreich, S. Goldwasser, and D. Ron, "Property testing and its connection to learning and approximation," *Journal of the ACM*, vol. 45, no. 4, pp. 653–750, Jul. 1998, ISSN: 0004-5411, 1557-735X. DOI: 10.1145/285055.285060. [Online]. Available: https://dl.acm.org/doi/10.1145/285055.285060 (visited on 04/26/2025).

[16] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, pp. 720–748, Sep. 1, 1999, ISSN: 0004-5411. DOI: 10.1145/324133.324234. [Online]. Available: https://dl.acm.org/doi/10.1145/324133.324234 (visited on 05/28/2025).

[17] D. Kegel. "The c10k problem." (1999), [Online]. Available: https://www.kegel.com/c10k.html (visited on 06/09/2025).

[18] Y. Yu, P. Manolios, and L. Lamport, "Model checking TLA+ specifications," in *Correct Hardware Design and Verification Methods*, Springer, Berlin, Heidelberg, 1999, pp. 54–66, ISBN: 978-3-540-48153-9. DOI: 10.1007/3-540-48153-2_6. [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-48153-2_6 (visited on 05/27/2025).

[19] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ACM, Sep. 2000, pp. 268–279, ISBN: 978-1-58113-202-1. DOI: 10.1145/351240.351266. [Online]. Available: https://dl.acm.org/doi/10.1145/351240.351266 (visited on 04/26/2025).

[20] D. Lea, "A java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*, ser. JAVA '00, New York, NY, USA: Association for Computing Machinery, Jun. 3, 2000, pp. 36–43, ISBN: 978-1-58113-288-5. DOI: 10.1145/337449.337465. [Online]. Available: https://dl.acm.org/doi/10.1145/337449.337465 (visited on 06/22/2025).

[21] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," *Theory of Computing Systems*, vol. 34, no. 2, pp. 115–144, Apr. 2001, ISSN: 1432-4350, 1433-0490. DOI: 10.1007/s00224-001-0004-z. [Online]. Available: http://link.springer.com/10.1007/s00224-001-0004-z (visited on 05/29/2025).

[22] L. Lamport, J. Matthews, M. Tuttle, and Y. Yu, "Specifying and verifying systems with TLA+," in *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, ser. EW 10, New York, NY, USA: Association for Computing

Machinery, Jul. 1, 2002, pp. 45–48, ISBN: 978-1-4503-7806-2. DOI: 10.1145/1133373.1133382. [Online]. Available: https://dl.acm.org/doi/10.1145/1133373.1133382 (visited on 05/27/2025).

[23] J. C. Reynolds, "Separation logic: A logic for shared mutable data structures," in *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, ser. LICS '02, USA: IEEE Computer Society, Jul. 22, 2002, pp. 55–74, ISBN: 978-0-7695-1483-3. (visited on 05/07/2025).

[24] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '05, New York, NY, USA: Association for Computing Machinery, Jul. 18, 2005, pp. 21–28, ISBN: 978-1-58113-986-0. DOI: 10.1145/1073970.1073974. [Online]. Available: https://dl.acm.org/doi/10.1145/1073970.1073974 (visited on 05/29/2025).

[25] R. Chapman, "Correctness by construction: A manifesto for high integrity software," in *Proceedings of the 10th Australian workshop on Safety critical systems and software - Volume 55*, ser. SCS '05, vol. 55, AUS: Australian Computer Society, Inc., Apr. 1, 2006, pp. 43–46, ISBN: 978-1-920682-37-8. (visited on 04/23/2025).

[26] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," presented at the Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 4963 LNCS, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-45749085681&doi=10.1007%2f978-3-540-78800-3_24&partnerID=40&md5=24529157f9dbe5e8a8c095bd3523db0e.

[27] S. Merz, "The specification language TLA+," in *Logics of Specification Languages*, D. Bjørner and M. C. Henson, Eds., Series Title: Monographs in Theoretical Computer Science. An EATCS Series, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 401–451, ISBN: 978-3-540-74106-0 978-3-540-74107-7. DOI: 10.1007/978-3-540-74107-7_8. [Online]. Available: http://link.springer.com/10.1007/978-3-540-74107-7_8 (visited on 05/30/2025).

[28] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," *SIGPLAN Not.*, vol. 44, no. 10, pp. 227–242, Oct. 25, 2009, ISSN: 0362-1340. DOI: 10.1145/1639949.1640106. [Online]. Available: https://dl.acm.org/doi/10.1145/1639949.1640106 (visited on 05/28/2025).

[29] Ž. Vrba, H. Espeland, P. Halvorsen, and C. Griwodz, "Limits of work-stealing scheduling," in *Job Scheduling Strategies for Parallel Processing*, ISSN: 1611-3349, Springer, Berlin, Heidelberg, 2009, pp. 280–299, ISBN: 978-3-642-04633-9. DOI: 10.1007/978-3-642-04633-9_15. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-04633-9_15 (visited on 05/29/2025).

[30] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz, "Verifying safety properties with the TLA + proof system," in *Automated Reasoning*, ISSN: 1611-3349, Springer, Berlin, Heidelberg, 2010, pp. 142–148, ISBN: 978-3-642-14203-1. DOI: 10.1007/978-3-642-14203-1_12. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-14203-1_12 (visited on 05/30/2025).

[31] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich, "Why3: Shepherd your herd of provers," presented at the Boogie 2011: First International Workshop on Intermediate Verification Languages, 2011, p. 53. DOI: 10/document. [Online]. Available: https://inria.hal.science/hal-00790310 (visited on 04/26/2025).

[32] J. L. Tokar PhD, F. D. Jones, P. E. Black PhD, and C. E. Dupilka, "Software vulnerabilities precluded by spark," in *Proceedings of the 2011 ACM annual international conference on Special interest group on the ada programming language*, ser. SIGAda '11, New York, NY, USA: Association for Computing Machinery, Nov. 6, 2011, pp. 39–46, ISBN: 978-1-4503-1028-4. DOI: 10.1145/2070337.2070356. [Online]. Available: https://dl.acm.org/doi/10.1145/2070337.2070356 (visited on 04/26/2025).

[33] C. Baumann, B. Beckert, H. Blasum, and T. Bormer, "Lessons learned from microkernel verification — specification is the new bottleneck," *Electronic Proceedings in Theoretical Computer Science*, vol. 102, pp. 18–32, Nov. 26, 2012, ISSN: 2075-2180. DOI: 10.4204/EPTCS.102.4. [Online]. Available: http://arxiv.org/abs/1211.6186v1 (visited on 04/23/2025).

[34] RTCA, Inc. and EUROCAE, *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*, Standard Document, RTCA DO-178C / EUROCAE ED-12C, December 2011, RTCA, Inc., 2012. [Online]. Available: https://products.rtca.org/2181fb0/.

[35] J.-C. Filliâtre and A. Paskevich, "Why3 — where programs meet provers," in *Programming Languages and Systems*, ISSN: 1611-3349, Springer, Berlin, Heidelberg, 2013, pp. 125–128, ISBN: 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_8. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-37036-6_8 (visited on 04/26/2025).

[36] B. Beckert and R. Hähnle, "Reasoning and verification: State of the art and current trends," *IEEE Intelligent Systems*, vol. 29, no. 1, pp. 20–29, Jan. 2014, ISSN: 1941-1294. DOI: 10.1109/MIS.2014.3. [Online]. Available: https://ieeexplore.ieee.org/document/6730832/ (visited on 04/23/2025).

[37] H. Ribic and Y. D. Liu, "Energy-efficient work-stealing language runtimes," *SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 513–528, Feb. 24, 2014, ISSN: 0163-5964. DOI: 10.1145/2654822.2541971. [Online]. Available: https://dl.acm.org/doi/10.1145/2654822.2541971 (visited on 05/29/2025).

[38] D. Hoang, Y. Moy, A. Wallenburg, and R. Chapman, "SPARK 2014 and GNATprove," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 6, pp. 695–707, Nov. 1, 2015, Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 6 Publisher: Springer Berlin Heidelberg, ISSN: 1433-2787. DOI: 10.1007/s10009-014-0322-5. [Online]. Available: https://link.springer.com/article/10.1007/s10009-014-0322-5 (visited on 04/25/2025).

[39] N. Matsakis. "Rayon: Data parallelism in rust · baby steps." (Dec. 18, 2015), [Online]. Available: https://smallcultfollowing.com/babysteps/blog/2015/12/18/rayon-data-parallelism-in-rust/ (visited on 05/29/2025).

[40] S. Conchon, A. Coquereau, M. Iguernlala, and A. Mebsout, "Alt-ergo 2.2," presented at the SMT Workshop: International Workshop on Satisfiability Modulo Theories, Jul. 12, 2018. [Online]. Available: https://inria.hal.science/hal-01960203 (visited on 04/26/2025).

[41] International Organization for Standardization, *Road vehicles – functional safety*, ISO 26262-1:2018, Geneva, 2018. [Online]. Available: https://www.iso.org/standard/68383.html.

[42] L. Creuse, J. Huguet, C. Garion, and J. Hugues, "SPARK by example: An introduction to formal verification through the standard c++ library," *Ada Lett.*, vol. 38, no. 2, pp. 89–96, Dec. 6, 2019, ISSN: 1094-3641. DOI: 10.1145/3375408.3375415. [Online]. Available: https://dl.acm.org/doi/10.1145/3375408.3375415 (visited on 06/04/2025).

[43] B. Marre, B. Blanc, P. Mouy, Z. Chihani, F. Vedrine, and F. Bobot, *COLIBRI*, Jun. 2, 2019. [Online]. Available: http://smt2019.galois.com/papers/tool_paper_3.pdf (visited on 04/26/2025).

[44] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," presented at the 14th USENIX Workshop on Offensive Technologies (WOOT 20), 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi (visited on 04/26/2025).

[45] G.-A. Jaloyan, C. Dross, M. Maalej, Y. Moy, and A. Paskevich, "Verification of programs with pointers in SPARK," in *Formal Methods and Software Engineering*, S.-W. Lin, Z. Hou, and B. Mahony, Eds., vol. 12531, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2020, pp. 55–72, ISBN: 978-3-030-63405-6 978-3-030-63406-3. DOI: 10.1007/978-3-030-63406-3_4. [Online]. Available: http://link.springer.com/10.1007/978-3-030-63406-3_4 (visited on 06/04/2025).

[46] H. Barbosa *et al.*, "Cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ISSN: 1611-3349, Springer, Cham, 2022, pp. 415–442, ISBN: 978-3-030-99524-9. DOI: 10.1007/978-3-030-99524-9_24. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-99524-9_24 (visited on 04/26/2025).

[47] S. T. Taft, "A work-stealing scheduler for ada 2022, in ada," *Ada Lett.*, vol. 42, no. 1, p. 80, Dec. 19, 2022, ISSN: 1094-3641. DOI: 10.1145/3577949.3577964. [Online]. Available: https://dl.acm.org/doi/10.1145/3577949.3577964 (visited on 05/29/2025).

[48] CENELEC, *EN 50716: Railway Applications - Requirements for software development*, European Standard, Published by CENELEC, 2023. [Online]. Available: https://www.nen.nl/en/nen-en-50716-2023-en-318105.

[49] A. Gannous, "Interstices in the certification of safety critical avionics software: Boeing 737-MAX MCAS case study," in *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)*, Las Vegas, NV, USA: IEEE, Jul. 24, 2023, pp. 2664–2667, ISBN: 979-8-3503-2759-5. DOI: 10.1109/CSCE60160.2023.00425. [Online]. Available: https://ieeexplore.ieee.org/document/10487505/ (visited on 04/26/2025).

[50] G. Kielanski and B. Van Houdt, "Performance analysis of work stealing strategies in large-scale multithreaded computing," *ACM Trans. Model. Comput. Simul.*, vol. 33, no. 4, 15:1–15:23, Oct. 26, 2023, ISSN: 1049-3301. DOI: 10.1145/3584186. [Online]. Available: https://dl.acm.org/doi/10.1145/3584186 (visited on 05/29/2025).

[51] *Ariane flight v88*, in *Wikipedia*, Page Version ID: 1241178269, Aug. 19, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Ariane_flight_V88&oldid=1241178269 (visited on 04/24/2025).

[52] C. B. Jones, "Three early formal approaches to the verification of concurrent programs," *Minds and Machines*, vol. 34, no. 1, pp. 73–92, Feb. 1, 2024, Company: Springer Distributor: Springer Institution: Springer Label: Springer Number: 1 Publisher: Springer Netherlands, ISSN: 1572-8641. DOI: 10.1007/s11023-023-09621-5. [Online]. Available: https://link.springer.com/article/10.1007/s11023-023-09621-5 (visited on 04/26/2025).

[53] C. Dross, "Containers for specification in SPARK," *International Journal on Software Tools for Technology Transfer*, pp. 1–13, May 27, 2025, Company: Springer Distributor: Springer Institution: Springer Label: Springer Publisher: Springer Berlin Heidelberg, ISSN: 1433-2787. DOI: 10.1007/s10009-025-00789-y. [Online]. Available: https://link.springer.com/article/10.1007/s10009-025-00789-y (visited on 06/04/2025).
[54] C. Garion, *Tofgarion/spark-by-example*, original-date: 2018-05-09T07:59:36Z, May 26, 2025. [Online]. Available: https://github.com/tofgarion/spark-by-example (visited on 05/31/2025).
[55] *Rayon-rs/rayon*, original-date: 2014-10-02T15:38:05Z, May 29, 2025. [Online]. Available: https://github.com/rayon-rs/rayon (visited on 05/29/2025).
[56] ""learn.adacore.com"," learn.adacore.com. (), [Online]. Available: https://learn.adacore.com (visited on 06/04/2025).
[57] "5.11. SPARK libraries — SPARK user's guide 26.0w." (), [Online]. Available: https://docs.adacore.com/spark2014-docs/html/ug/en/source/spark_libraries.html (visited on 06/04/2025).
[58] "7.8. how to investigate unproved checks — SPARK user's guide 26.0w." (), [Online]. Available: https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/source/how_to_investigate_unproved_checks.html#investigating-unprovable-properties (visited on 05/28/2025).
[59] "8. applying SPARK in practice — SPARK user's guide 26.0w." (), [Online]. Available: https://docs.adacore.com/live/wave/spark2014/html/spark2014_ug/en/usage_scenarios.html (visited on 05/09/2025).
[60] "Akka.dispatch.dispatchers." (), [Online]. Available: https://doc.akka.io/api/akka/1.2/akka/dispatch/Dispatchers$.html (visited on 05/29/2025).
[61] "Alternative provers — SPARK user's guide 26.0w." (), [Online]. Available: https://docs.adacore.com/spark2014-docs/html/ug/en/appendix/alternative_provers.html (visited on 04/26/2025).
[62] alvinashcraft. "I/o completion ports - win32 apps." (), [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/fileio/i-o-completion-ports (visited on 06/16/2025).
[63] "An introduction to jorvik, the new tasking profile in ada 2022," The AdaCore Blog. (), [Online]. Available: https://blog.adacore.com/introduction-to-jorvik (visited on 04/26/2025).
[64] "Better embedded system SW: Automotive software defects." (), [Online]. Available: https://betterembsw.blogspot.com/p/potentially-deadly-automotive-software.html (visited on 04/24/2025).
[65] N. Deshpande, E. Sponsler, and N. Weiss, "Analysis of the go runtime scheduler,"
[66] "Epoll(7) - linux manual page." (), [Online]. Available: https://man7.org/linux/man-pages/man7/epoll.7.html (visited on 06/10/2025).
[67] "Io_uring(7) - linux manual page." (), [Online]. Available: https://man7.org/linux/man-pages/man7/io_uring.7.html (visited on 06/19/2025).
[68] "Kevent." (), [Online]. Available: https://man.freebsd.org/cgi/man.cgi?kevent (visited on 06/10/2025).
[69] L. Lamport, "Specifying concurrent systems with TLA+,"
[70] "Netpoll - the go programming language." (), [Online]. Available: https://go.dev/src/runtime/netpoll.go (visited on 06/19/2025).
[71] "Parasail/lwt at main · parasail-lang/parasail," GitHub. (), [Online]. Available: https://github.com/parasail-lang/parasail/tree/main/lwt (visited on 05/29/2025).
[72] "TLA+ proof system." (), [Online]. Available: https://proofs.tlapl.us/doc/web/content/Home.html (visited on 05/30/2025).
[73] "Tokio::runtime - rust." (), [Online]. Available: https://docs.rs/tokio/1.45.1/tokio/runtime/index.html#multi-threaded-runtime-behavior-at-the-time-of-writing (visited on 05/28/2025).