How to train your Z3: A SPARK journey to automated QA

Introduction

Critical software fails rarely — when it does, people notice. Formal verification offers mathematical guarantees but is reputedly "too heavyweight."
Goal: Evaluate whether SPARK (Ada's provable subset) can deliver industrial-strength guarantees for both sequential algorithms and concurrent systems.

Objectives

Understand the SPARK approach to software engineering:

- RQ1 Can SPARK verify non-trivial sequential algorithms and data structures?
- RQ2 Can SPARK verify concurrent systems, or must we augment it?

But how does it *actually* work?

Formal verification in SPARK is done through a compiler backend called `GNATprove`. It takes in the annotated source code and outputs it in an intermediate form, for the **Why3** platform.

The Why3 platform then:

- generates verification conditions (VC) from the code
- turns VCs into problem instances in Satisfiability Modulo Theory (SMT)
- invokes one or multiple SMT solvers to attempt to prove the correctness of the programs, by finding proofs for the verification conditions (supported solvers include CVC5, Z3, Alt-Ergo, COLIBRI)

In SPARK, for example, a couple of the ways to create assertions are:

```
pragma Assert(...)
pragma Loop_Invariant(...)
with Pre => ..., Post => ...
```

Pre- and post-conditions, and loop invariants have to be given explicitly. Asserts are not always necessary, but they are a useful tool in debugging solver failures, and can help the solver find proofs for more complex statements much faster, if they already know the building blocks to be true.

Methods

To answer the questions, a few case studies have been implemented in Ada/SPARK:

Sequential:

- Insertion sort
- Quicksort
- HashMap

Concurrent:

- Pub/Sub channel
- IO-optimized runtime task scheduler.

Results

	Ada	SPARK	TLA+	Ratio
Insertion sort	100	838		8.3
QuickSort	70	966		13.8
Hash Map	100	644		6.4
Pub/Sub channel	100		170	1.7
Task scheduler	2487		4249	1.7

Important Results

Writing correct-by-construction code is difficult, but more than worth-while in the long run.

It helps catch a lot of bugs at verification time, as opposed to run time, through a very extensive testing effort. The down-side is that it demands a significantly higher upfront investment, but this investment usually pays off during both the testing and maintenance phases of the SDLC.

SPARK cannot fully prove functional correctness of concurrent code, a model checker like TLC is required for this. SPARK is only able to prove the absence of data races, thanks to a few specific language constructs. It is not enough to fully model a concurrent system.

Additionally, we offer an IO-optimized task scheduler for Ada, the first to our knowledge capable of tackling the C10k problem, which we have also (partially) formally verified, and showed that it has a size-parametric fairness property, given a fair underlying OS scheduler.

Conclusion

- RQ1: SPARK proves full functional correctness for algorithms except when intensive heap use is unavoidable.
- RQ2: SPARK alone covers some safety properties; combining with TLA covers all safety and liveness properties — tractable in practice.
- Cost driver: writing good contracts > solver speed.
- Hybrid stacks pay off: delegating liveness to TLA kept specs readable.

Additional Information

Code available at `https://github.com/dnbln/formalgorithms-code`.
Contact Email: D.Blanovschi@student.tudelft.nl

TUDelft

Ada VERIFIED

Extra: A bit about Hoare logic

Hoare logic is a formal system for rigorous reasoning about the **correctness** of software. It forms the basis of formal verification.

The building block of this logic is a **Hoare triple**, noted down as $\{P\}C\{Q\}$, where P and Q are assertions and C is a command. It means that if assertion P holds, and we run command C, then assertion Q now holds. All sequential algorithms can be reduced to a list of commands, but assertions P and Q are needed for each command C to form Hoare triples. These assertions typically come from annotations in the source code.

Hoare logic has been extended to deal with more complex systems:

- Separation logic, for complex heap structures
- Probabilistic Hoare logic for randomized algorithms
- Owicki-Gries logic for concurrent systems

Extra: Task interleavings

2 threads running statements [S1; S2] and [S3; S4] respectively, in parallel, is the same as non-deterministically choosing between any of the following possible behaviors:

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

Extra: TCP echo server – scheduler benchmarks against Tokio

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

Table 1: 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, **scheduler-heavy**)

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

Table 2: 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, **IO-heavy**)