SPARK as a Design Lens: Beyond Verification in High-Assurance Systems

Jacob Seman

Abstract

SPARK is typically discussed as a verification tool, a way to prove that code satisfies its specification. This paper argues that verification carries a significant secondary benefit, particularly for small infrastructure projects: architectural clarity. Writing SPARK contracts forces the engineer to define boundaries, assumptions, and scope before implementation begins. The act of specifying preconditions and postconditions serves as cognitive scaffolding that shapes design, limits complexity, and produces more maintainable systems. We illustrate this with `hadlink`, a URL redirection service where SPARK contracts guided architectural decisions from project inception.

1 Introduction

The conventional view of formal verification treats it as a post-hoc correctness check. You write the code, then you write the specification, then the prover tells you whether they match. This framing positions verification as an auditing step, valuable but separate from design. SPARK supports this workflow, but it also supports something else: using contracts as a design-time discipline that shapes architecture as a side effect.

When you write a SPARK contract before writing the implementation, you are forced to answer questions that might otherwise remain implicit. What can this function assume about its inputs? What does it guarantee about its outputs? What states are unreachable? These questions have architectural consequences. A precondition that requires normalized input implies that normalization happens elsewhere. A postcondition that guarantees a security property implies that the property need not be re-checked downstream. The contract becomes a design document that the compiler enforces.

This paper explores the architectural effects of contract-first development using SPARK, drawing on `hadlink`, a URL redirection service where contracts guided design from project inception. The primary value of SPARK remains verification: machine-checked guarantees about program behavior. But the discipline of writing contracts produces cleaner architecture as a welcome side effect, and that effect is worth examining.

2 Contracts as Design Decisions

Every precondition, postcondition, and type declaration is a decision about what the system can and cannot safely assume. Consider the contract for `hadlink`'s canonicalization function:

```
function Canonicalize (Input : String) return Canonicalize_Result
with
```

```
  Pre  => Input'Length >= 1 and then
          Input'Length <= Max_URL_Length and then
          Input'First = 1 and then
          Input'Last < Integer'Last - 10,
  Post => (if Canonicalize'Result.Status = Success
           then To_String (Canonicalize'Result.URL) = Input and then
                Is_HTTP_Or_HTTPS (Canonicalize'Result.URL) and then
                Not_Private_Address (Canonicalize'Result.URL) and then
                No_Credentials (Canonicalize'Result.URL));
```

This postcondition is not just a correctness check. It is a design statement: on success, the output preserves the input exactly, and four security properties hold. Every downstream consumer can rely on these properties without re-checking them. The Haskell service layer does not validate URLs after canonicalization succeeds because the contract guarantees the properties hold, and the prover has verified the guarantee. The contract defines the trust boundary.

The same principle applies to short code generation:

```
function Make_Short_Code (URL : Valid_URL; Secret : Secret_Key)
  return Short_Code
with
  Pre  => Length (URL) >= 7,
  Post => Length (Make_Short_Code'Result) = Short_Code_Length;
```

The contract specifies fixed-length output, deterministic behavior, and no side effects. The precondition requires a `Valid_URL`, which can only be constructed through successful canonicalization; the type system enforces the call ordering. These are architectural commitments encoded as proof obligations.

3 Architecture Emerging from Constraints

SPARK constraints directly influenced `hadlink`'s architecture in several ways. Each constraint, initially encountered as a limitation, became an architectural decision that improved the system.

3.1 Opaque Types as Proof-Driven Encapsulation

`Valid_URL` and `Short_Code` are declared as private types:

```
type Valid_URL is private;
type Short_Code is private;
```

This was not a style choice. The prover cannot reason about internal representation unless construction goes through a proven function. This enforces an invariant: `Valid_URL` can only be created by successful canonicalization. The type system becomes an architectural boundary. Code that receives a `Valid_URL` knows, by construction, that the URL has passed all validation checks.

3.2 Expression Functions for Proof Transparency

Query functions are written as expression functions (single-line definitions) because they are fully expanded during proof:

```
function Is_HTTP_Or_HTTPS (URL : Valid_URL) return Boolean is
  (Has_Valid_Scheme (To_String (URL)));
```

A multi-line body function would require the prover to reason about function behavior indirectly, potentially requiring additional assumptions. The proof requirement drove the code structure toward simpler, more transparent definitions.

3.3 Ghost Lemma for Predicate Substitution

An attempt to use `Type_Invariant` was blocked by SPARK RM 7.3.2(2) [1], which restricts type invariants in certain contexts. The workaround, a ghost lemma with two documented assumes for pure function determinism, confined all trust assumptions to a single location:

```
pragma Assume (Has_Credentials (A) = Has_Credentials (B),
  "Pure function determinism: A = B implies f(A) = f(B)");
```

These are the only two assumes in the entire codebase, both in a ghost procedure that generates no runtime code. The constraint shaped the proof strategy, which shaped the architecture: assumptions are explicit, localized, and documented.

3.4 Split Binaries from Proof Boundaries

The separation into shorten and redirect binaries was reinforced by SPARK. The redirect path needs no validation logic because URLs in the database are already canonical, which means the redirect binary has zero SPARK dependency. What began as a proof boundary naturally became an architectural boundary, and ultimately a deployment boundary. The component exposed to untrusted network traffic carries the smallest possible trusted computing base.

4 Proof Results

The verification results reflect the architectural choices described above. All 137 proof obligations are discharged automatically by the CVC5 SMT solver.

Metric	Value
Total proof obligations	137
Verified automatically (CVC5)	137 (100%)
Explicit assumes	2 (ghost lemma only)
Assumes in business logic	0

Table 1: Verification summary for `hadlink`'s SPARK core.

The proof obligations break down by type: range checks (31), overflow checks (30), preconditions (29), loop invariants (18), index checks (11), postconditions (9), and other checks including assertions, length, division, and initialization (9). All checks are solved in minimal SMT steps. The proofs are neither expensive nor fragile; they are natural consequences of contracts that accurately describe the code's behavior. When contracts match implementation cleanly, the prover's job is straightforward.

5 Lessons Beyond SPARK

The contract-first mindset transfers even without SPARK. Several principles emerge from the practice that apply to any engineering context.

First, every function should declare its assumptions explicitly. Even without machine-checked contracts, documenting preconditions leads to cleaner APIs. Callers know what they must provide; implementations know what they can assume. Ambiguity at function boundaries is a source of bugs, and explicit assumptions reduce that ambiguity.

Second, defining what a system will not do is as valuable as defining what it will do. `hadlink`'s non-goals (no analytics, no user accounts, no custom aliases) are documented and enforced. This practice, common in mature engineering organizations, prevents scope creep and communicates intent. The discipline of writing non-goals mirrors the discipline of writing preconditions: both constrain the system in ways that make reasoning easier.

Third, if you cannot state a function's postcondition, you do not fully understand what it does. The act of writing postconditions forces clarity about behavior. What does this function guarantee? What can its caller rely on? These questions have answers whether or not you write them down, but writing them down reveals gaps in understanding.

Finally, minimalism follows naturally from having to prove things about your code. Every additional feature is another proof obligation, another place where the contract must be specified and verified. This cost, even when only cognitive rather than computational, biases design toward simplicity. SPARK enforces these principles via the compiler [2]. Without SPARK, the same discipline can be applied through design reviews, documentation, and property-based testing. The compiler just makes it non-optional.

6 Conclusion

SPARK serves as both guardrail and mentor. The verification output is valuable: machine-checked guarantees that code satisfies its specification provide confidence that testing alone cannot. But the design discipline that emerges from writing contracts is also significant. For small infrastructure projects, formal methods need not be overhead. When applied early, they shape architecture in ways that reduce complexity and improve maintainability.

The question for such projects is not "can we afford to verify?" but "can we afford not to think this carefully?" The contracts must be written regardless; SPARK simply ensures they are written precisely and checked mechanically. The architectural clarity that results is a welcome side effect of that precision.

Availability

`hadlink` is open source and available at `https://github.com/Jbsco/hadlink`.

References

[1] AdaCore. *SPARK Reference Manual.* `https://docs.adacore.com/spark2014-docs/html/lrm/`

[2] AdaCore and Thales. *Implementation Guidance for the Adoption of SPARK*, Release 1.2. AdaCore, 2022. `https://www.adacore.com/uploads/books/Spark-Guidance-1.2-web.pdf`