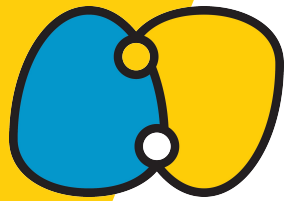
runtime verification



Optimising the symbolic execution of KEVM

Ana Pantilie

Runtime Verification Inc.

Preliminaries

runtime verification

- The K Framework: consists of the K language and various tools
  - Symbolic execution engine and formal verifier: the Haskell backend
  - Concrete execution engine: the LLVM backend
- KEVM: the semantics of EVM modeled in K
- Special thanks to:
  - Jost Berthold, Sam Balco (Haskell backend team)
  - Everett Hildenbrandt (CTO)

Overview

runtime verification

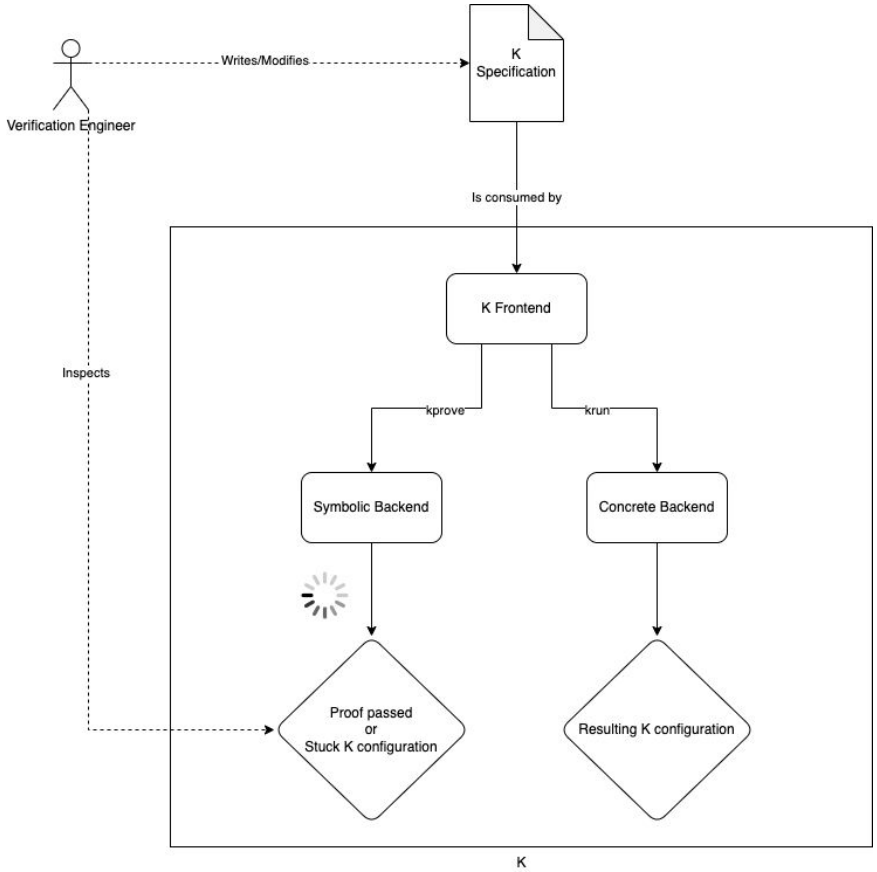
- Modern K
- First steps towards a modern symbolic backend
- A novel approach to implementing a fast symbolic execution engine

runtime verification

Modern K

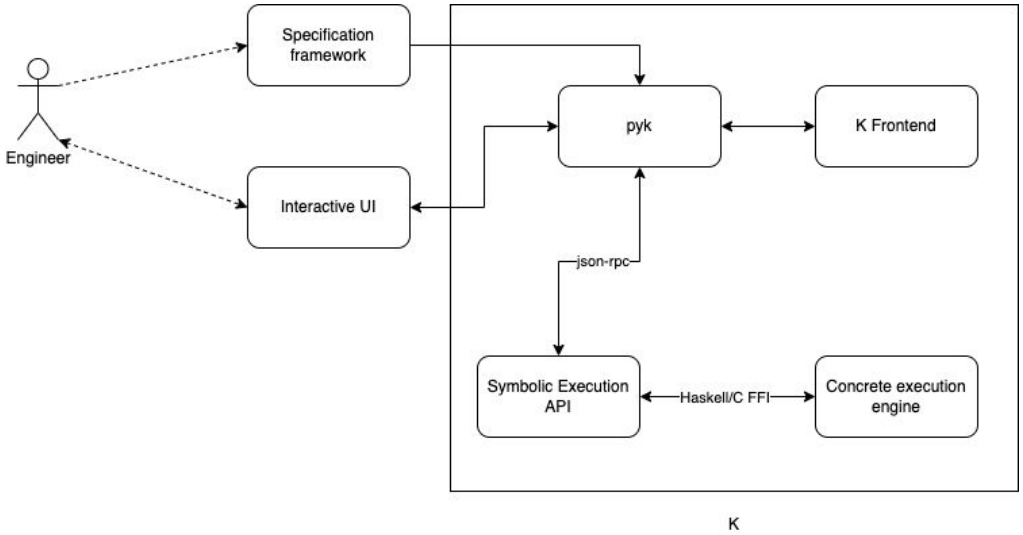
Old K workflow

- Fluency in K is required
- Very large feedback loop
- Symbolic backend is opaque to users
- The symbolic backend is too slow!



Working with modern K

- pyk: a Python package for interacting programmatically with K
- The symbolic execution API: exposes a small set of primitives required for implementing proof strategies
- Potentially no K knowledge is required for end users



Modern K projects

runtime verification

- KEVM Foundry (see the previous presentation by Andrei Vacaru)
- Work in progress: modernising KWasm, KMIR

runtime verification

A modern symbolic backend (first steps)

The Haskell backend

runtime verification

- https://github.com/runtimeverification/haskell-backend
- Designed as a matching logic interpreter (the mathematical foundation for K) with a built-in all-path reachability proving strategy
- Focus on completeness (to the detriment of performance)
- Over 150k lines of Haskell code
- Used to export two main executables: kore-exec and kore-repl
- Monolithic pipeline-like architecture, uses a text-based interface
- Limited interactivity through kore-repl, bad interoperability with kore-exec

A symbolic execution API

runtime verification

- Architectural overhaul of the Haskell backend
- A new executable, kore-rpc, which launches a server exposing the symbolic execution API over JSON-RPC
- Provides users with immense flexibility allowing for language-specific optimisations
- Exposes the three main symbolic execution primitives we have identified:
  - Execute
  - Simplify
  - Check-implication

Fast symbolic execution

A novel approach

runtime verification

- Challenge: how can we implement a fast symbolic execution engine, without losing the generality of the K approach?
- We have a big advantage now: the current symbolic execution backend
- Our philosophy:
  - Conformance testing driven development
  - Extreme programming concepts such as “you aren’t gonna need it”
  - Correctness > simplicity > completeness

The Haskell backend booster

runtime verification

- Closed-source tool which boosts regular K symbolic execution
- Built incrementally: the engine falls back to the regular symbolic backend when it can’t progress
- New focus: the needs of K instead of matching logic
- KEVM is the semantics of choice for the first version
- Risk: overfitting -> counteracted by focusing on maintaining design flexibility

Studying KEVM execution

runtime verification

- Observations:
  - A large part of the execution does not branch
  - Definedness checking is very expensive
  - We can index rewrite rules based on the main symbol of the <k> cell
  - Many rules only need a very simple, free constructor unification algorithm

Too optimistic? No problem, we can gather counterexamples and improve the implementation

Writing fast Haskell

runtime verification

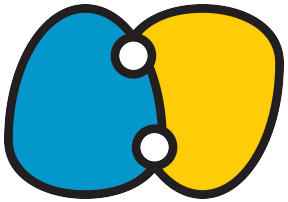
- A lot of effort spent in improving the performance of the open source backend:
  - Partnered with Serokell to identify implementation bottlenecks, check out the blog post!
- Stick to simple Haskell:
  - much easier to reason about performance
  - do not use abstractions just for the sake of them
- Call into the LLVM backend by C FFI to avoid reimplementing concrete simplification => this work led to improving LLVM backend as well
- Profile, profile, profile! Document useful kinds of profiling

Conclusion

runtime verification

Writing a fast, language independent symbolic execution engine is an open problem and we think that an experimental approach is the most practical way forward to productize K.

runtime verification



Questions?

https://runtimeverification.com/

@rv_inc

https://discord.com/invite/CurfmXNtbN

contact@runtimeverification.com