runtime verification

The K Framework

A tool kit for language semantics and proofs

jost.berthold (at runtimeverification dot com or gmail)

19 Oct, 2022, FP Syd

What is K?

- K is an *operational semantics framework* based on rewriting.
  - Specify your language or system as a K definition.
  - The K compiler derives a number of tools (parser, printer, interpreter, prover)

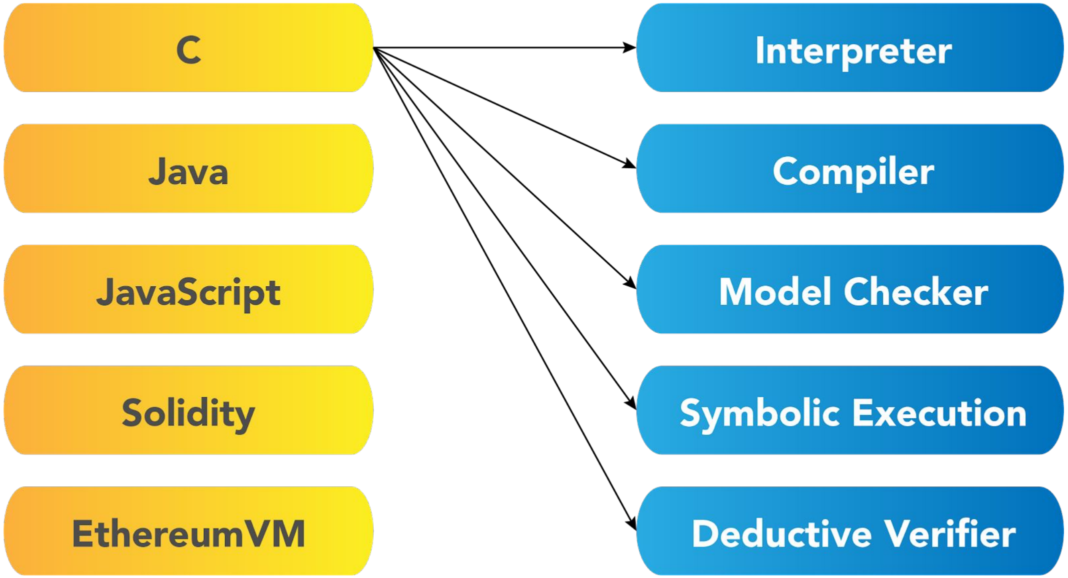
- Project started almost 20 years ago, building on earlier rewriting systems

- K's logical foundation is [Matching Logic]
  - Many-sorted first-order formalism

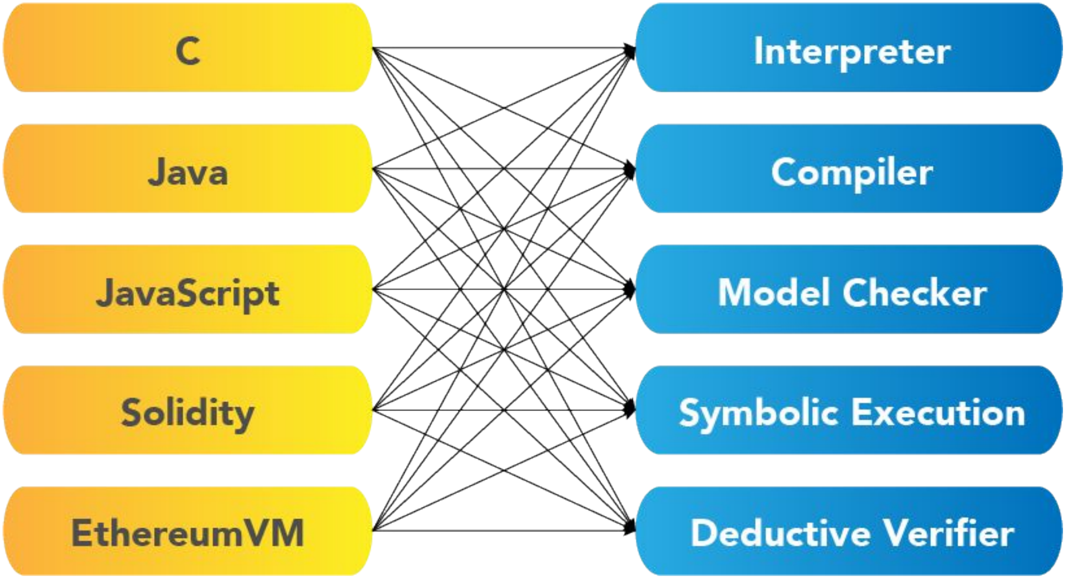
- Given a K specification, there are two main backends you can use:
  - LLVM backend is for *concrete execution*, you get a fast interpreter out of it.
  - Haskell backend is for *symbolic execution*, you get a reachability verification engine and model checker out of it.

- Webpage: **https://kframework.org**

The Problem: Too Many Tools

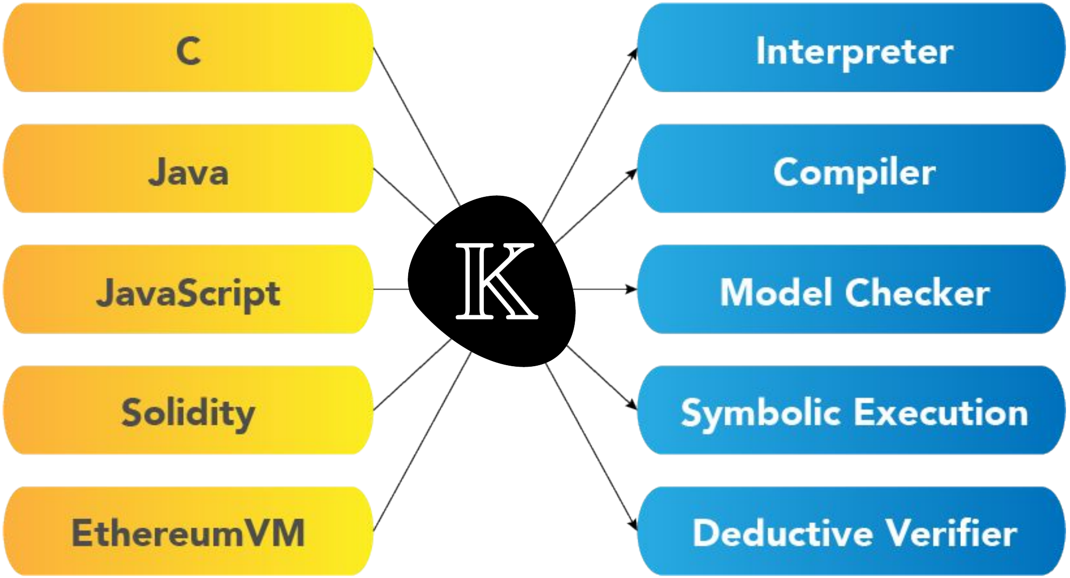


The Problem: Too Many Tools



The K Approach

- Develop each language and each tool once:



- Updates to tools benefit *all* the languages

Running Example: A small language

runtime verification

https://github.com/jberthold/k-examples-fp-syd/

- A little expression calculator,
  - Introducing basic concepts of K
- to which we will add variables
  - Extending the program configuration (state) which was automatic before
- and function calls
  - again extending the program configuration
- Finally, we turn it into a variant of IMP by adding statements
  - With only a few small adjustments, showing the framework’s flexibility
- And we can symbolically execute and prove some properties

Expression calculator

- Integer and Boolean expressions
  - Simplistic general Exp type
  - Evaluated by top-level rewriting

```
$ kompile expressions.md –backend haskell
$ krun -cPGM="2 * 3 + 4"
<k>
  10 ~> .
</k>
$ echo “2 * 3 + true” > stuck.expr
$ krun stuck.expr
<k>
  6 + true ~> .
</k>
```

(This </k> will be explained soon)

```
module EXPRESSIONS-SYNTAX
  imports INT-SYNTAX
  imports BOOL-SYNTAX

  syntax Exp ::= Int | Bool
               > left:
                 Exp “+” Exp [attrib.s] | …
               > left:
                 Exp “&&” Exp [attrib.s] | …
endmodule

module EXPRESSIONS
  imports INT
  imports EXPRESSIONS-SYNTAX

  rule <k>I:Int + I2:Int => I +Int I2 ... </k>
endmodule
```

Expression calculator

- We use built-in syntax and operations for `Int` and `Bool`
- A K *definition* is built up from a set of *modules*
  - *Main module* and *main syntax module* assumed from file name
- *Attributes* control features of the compiler to
  - generate boring code
  - refer to implementations (`+Int`)
  - provide hints to the back-end

```
module EXPRESSIONS-SYNTAX
  imports INT-SYNTAX
  imports BOOL-SYNTAX

  syntax Exp ::= Int | Bool
               > left:
                 Exp "+" Exp [attrib.s] | …
               > left:
                 Exp "&&" Exp [attrib.s] | …
endmodule

module EXPRESSIONS
  imports INT
  imports EXPRESSIONS-SYNTAX

  rule <k>I:Int + I2:Int => I +Int I2 ... </k>
endmodule
```

Argument evaluation (heat/cool)

- Built-in `+Int` requires `Int`
- We can model "suspending" the operation to evaluate the argument (termed "heating" and "cooling"
  - Needs a *continuation*
  - And additional helper constructors for each argument and operation
  - And a way to decide which rule to apply (here: Sort annotations)
- This introduces the *K cell* and its sequence of `KItem` s (`~>`)
- NB … syntax for irrelevant parts

```
module EXPRESSIONS-SYNTAX
  … syntax Exp ::= … | Exp "+" Exp | …
endmodule

module EXPRESSIONS
  imports INT
  imports EXPRESSIONS-SYNTAX

  rule <k>I:Int + I2:Int => I +Int I2 ...</k>
    [priority(50)] // default

  syntax KItem ::= freeze1(Int)
                 | freeze2(Exp)
  // heat
  rule <k> E:Int + X:Exp => X ~> freeze1(E)
    [priority(51)]
  rule <k> X:Exp + E:Exp => X ~> freeze2(E)
    [owise] // priority(200)
  // cool
  rule <k> I:Int ~> freeze1(E) => E + I ...</k>
  rule <k> I:Int ~> freeze2(E) => I + E ...</k>

endmodule
```

Argument evaluation (heat/cool)

Instead of writing all of that:

- Helper definition `KResult` to decide what sorts are evaluated

- `seqstrict` attribute
    - The compiler (`kompile`) generates code to evaluate the arguments
    - Variant `strict` generates multiple evaluation orders

```
module EXPRESSIONS-SYNTAX
  imports INT-SYNTAX
  imports BOOL-SYNTAX

  syntax Exp ::= Int | Bool
               | Exp "+" Exp [seqstrict]
               | …
               | Exp "&&" Exp [seqstrict]
endmodule

module EXPRESSIONS
  imports INT
  imports EXPRESSIONS-SYNTAX

  rule <k>I:Int + I2:Int => I +Int I2 ...</k>
  …
  syntax KResult ::= Int | Bool

endmodule
```

Adding variables

runtime verification

- Let’s add variables:
  - Built-in identifier sort
  - Let-bindings in expressions
- But we need to store the bindings somewhere
  - Adding a variable store to the program state (*configuration*)
  - Using a built-in (unsorted) `Map` (that we can match on in rules)
- We do not allow overwrites here
- Also adding a expression-if so we can write interesting programs

```
requires “../1-expressions/expressions.md”
module VARIABLES-SYNTAX
  imports EXPRESSION-SYNTAX
  imports ID-SYNTAX

  syntax Exp ::= Id
               | “let” Id “=” Exp “in” Exp [strict(2)]
               | Exp “?” Exp “:” Exp [strict(1)]
endmodule

module VARIABLES
  imports ...

  configuration <T>
                  <k> $PGM:Exp </k>
                  <store> .Map </store>
                </T>

  rule <k> X:Id => V ... </k>
       <store> ... X |-> V ... </store>

  rule <k> let X:Id = V in E => E ...</k>
       <store> M => M [ X <- V ] </store>
    requires notBool (X in_keys(M)) andBool
isKResult(V)

  rule <k> B:Bool ? E : _ => E ...</k> requires B
  rule <k> B:Bool ? _ : E => E ...</k> requires notBool
B
endmodule
```

Let's put some fun into this language!

runtime verification

- Let's add functions:
  - A program is a set of declarations
  - After processing declarations,
  - we evaluate the `main` function.
- We use built-in `List` syntax
  - .Decls is an empty list, .Map an empty map
- Needs some more components in the configuration
  - Function store (param.s, body)
  - Call stack (remembers bindings)
- We need to ensure arguments are *evaluated* using the caller's bindings
  - Here solved using a helper store

```
requires "../2-variables/variables.k"
module FUNCTIONS-SYNTAX
  imports VARIABLES-SYNTAX

  syntax Exp ::= Id "(" Args ")"
  syntax Args ::= List{Exp, ","}

  syntax Decl ::= Id "(" Params ")" "=" Body
  syntax Body ::= Exp
  syntax Params ::= List{Id, ","}

  syntax Decls ::= NeList{Decl, ""}
  syntax Id ::= "main" [token] // mentioned in a rule
endmodule

module FUNCTIONS
  imports ...

  configuration
    <T>
      <k> $PGM:Decls </k>
      <store> .Map </store>
      <args> .Map </args>
      <functions> .Map </functions>
      <call-stack> .List </call-stack>
    </T>

  // execute main afterwards
  rule <k> .Decls => main(.Args) ~> . </k>
  …
```

Finally, we add IMP statements

- The usual IMP language:
  - Integers and boolean expressions
  - Variables and assignment
  - `if` and `while` for control flow
  - `return` (we keep the functions)
- Semantics is easy:
  - Assignment replaces the `let`
  - `if` uses a built-in `#if`
  - `while` is unrolled
  - `return` throws away the existing K continuation

```
requires "../2-variables/variables.k"
module STATEMENTS-SYNTAX
  imports VARIABLES-SYNTAX
  … // most of what we had in FUNCTIONS, but:
  syntax Body ::=Stmt

  syntax Stmts ::= List{Stmt, ";"}

  syntax Stmt ::= "return" Exp                        [strict]
                | Id "=" Exp                          [strict(2)]
                | "if" "(" Exp ")" Stmt "else" Stmt [strict(1)]
                | "while" "(" Exp ")" Stmt
                | "{" Stmts "}"
endmodule

module STATEMENTS
  imports ...
  … // config and rules we had in FUNCTIONS
  rule <k> X = V => . ...  </k>
       <store> M => M [ X <- V ] </store> requires isKResult(V)

  rule <k> if (B) S1 else S2 => #if B #then S1 #else S2 #fi ...</k>

  rule <k> while ( B ) S =>
              if (B) { S ; while ( B ) S } else { .Stmts } ... </k>

  rule <k> return V ~> _ => V ~> Cont </k>
     <store> _ => M </store>
     <call-stack> ListItem(#state(M,Cont)) Rest => Rest </call-stack>
       requires isKResult(V)
```

Proving Properties

Setup for proofs:

- A verification module (compiled)
  - Imports language definition,
  - defines identifiers used in proofs,
  - May contain simplification rules
- A specification module (not compiled)
  - Imports verification module
  - Contains claims to prove
  - Any claim from the file can be applied while proving another (or the same, except in the first step)

DEMO

```
$ kompile –main-module VERIFICATION my-spec.k
$ kprove my-spec.k [--debugger]
```

```
requires “path/to/my/language/definition.k”
module VERIFICATION
  imports ... // language definition

  syntax Id ::= “f” [token]
              | “x” [token] // mentioned in claims

  rule LHS => RHS requires … [simplification]
endmodule

module MY-SPEC
 claim [aClaim]:
  <k> 1 + 2 + 3 + 4 => 10 ... </k>

 claim [another]:
  <k> f(x) = …anExpr ~> f(N) => …thing(N) … </k>
  <store> .Map => ?_ </store> // don’t care
  <args> _ </args> // unchanged, don’t care
 // rest of config unchanged (will be a problem)
    requires N <Int 42
endmodule
```

K in Practice: Runtime Verification

- Company founded in 2010

- Initially small and mostly focussed on embedded software
  - Verification in aerospace and automotive software systems
  - Research projects

  Nowadays (since ~2018) mainly auditing for blockchain software
  - Auditing smart contracts and consensus protocols
  - Formalisation of important token standards in the Eth ecosystem
- K is used for a number of verification projects by RuntimeVerification
- Several languages have been modelled in K to enable property proofs
  - Java, Python, C, Ethereum VM, Web Assembly, and more…

A larger semantics: KEVM

- Online: https://jellopaper.org
- GitHub: https://github.com/runtimeverification/evm-semantics
- K semantics of the Ethereum Virtual Machine.
  - Passes same conformance test-suite as other clients.
  - Enables symbolic execution (and thus verification) of EVM bytecode.
- Example standalone K proof (`transfer` function of an ERC20)
- Large-scale proving with K and ACT (from Multi-Collateral Dai system - 1011 proofs)

Tool Example: Opcode Summaries

- Pipeline architecture of KEVM
  - Entrypoint: https://github.com/kframework/evm-semantics/blob/master/evm.md#single-step
  - Each opcode takes ~ 10 execution steps, eg:
    - Check that stack won't over/under-flow,
    - Check that gas won't run out,
    - Check for static mode violations …
  - Pro: allows modular (and compact) definition of KEVM
  - Con: K proofs to take more execution steps overall (performance issue)

- Optimisation: (automatically) summarise opcode executions into a single K rule:
  - Taking "bigger steps" in small-steps semantics
  - Examples https://github.com/kframework/evm-semantics/blob/master/optimizations.md
  - New rules are verified to be accurate all-path summary

Resources

- K Tutorial: https://kframework.org/k-distribution/k-tutorial/
- Semantics Based Program Verifiers for All Languages: How we discharge reachability claims.
- Matching Logic: The logical formalism behind K.
- KEVM: A complete formal semantics of the EVM
- https://kframework.org
- http://matching-logic.org