runtime verification

The K Framework

A tool kit for language semantics and verification

AM Session

Introduction to K

Jin Xing Lim

21st International Symposium on Automated Technology
for Verification and Analysis (ATVA 2023)
24 October 2023

Before we start …

Make sure you install K (as it may take quite a while to install):

```
$ bash <(curl https://kframework.org/install)
$ kup install k
```

Who are we?

Runtime Verification Inc. is a software quality assurance company aimed at using formal methods to perform security audits on virtual machines and smart contracts on public blockchains.

It is dedicated to improving the safety, reliability, and correctness of software systems in the blockchain field (and other fields, too!)

Look for us!

Jin Xing Lim
@0xJinXingLim
(AM session)

Palina Tolmach
@palinatolmach
(PM session)

Overview

- **AM Session:** Introduction to K
- **PM Session:** Smart Contract Verification with KEVM

Github repository for all materials

https://github.com/runtimeverification/k-tutorial-atva-2023

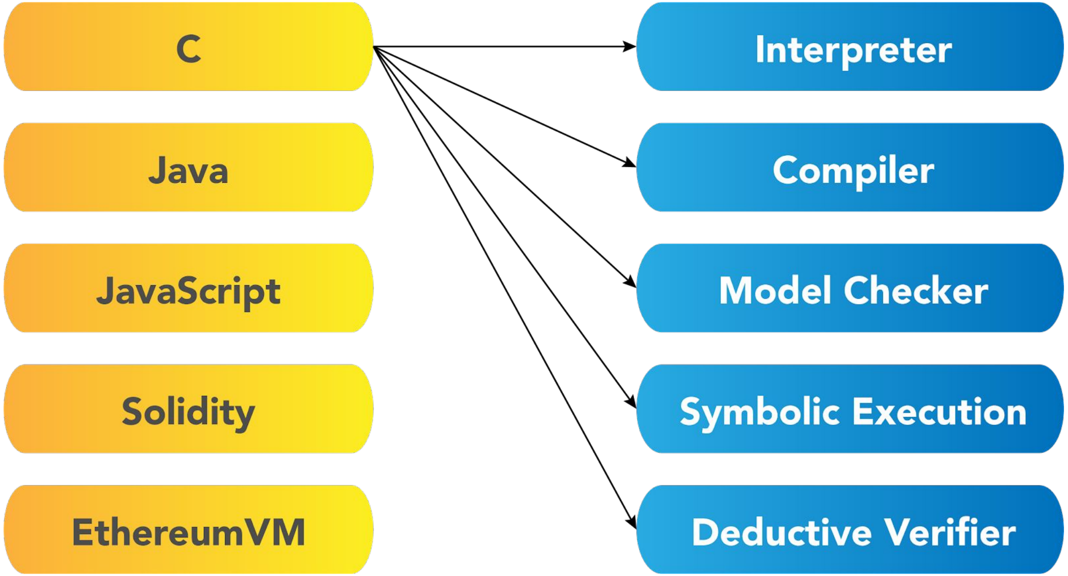
AM Session Overview

- What is K?
- K Hands-on
- K’s Logical Foundation: Matching Logic

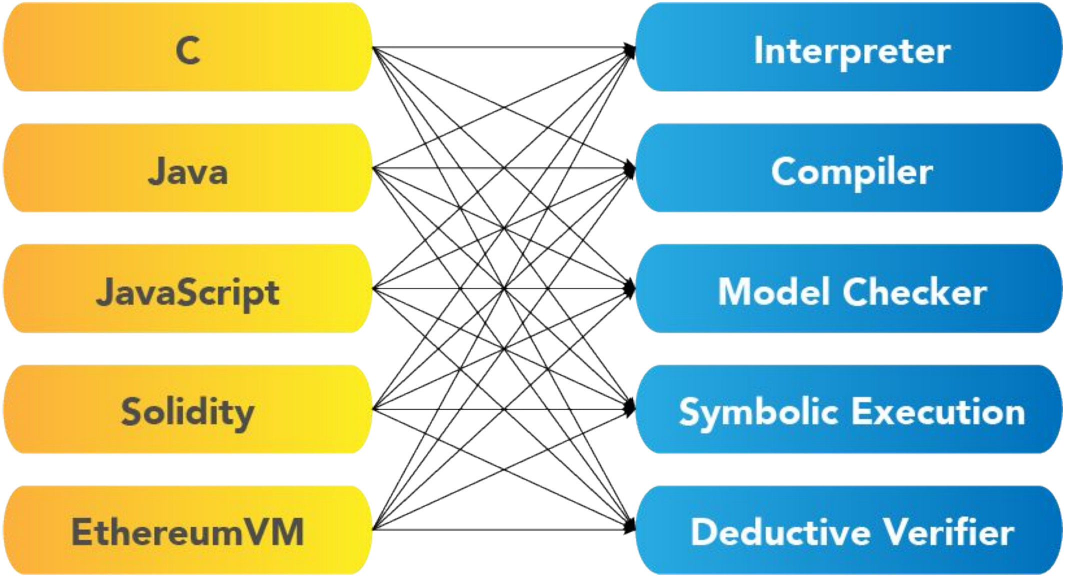
What is K?

The Problem: Too Many Tools

runtime verification

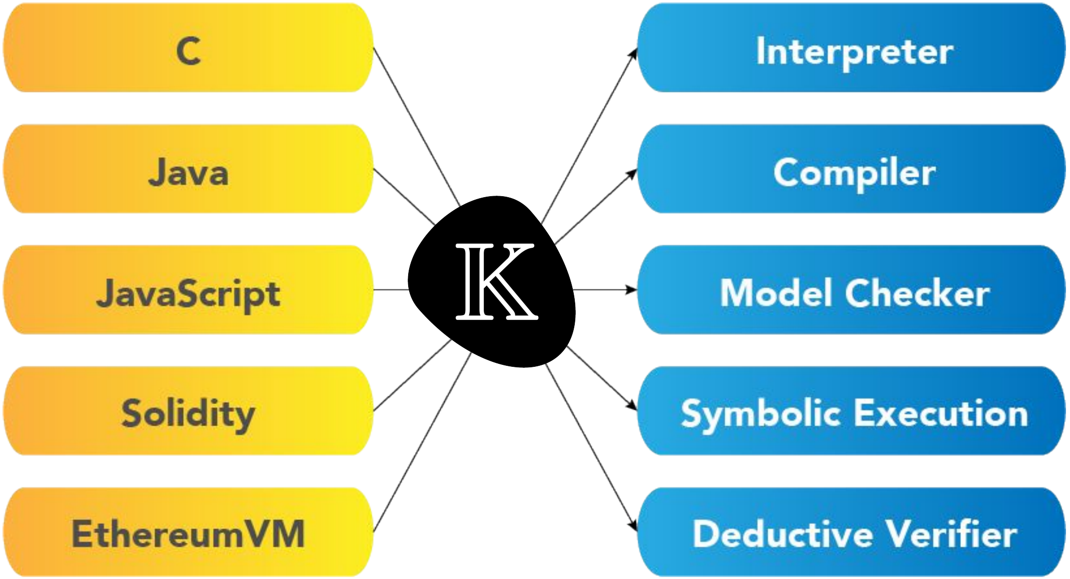


The Problem: Too Many Tools



The K Approach

- Develop each language and each tool ***once***:



- Updates to tools benefit ***all*** the languages

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**

Applications

- K implementations of:
  - C
  - Java
  - Python
  - Rust
  - Boogie
  - **Ethereum VM (PM Session)**
  - WebAssembly
  - …and more

Where to find them?

`github.com/runtimeverification`
`github.com/kframework`

K Hands-on

Step-by-step tutorial

Github repository for all materials

https://github.com/runtimeverification/k-tutorial-atva-2023

Other K materials

- K Github repository
- Do the K tutorial!
- Build programming languages in K!
- K User Manual
- K research problems

K’s Logical Foundation: Matching μ-Logic

Matching μ-Logic - Signature

Definition (Matching μ-Logic Signature):
A matching μ-logic signature is a tuple *(S, Var, Σ)* where:

- *S* is a non-empty set of sorts
- *Var* = *EVar* ∪ *SVar* is a disjoint union of two countably infinite *S*-indexed sets of sorted variables
- *Σ* is a *(S* × S)*-indexed set of countably many many-sorted symbols, such that $\Sigma = \{\Sigma_{s1,\ \ldots,\ sn,\ s}\}_{s1 \ldots sn, s \in S}$

Notations:

- $x : s$, where $x \in EVar_s$ and $s \in S$ means "*x* is an element variable of sort *s*"
- $X : s$, where $X \in SVar_s$ and $s \in S$ means "*X* is a set variable of sort *s*"

Matching μ-Logic - Signature

Examples from calc.k

Definition (Matching μ-Logic Signature):

A matching μ-logic signature is a tuple (S, Var, Σ) where:

- S is a non-empty set of sorts (e.g., $S = \{Int\}$)
- $Var = EVar \cup SVar$ is a disjoint union of two countably infinite S-indexed sets of sorted variables (e.g., $I1$, $I2$ in *rule* $I1 + I2 \Rightarrow I1 +Int\ I2$)
- Σ is a $(S^* \times S)$-indexed set of countably many many-sorted symbols, such that $\Sigma = \{\Sigma_{s1,\ \dots,\ sn,\ s}\}_{s1 \dots sn, s \in S}$ (e.g., $\Sigma = \{+_{Int\ x\ Int \to Int}, -_{Int\ x\ Int \to Int}, \dots\}$)

Notations:

- $x : s$, where $x \in EVar_s$ and $s \in S$ means "x is an element variable of sort s"
- $X : s$, where $X \in SVar_s$ and $s \in S$ means "X is a set variable of sort s"

Matching μ-Logic - Pattern

Definition (Matching μ-Logic Pattern):

A matching μ-logic pattern for a signature *(S, Var, Σ)*, is defined inductively as follows:

φ_s ::= $x : s \in EVar_s$
 | $X : s \in SVar_s$
 | $\varphi_s \wedge \varphi_s'$
 | $\neg \varphi_s$
 | $\exists x : s'.\varphi_s$
 | $\sigma(\varphi_{s1,\ \ldots,}\varphi_{sn})$ if symbol $\sigma \in \Sigma_{s1,\ \ldots,\ sn,\ s}$
 | $\mu X : s.\varphi_s$ if φ_s is positive in $X : s$ (least fixpoint*)

* least solution, under set containment, of the equation $X : s$ "=" φ_s of set variable $X : s$ (intuitively, it means finding the solution from bottom up)

Matching μ-Logic - Pattern

More pattern notations

Notations:

- $\boldsymbol{\varphi_1} \vee \boldsymbol{\varphi_2} \equiv \neg(\neg\varphi_1 \wedge \varphi_2)$
- $\boldsymbol{\varphi_1} \rightarrow \boldsymbol{\varphi_2} \equiv \neg\varphi_1 \vee \varphi_2$
- $\boldsymbol{\varphi_1} \leftrightarrow \boldsymbol{\varphi_2} \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$
- $\boldsymbol{\forall x : s.\varphi} \equiv \neg \exists x : s.\neg\varphi$
- $\top_{\mathbf{s}} \equiv \exists x : s.x$ (#Top)
- $\boldsymbol{\bot_s} \equiv \neg\top_s$ (#Bottom)
- $\boldsymbol{\nu X : s.\varphi_s} \equiv \neg\mu X : s.\neg\varphi_s[\neg X/X]$ (greatest fixpoint)

* greatest solution, under set containment, of the equation X : s “=” φ_s of set variable X : s (intuitively, it means finding the solution from top down)

Matching μ-Logic - Definedness

Definition (Definedness):
For any signature *(S, Var, Σ)* we can add a unary symbol $\lceil \_ \rceil_s^{s'} \in \Sigma_{s,s'}$ called **definedness**. We can also add the **definedness axiom**, $\lceil x : s \rceil_s^{s'}$.

Intuitively, you can think of definedness symbol as the ceiling function, which means in the semantics, $\lceil \varphi \rceil_s^{s'}$ *will be evaluate to #Top if pattern φ matches at least 1 element, i.e., a set that is non-empty.*

Matching μ-Logic - Definedness

Remark:

- Definedness allows us to syntactically construct "predicates" from general ML patterns. That is, the above symbol applications will evaluate to either ⊤ or ⊥.
- In the Haskell backend, we depend on these properties and make a strong distinction between "terms" and "predicates" for optimisation purposes.

Matching μ-Logic - Definedness

New predicates created due to definedness

Notations:

- $\lfloor \boldsymbol{\varphi} \rfloor_{s}^{s'} \equiv \neg \lceil \neg \varphi \rceil_{s}^{s'}$ (totality)
  Think of $\lfloor \_ \rfloor_{s}^{s'}$ as the floor function, i.e., dual of definedness, where the pattern φ has matched everything
- $\mathbf{x : s} \in_{s}^{s'} \boldsymbol{\varphi} \equiv \lceil x \wedge \varphi \rceil_{s}^{s'}$ (membership)
- $\boldsymbol{\varphi_1} =_{s}^{s'} \boldsymbol{\varphi_2} \equiv \lfloor \varphi_1 \leftrightarrow \varphi_2 \rfloor_{s}^{s'}$ (equality)
- $\boldsymbol{\varphi_1} \subseteq_{s}^{s'} \boldsymbol{\varphi_2} \equiv \lfloor \varphi_1 \rightarrow \varphi_2 \rfloor_{s}^{s'}$ (set containment)

Matching μ-Logic - Reachability

Notions of reachability

Definition (One-path next):
A matching μ-logic signature *(S, Var, Σ)* can be extended with an additional sort, *topConfig*, and a unary symbol • $\in \Sigma_{topConfig,topConfig}$, called **one-path next**.

Recall: We are doing proofs and rewriting in these <*configuration*> … <*configuration*>. This <*configuration*> … <*configuration*> is of sort *topConfig.*

Notation: ◦ **φ ≡** ¬ • ¬ φ **(all-path next)**

Intuition:

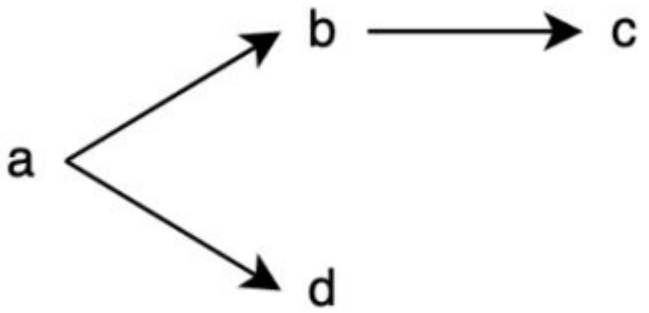
- • φ is matched by configurations that have at least one next configuration that matches φ.
- ◦ φ is matched by configurations for which all next configurations match φ.

Matching μ-Logic - Reachability

One/All-path next example

- The following is a transition system which can be formalised in matching μ-logic. Note that *a, b, c, d* are *constructors* (i.e., configurations that are distinct from each other and only matches itself).

- Question:
  1. What is the result of • b?
  2. How about the result of ◦ b?

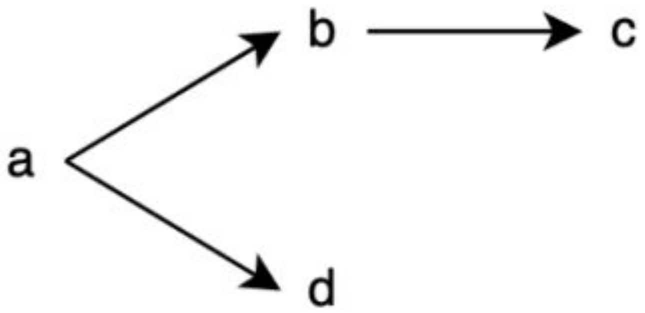


Matching μ-Logic - Reachability

One/All-path next example (solution)

- The following is a transition system which can be formalised in matching μ-logic. Note that *a, b, c, d* are *constructors* (i.e., configurations that are distinct from each other and only matches itself).

- Question:
  a. What is the result of • b?
  b. How about the result of ◦ b?



- • b = a
- ◦ b = c ⋁ d

Matching μ-Logic - Reachability

All-path reachability

Definition (All-path reachability):
We define the **all-path reachability** modality, weak always finally as:

$$\mathbf{<w>\varphi} \equiv \nu X.(\varphi \lor (\circ X \land \bullet \top))$$

Intuition:

- Either φ holds immediately (greatest fixpoint, $\nu X.\varphi$)
- Or $\nu X.(\circ X \land \bullet \top)$ ensures that we actually make steps on all paths to reach the destination.

Matching μ-Logic - Reachability

Encoding of reachability

Definition (Rewrite Rule):

A **rewrite rule** is an implication of the form:

$$\forall x_1, x_2, \ldots .\varphi(x_1, x_2, \ldots) \rightarrow \bullet \exists y_1, y_2, \ldots .\psi(x_1, ..., y_1, ...)$$

Definition (All-Path Reachability Claim):

An **all-path reachability claim** is an implication of the form:

$$\forall x_1, x_2, \ldots .\varphi(x_1, x_2, ...) \rightarrow < w > \ \exists y_1, y_2, \ldots .\psi(x_1, ..., y_1, ...)$$

control-flow.k as MμL Theory

- Sorts: *S = { KResult, Int, Bool, Id, Exp, IExp, BExp, Stmt, Block, … }*
- Variables: *Var = EVar* ∪ *SVar* is a disjoint union of two countably infinite S-indexed sets of sorted variables
- Symbols: $\Sigma = \{ \wedge_{IExp \text{ x } IExp \to IExp}, \ldots, <=_{BExp \text{ x } BExp \to BExp}, \ldots, if_{BExp \text{ x } Block \text{ x } Block \to Stmt}, \ldots \}$
- Example of rewrite rule (*rule <k> I1 + I2 => I1 +Int I2 ... </k>)*:

$$\forall I1, I2.\varphi(+(I1, I2)) \to \bullet\, \psi(+Int(I1, I2))$$

- Example of claim (last one with *while* loop in control-flow-spec.k):

$$leftTerm \wedge requires \to < w > rightPattern$$

where *leftTerm* includes the *while* loop in *<k>* cell, *S:Int, N:Int* in the *<mem>* cell and *requires* is *>=Int(N, 0)*

More on Program as MµL Theory

- Try to map procedures.k as a MµL theory

- For more advanced developers, you should look at the definition of the theory, and compare it to the *definition.kore* file generated by the K frontend, in the ...*-kompiled* directory, which the Haskell backend consumes. They are more or less the same.

More on Matching Logic

Find out more at

http://www.matching-logic.org/

Questions?

https://runtimeverification.com/

@rv_inc

https://discord.com/invite/CurfmXNtbN

contact@runtimeverification.com