runtime verification

Formal Verification in K

Ana Pantilie

May 3rd, 2022

Outline

RV and K: an overview

The K Framework

Symbolic Execution in K

Using the Haskell Backend

RV and K: an overview

Who is Runtime Verification, Inc.?

- Runtime Verification Inc. is a startup headquartered in Urbana, Illinois, USA, with staff around the world, including Bucharest, Romania.
- The company specialises in security for the blockchain and embedded domains. Our services include code audits and verification using a formal methods-based approach.
- The company is named after runtime verification as a technique for analysing programs as they execute, observing the results of the execution and using those results to find bugs.
- One of our unique technologies is the K Framework, a semantic framework for design, implementation and formal reasoning.

What is K?

- K is a framework for deriving programming languages tools from their semantic specifications.
- What is a semantic specification?

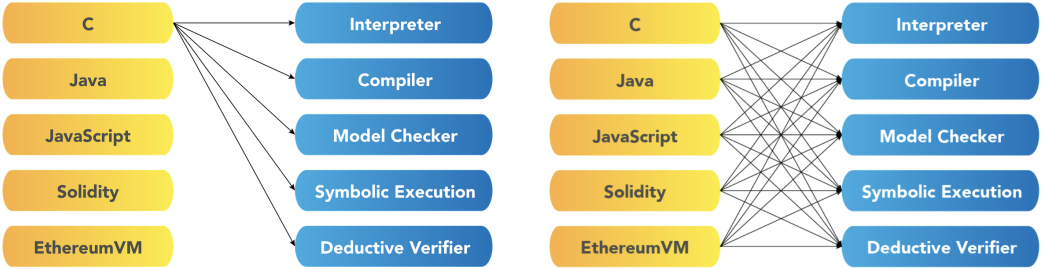
What is K?

runtime verification

- K is a framework for deriving programming languages tools from their semantic specifications.
- What is a semantic specification?
- For a programming language *L*, a semantic specification (or just semantics) is a mathematical model of the language and its behaviour.
- Specifying a semantics in K is similar to specifying the operational semantics of a programming language.
- Historically, when designing programming languages people skipped specifying their semantics: it was generally considered time consuming and useless.
- This resulted in the implementation of programming languages dictating the behaviour they should have. A direct corollary of this is that different implementations can have different behaviours.

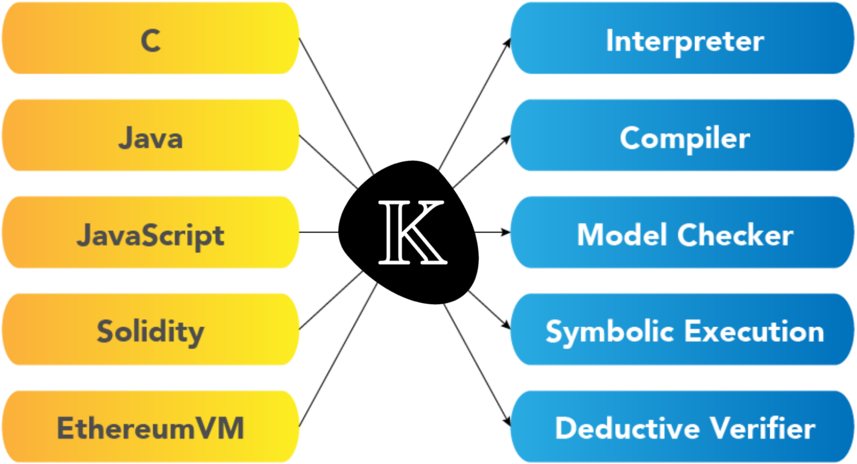
The Problem

runtime verification



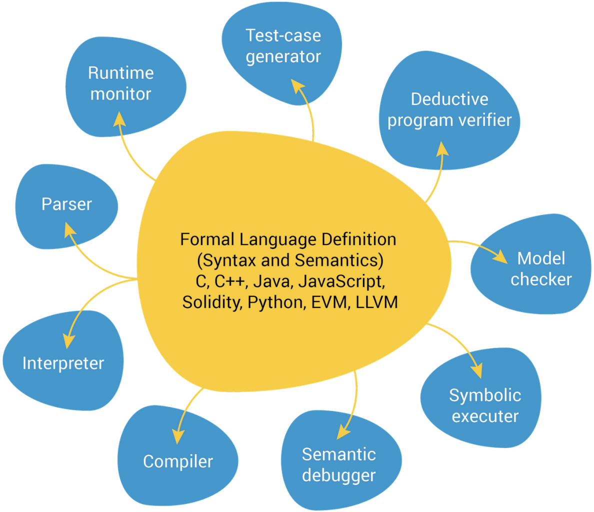
The K Solution

A language independent framework



Semantics-driven PL design

runtime verification



The K Framework

Defining Semantics in K

Three main components:

- Syntax: encodes the system primitives
- Configuration: encodes the system state
- Transition Rules: encode the system behavior

Defining Semantics in K

Defining Syntax

```
syntax AExp ::= Int | Id ...

syntax Stmt ::= Id "=" AExp ";" | ...
```

- K allows defining language syntax using Backus-Naur form
- The primitive constructs of the language inhabit user-defined sorts (categories of primitive constructs)

```
syntax BExp ::= Bool
              | BExp "&&" BExp        [left, strict(1)]
              | ...
```

- Users can specify attributes which provide parsing/execution hints to K

Defining Semantics in K

Defining Configurations

```
configuration <T>
                <k> Stmt </k>
                <state> .Map </state>
              </T>
```

- Configurations are split into cells, which describe the different components of the program state
- The contents of the cells contain syntactical constructs defined earlier

Defining Semantics in K

Defining Transition Rules

```
rule
    <k> X:Id => I ...</k>
    <state>... X |-> I ...</state>
[label(variable_lookup)]


rule
    <k> X = I:Int; => . ...</k>
    <state>... X |-> (_ => I) ...</state>
[label(variable_assignment)]
```

- Transition rules describe how the configuration (program state) evolves during execution
- We say "the LHS configuration is rewritten to the RHS configuration"
- Rules apply by unifying the LHS configuration with the configuration which describes the program being executed
- K is built for making writing these rules as efficient and maintainable as possible: "local" description of state transformation, possibility of omitting parts of the state which do not get modified etc.

Defining Semantics in K

An example (1)

(Simplified) K definition:

```
syntax AExp ::= Int | Id

syntax Stmt ::= Id "=" AExp ";"         [strict(2)]

configuration <T>
                <k> Stmt </k>
                <state> .Map </state>
              </T>

rule
    <k> X = I:Int; => . ...</k>
    <state>... X |-> (_ => I) ...</state>
[label(variable_assignment)]
```

Program:

```
n = 10;
```

Defining Semantics in K

An example (2)

Program in K:

```
<T>
  <k> n = 10; </k>
  <state> .Map </state>
</T>
```

Program state after applying the transition rule:

```
<T>
  <k> .K </k>
  <state> n |-> 10 </state>
</T>
```

The rule unifies with the configuration, meaning that it finds a way in which the LHS "overlaps" with the program. Specifically, this overlap exists when the variable X is equal to n and the variable I is equal to 10. A more technical detail is that the rest of the map inside the rule will be equal to the empty map. These associations form a substitution.

K's Program Execution Capabilities

What if we had a program like:

```
n = N:Int;
```

- Can this be executed using K?

K's Program Execution Capabilities

runtime verification

What if we had a program like:

```
n = N:Int;
```

- Can this be executed using K?
- The answer is yes! What is this kind of execution called, when the program contains elements which are not concrete?

K's Program Execution Capabilities

runtime verification

What if we had a program like:

```
n = N:Int;
```

- Can this be executed using K?
- The answer is yes! What is this kind of execution called, when the program contains elements which are not concrete?
- One of the most important features of K is that it can do symbolic execution. Configurations become templates for programs, abstracting over a set of concrete programs which match it.

K's Program Execution Capabilities

What if we had a program like:

```
n = N:Int;
```

- Can this be executed using K?
- The answer is yes! What is this kind of execution called, when the program contains elements which are not concrete?
- One of the most important features of K is that it can do symbolic execution. Configurations become templates for programs, abstracting over a set of concrete programs which match it.
- The innovative aspect here is that defining the semantics in K gives us a description of how programs execute, and using this K is able to do symbolic execution and check on all paths if execution reaches a specified state.
- This means that K can naturally do deductive verification as well!

Symbolic Execution in K

Symbolic Execution in K

- K's modern symbolic execution engine: the Haskell backend
- Developed and maintained by a dedicated team at RV
- Based on the logical formalisms for K, matching logic and reachability logic

Using the Haskell Back-end

Example: the IMP Language

Definition and program specification in K

- imp.k
- sum-spec.k

Proving with the Haskell backend

runtime verification

- First, kompile the definition for the Haskell backend:

```
$ kompile --backend haskell imp.k

```

- Then, use kprove on the specification file:

```
$ kprovex sum-spec.k

```

Proving with the Haskell backend

runtime verification

- First, kompile the definition for the Haskell backend:

```
$ kompile --backend haskell imp.k

```

- Then, use kprove on the specification file:

```
$ kprovex sum-spec.k

```

- The output we get is:

```
#Top

```

- That means the proof passed, but it doesn't give us much information about how it managed to prove the specification.

Proving with the Haskell backend

The Interactive Proof Debugger

- Thankfully, the Haskell backend has an interactive debugger mode which allows us to execute each semantic step separately and inspect the state of the proof

```
$ kprovex sum-spec.k --debugger
Welcome to the Kore Repl! Use 'help' to get started.

Kore (0)>

```

- The help command will give you a list of all the available commands in the kore-repl
- Kore = the internal language of the Haskell backend

Proving with the Haskell backend

The Interactive Proof Debugger

- Each claim is proven separately, so to start we can use the following command to see which one we are currently proving

```
  Kore (0)> claim
  You are currently proving claim 0
  \implies{SortGeneratedTopCell{}}(
    \and{SortGeneratedTopCell{}}(
    /*... requires predicate ...*/ ,
    /*... left term ...*/
    ),
    weakAlwaysFinally{SortGeneratedTopCell{}}(
    /*... right pattern ...*/
    )
)

```

- The whole output: claim0.kore

Proving with the Haskell backend

The Interactive Proof Debugger

- The previous command resulted in a big pattern in the Kore language
- Fortunately, we don’t need to understand Kore to use the interactive debugger

```
Kore (0)> load kast.kscript
Kore (0)> kclaim
You are currently proving claim 0
    /* left term */
  #And
    /* requires predicate */
#Implies
  #wAF (
   /* right pattern */
     )

```

- The whole output: claim0.k

Proving with the Haskell backend

Observations until now

- The K produced by the previous command differs from the original K claim
- This K comes from a verbatim translation of Kore
- We're closer to the internal representation of claims in the Haskell backend
- We're proving the claim with while at the top, this means we're proving the invariant
- Let's go back to the proof!

Proving with the Haskell backend

The first step

- Using the debugger, we can take a single execution step and inspect the new configuration

```
Kore (0)> step
Kore (1)> konfig
Config at node 1 is:
  <T>
    <k>
      if ( ! n <= 0 ) { { sum = sum + n ; n = n + -1 ; } while ( ! n <= 0 ) { sum = sum + n ; n =
     n + -1 ; } } else { } ~> _DotVar2 ~> .
    </k>
    <state>
      n |-> N:Int
      sum |-> S:Int
    </state>
  </T>
#And
  {
    true
  #Equals
    N >=Int 0
  }

```

Proving with the Haskell backend

The first step

- Let's look at the rule which was applied to reach this new configuration

```
Kore (1)> krule
  #Top
#And
  <T>
    <k>
      while ( RuleVarB ) RuleVarS ~> RuleVar_DotVar2 ~> .
    </k>
    RuleVar_DotVar1
  </T> => <T>
  <k>
    if ( RuleVarB ) { RuleVarS:Block while ( RuleVarB ) RuleVarS } else { } ~> RuleVar_DotVar2 ~>
      .
  </k>
  RuleVar_DotVar1
</T>
/Users/anapantilie/RV/LOSPresentation/ImpDemo/imp.k:69:10-69:55
Axiom 28

```

Proving with the Haskell backend

The second step

runtime verification

- Let's run another step to see how the if at the top starts getting rewritten

```
Kore (1)> step
Kore (2)> konfig
Config at node 2 is:
  <T>
    <k>
      ! n <= 0 ~> #freezerif(_)_else__IMP-SYNTAX_Stmt_BExp_Block_Block0_ ( { { sum = sum + n ; n =
      n + -1 ; } while ( ! n <= 0 ) { sum = sum + n ; n = n + -1 ; } } ~> . , { } ~> . ) ~>
     _DotVar2 ~> .
    </k>
    <state>
      n |-> N:Int
      sum |-> S:Int
    </state>
  </T>
#And
  {
    true
  #Equals
    N >=Int 0
  }

```

Proving with the Haskell backend

The second step

- Let's see the rule that applied

```
Kore (2)> krule
  <T>
    <k>
      if ( RuleVarHOLE ) RuleVarK1 else RuleVarK2 ~> RuleVar_DotVar2 ~> .
    </k>
    RuleVar_DotVar1
  </T>
#And
  {
    true andBool notBool isKResult ( RuleVarHOLE:BExp ~> . )
  #Equals
    true
  } => <T>
  <k>
    RuleVarHOLE:BExp ~> #freezerif(_)_else__IMP-SYNTAX_Stmt_BExp_Block_Block0_ ( RuleVarK1:Block
     ~> . , RuleVarK2:Block ~> . ) ~> RuleVar_DotVar2 ~> .
  </k>
  RuleVar_DotVar1
</T>
/Users/anapantilie/RV/LOSPresentation/ImpDemo/imp.k:18:22-19:59
Axiom 8

```

Proving with the Haskell backend

K and contextual evaluation

- We didn't actually define this rule in the semantics
- The previous rule was generated by if's strictness annotation
- What needs to be evaluated first gets pushed to the top of the K cell
- The remaining computations (which depend on these evaluations) are remembered and sequenced later in the computation list
- All the necessary rules to achieve this are generated by K's frontend, the backend doesn't do this sort of "magic"
- For this presentation, we will skip all these steps related to contextual evaluation
- If you're interested in understanding K better, I recommend studying these steps as an exercise!

Proving with the Haskell backend

Rewriting the if

runtime verification

- The steps between node 2 and node 10 take care of evaluating the if's condition

```
Kore (10)> konfig
Config at node 10 is:
  <T>
    <k>
      if ( notBool N <=Int 0 ) { { sum = sum + n ; n = n + -1 ; } while ( ! n <= 0 ) { sum = sum + n ; n = n + -1 ; } } else { } ~> _DotVar2 ~> .
    </k>
    <state>
      n |-> N:Int
      sum |-> S:Int
    </state>
  </T>
#And
  {
    true
  #Equals
    N >=Int 0
  }

```

- If we run another execution step, what do you expect will happen?

Proving with the Haskell backend

Rewriting the if

- The proof branches, and by inspecting the two new configurations we can see on which condition

```
Kore (10)> step
Stopped after 0 step(s) due to branching on [11,12]

```

- In $config_{11}$ we get $N >_{Int} 0$

```
#And
  {
    false
  #Equals
    N <=Int 0
  }

```

- In $config_{12}$ we get $N ==_{Int} 0$

```
#And
  {
    true
  #Equals
    N <=Int 0
  }

```

Proving with the Haskell backend

The "base case"

- Let's look at the configuration when $N ==_{Int} 0$

```
Kore (13)> konfig
Config at node 13 is:
  <T>
    <k>
      _DotVar2
    </k>
    <state>
      n |-> N:Int
      sum |-> S:Int
    </state>
  </T>
#And
  {
    true
  #Equals
    N <=Int 0
  }
#And
  {
    true
  #Equals
    N >=Int 0
  }

```

Proving with the Haskell backend

The "base case"

- By looking at the destination, with some simplification the two are identical
- We can expect the proof is trivial on this branch

```
Kore (13)> dest | kast -i kore -o pretty -d . /dev/stdin
Destination at node 13 is:
<T>
  <k>
    _DotVar2
  </k>
  <state>
    n |-> 0
    sum |-> S +Int ( N +Int 1 ) *Int N /Int 2
  </state>
</T>

```

- The backend will unify the configuration and the destination, and see that the condition is also satisfied

Proving with the Haskell backend

The "base case"

- Stepping again will only tell us that the proof has been finished on this branch

```
Kore (13)> step
Stopped after 0 step(s) due to reaching end of proof on current branch.

```

Proving with the Haskell backend

The "inductive case"

- Let’s go back to the branch we haven’t yet started proving, where $N >_{Int} 0$

```
Kore (11)> konfig
Config at node 11 is:
  <T>
    <k>
      { { sum = sum + n ; n = n + -1 ; } while ( ! n <= 0 ) { sum = sum + n ; n = n + -1 ; } } ~>
     _DotVar2 ~> .
    </k>
    <state>
      n |-> N:Int
      sum |-> S:Int
    </state>
  </T>
#And
  {
    false
  #Equals
    N <=Int 0
  }
#And
  {
    true
  #Equals
    N >=Int 0
  }

```

Proving with the Haskell backend

The "inductive case"

- We see that we have one *while* loop iteration to execute, for brevity we will skip till we reach the *while* again

```
Kore (11)> step 100
Kore (35)> konfig
Config at node 35 is:
  <T>
    <k>
      while ( ! n <= 0 ) { sum = sum + n ; n = n + -1 ; } ~> _DotVar2 ~> .
    </k>
    <state>
      n |-> N +Int -1
      sum |-> S +Int N
    </state>
  </T>
#And
  {
    false
  #Equals
    N <=Int 0
  }
#And
  {
    true
  #Equals
    N >=Int 0
  }

```

Proving with the Haskell backend

The "inductive case"

- Stepping now will apply the claim itself as a circularity
- It applies because the two terms unify, and the precondition is satisfied
- The unification results in the following substitution:
  $\sigma = \{N_R = N_C +_{Int} -1, S_R = S_C +_{Int} N_C\}$
- The LHS and the RHS of the claim also unify, the conditions hold therefore the claim is proven
- We'll see the resulting configuration on the next slide

Proving with the Haskell backend

The "inductive case"

```
Kore (36)> konfig
Config at node 36 is:
  <T>
    <k>
      _DotVar2
    </k>
    <state>
      n |-> 0
      sum |-> S +Int N +Int N *Int ( N +Int -1 ) /Int 2
    </state>
  </T>
#And
  {
    false
  #Equals
    N <=Int 0
  }
#And
  {
    true
  #Equals
    N +Int -1 >=Int 0
  }
#And
  {
    true
  #Equals
    N >=Int 0
  }

```

Proving with the Haskell backend

The "inductive case"

- The destination:

```
Kore (36)> dest | kast -i kore -o pretty -d . /dev/stdin
Destination at node 36 is:
<T>
  <k>
    _DotVar2
  </k>
  <state>
    n |-> 0
    sum |-> S +Int ( N +Int 1 ) *Int N /Int 2
  </state>
</T>

```

- And the last step in the claim's proof:

```
 Kore (36)> step
Stopped after 0 step(s) due to reaching end of proof on current branch.
```

Proving with the Haskell backend
The main claim

- Let's use the following command to see the whole proof status

```
Kore (36)> proof-status
Current proof status:
  claim 1: NotStarted
  claim 0: Completed

```

- We can switch to proving the second claim, which we know is our main claim as follows

```
Kore (36)> prove 1
Switched to proving claim 1
Kore (0)>

```

- Let's not study each step of this proof, but it can be useful as an exercise!

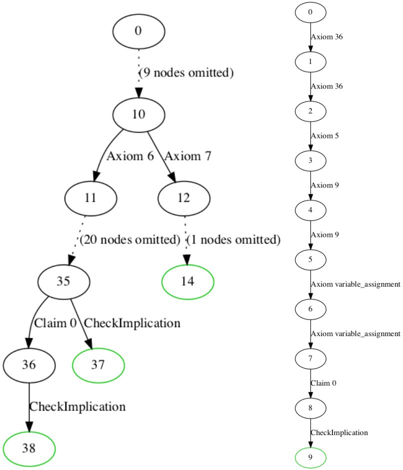
```
Kore (0)> step 100
Stopped after 8 step(s) due to reaching end of proof on current branch.
Kore (8)>

```

Proving with the Haskell backend

Visualising the proof

The graph command inside the debugger generates a graph visualisation of the proof for the currently selected claim.



Proving with the Haskell backend

Conclusions

- We've seen that K provides a general solution to defining programming languages
- K generates the necessary tooling for free, providing a semantics-based approach to PL development
- K offers symbolic execution and deductive verification based on a sound, mathematical formalism, again, for free
- We've seen how to use this verifier in practice on a toy example

Thank you!

runtime verification

References and Further Reading

References and Further Reading

1. GitHub repository for the Haskell Backend
2. GitHub repository for the K Framework
3. The K User Manual
4. The K Overview
5. Verifying Wasm Programs with K, Stephen Skeirik, 2019
6. X. Chen, G. Rosu, *Matching μ-Logic*, LICS'19 ACM/IEEE, pp 1-13, June 2019
7. T. Serbanuta, V. Serbanuta, X. Chen, *Proving All-Path Reachability Claims*

- For any questions feel free to reach out at `ana.pantilie@runtimeverification.com`!