Formal Design, Implementation and Verification of Blockchain Languages

Grigore Rosu
University of Illinois at Urbana-Champaign
President & CEO, Runtime Verification Inc.

Waterloo, 2019-10-05
Ideal Language Framework Vision

- Deductive program verifier
- Model checker
- Symbolic execution
- (semantic) Debugger
- Compiler
- Interpreter
- Parser

Formal Language Definition (Syntax and Semantics)
Current State-of-the-Art
- Sharp Contrast to Ideal Vision -

Separate tools, by separate teams, little to no code shared

C
Java
JavaScript
Solidity
Ethereum VM

Interpreter
Compiler
Model Checker
Symbolic Execution
Deductive Verifier
Current State-of-the-Art
- Sharp Contrast to Ideal Vision -


L
- C
- Java
- JavaScript
- Solidity
- Ethereum VM

T
- Interpreter
- Compiler
- Model Checker
- Symbolic Execution
- Deductive Verifier

...
How It Should Be

Ideal Language Framework

- Interpreter
- Compiler
- Model Checker
- Symbolic Execution
- Deductive Verifier

C
Java
JavaScript
Solidity
Ethereum VM

...
Our Attempt: the K Framework

http://kframework.org

• We tried various semantic styles, for >15y and >100 top-tier conference and journal papers:
  – Small-/big-step SOS; Evaluation contexts; Abstract machines (CC, CK, CEK, SECD, ...); Chemical abstract machine; Axiomatic; Continuations; Denotational;...

• But each of the above had limitations
  – Especially related to modularity, notation, verification

• K framework initially \textit{engineered}: keep advantages and avoid limitations of various semantic styles
  – Then theory came
Complete K Definition of KernelC
Complete K Definition of KernelC

Syntax declared using annotated BNF

**SYNTAX**  \[\text{Exp} ::= \cdots \mid \text{Exp} = \text{Exp} \ [\text{strict}(2)]\]
Complete K Definition of KernelC

Configuration given as a nested cell structure. Leaves can be sets, multisets, lists, maps, or syntax.
Complete K Definition of KernelC

Semantic rules given contextually

\[
\frac{X = V}{V} \quad \frac{X \rightarrow \_}{V}
\]

rule

\[
k: X = V \Rightarrow V \quad \text{env: } \_ \Rightarrow (_ \Rightarrow V)
\]
K Scales

Several large languages were recently defined in K:

- **JavaScript ES5**: by Park et al [PLDI’15]
  - Passes existing conformance test suite (2872 programs)
  - Found (confirmed) bugs in Chrome, IE, Firefox, Safari
- **Java 1.4**: by Bogdanas et al [POPL’15]
- **x86**: by Dasgupta et al [PLDI’19]
- **C11**: Ellison et al [POPL’12, PLDI’15]
  - 192 different types of undefined behavior
  - 10,000+ program tests (gcc torture tests, obfuscated C, ...)
  - Commercialized by startup (Runtime Verification, Inc.)
- **EVM** [CSF’18], **Solidity**, **IELE** [FM’19], **Plutus**, **Vyper**, ...
Ideal Language Framework Vision [K]

Formal Language Definition (Syntax and Semantics)

- Parser
- Deductive program verifier
- Interpreter
- Model checker
- Compiler
- Symbolic execution
- (semantic) Debugger

K -> LLVM -> x86
State-of-the-Art

- Redefine the language using a different semantic approach (Hoare/separation/dynamic logic)
- Language specific, non-executable, error-prone

\[
\frac{H \vdash \{ \psi \land e \neq 0 \} \land s \{ \psi \}}{H \vdash \{ \psi \} \text{ while(e) s } \{ \psi \land e = 0 \}}
\]

\[
H \cup \{ \psi \} \text{ proc() } \{ \psi' \} \vdash \{ \psi \} \text{ body } \{ \psi' \}
\]

\[
H \vdash \{ \psi \} \text{ proc() } \{ \psi' \}
\]
Ideal Scenario

Use directly the trusted language model/semantics!

Language-independent proof system
– Takes operational semantics as axioms
– Derives reachability properties
– Sound and relatively complete for all languages!
Matching μ-Logic

[... LICS’13, RTA’15, OOPSLA’16, FSCD’16, LMCS’17, LICS’19]

16 proof rules only!

Simple proof checker (200 LOC, vs Coq’s 8000)!
Expressiveness of Matching $\mu$-Logic

- Pure Type Systems
- Order-Sorted Algebras
- Many-Sorted Algebras
- Untyped $\lambda$-Calculus
- Equational Logic
- First-Order Logic
- Separation Logic
- Matching Logic
- Polyadic and/or Hybrid Modal Logic
- Propositional Logic
- Normal Modal Logic
- Dynamic Logic
- Temporal Logics (LTL, CTL, CTL*, ...)
- Modal $\mu$-Logic

- Applicative Matching Logic
- Matching $\mu$-Logic
- Reachability Logic
- Separation Logic with Recursive Definitions
- Rewriting Logic
- Hoare Logic
- K

- First-Order Logic with Least Fixpoints

Reachability Logic (Semantics of K)
[LICS’13, RTA’14, RTA’15, OOPSLA’16]

• “Rewrite” rules over matching logic patterns:

\[ \varphi \Rightarrow \varphi' \]

Can be expressed in matching logic:
\[ \varphi \rightarrow \Diamond(\varphi') \]  \[ \Diamond \] is “weak eventually”

• Patterns generalize terms, so reachability rules capture rewriting, that is, operational semantics

• Reachability rules capture Hoare triples  [FM’12]

\[ \{\text{Pre}\} \text{Code}\{\text{Post}\} \equiv \widehat{\text{Code}} \land \widehat{\text{Pre}} \Rightarrow \epsilon \land \widehat{\text{Post}} \]

• Sound & relative complete proof system
  – Now proved as matching \( \mu \)-logic theorems
K Deductive Program Verifier = (Best Effort) Automation of M\(\mu\)L

- Evaluated it with the existing semantics of C, Java, JavaScript, EVM, and several tricky programs
- Morale:
  - Performance is comparable with language-specific provers!
Sum 1+2+...+n in IMP: Main

```
rule
  <k>
    int n, sum;
    n = N:Int;
    sum = 0;
    while (!(n <= 0)) {
      sum = sum + n;
      n = n + -1;
    }
  =>
    .K
  </k>
<state>
  .Map
  =>
    n  |-> 0
    sum |-> ((N +Int 1) *Int N /Int 2)
</state>
requires N >= Int 0
```
Sum $1+2+\ldots+n$ in IMP: Invariant

```plaintext
rule
  <k>
    while !(n <= 0) {
      sum = sum + n;
      n = n + -1;
    }
  =>
  .K
...</k>
<state>...
  n  |-> (N:Int => 0)
  sum |-> (S:Int => S +Int ((N +Int 1) *Int N /Int 2))
...</state>
requires N >=Int 0
```
Properties very challenging to verify automatically. We only found one such prover for C, based on a separation logic extension of VCC—Which takes 260 sec to verify AVL insert (ours takes 280 sec; see above)
K for the Blockchain
K Framework Vision

Parser

Interpreter

Compiler

Deductive program verifier

Model checker

Symbolic execution

(Blockchain Language Definition)

(Semantic) Debugger
KEVM: Semantics of the Ethereum Virtual Machine (EVM) in K

Complete semantics of EVM in K
- https://github.com/kframework/evm-semantics
- Passes 60,000+ tests of C++ reference implementation
- 25% faster than ethereumjs, used by Truffle
- 5x (only!) slower than ethereum-cpp
- Used as canonical EVM spec (replacing Yellow Paper)
What Can We Do with KEVM?

1) **Formal documentation**: [http://jellopaper.org](http://jellopaper.org)
What Can We Do with KEVM?

2) *Generate and deploy correct-by-construction EVM client/simulator/emulator*

Firefly tool: KEVM to run, analyze and monitor tests

Cardano testnet: mantis executing KEVM
What Can We Do with KEVM?

3) **Formally verify Ethereum smart contracts**

RV does that commercially. Won first Ethereum Security grant to verify Casper, then hired to formalize Beacon Chain (Serenity) and verify ETH1 -> ETH2 deposit contract.

"Many resources are shifting into testing, fuzzing, and audits over the coming months. We engaged **Runtime Verification** to formally verify the deposit contract and to formally specify the Beacon Chain. This is in addition to considerable effort by the research, development, and security teams involved in ETH 2.0 toward reliability and security." - **Ethereum Foundation**
Formalizing ERC20, ERC777, ... in K

- *K is very expressive for modeling:* languages, but also *token specifications and protocols; executable*
- To formally verify smart contracts, we also formalized token specifications, multisigs, etc.:
  - ERC20, ERC777, many others
- All our specs are *language-independent!*  
  - i.e., not specific to Solidity, not even to EVM
- We had the *first verified ERC20 contracts!*  
  - Written both in Solidity and in Vyper, verified as EVM
- Others use or integrate our framework and specs:  
  - DappHub ([KLab](#)), ETHWorks ([Waffle](#)), Consensys, Gnosis
Transfers \( _\text{value} \) amount of tokens to address \( _\text{to} \), and MUST fire the \Transfer \ event. The function SHOULD throw if the \_from \ account balance does not have enough tokens to spend.

\textit{Note} Transfers of 0 values MUST be treated as normal transfers and fire the \Transfer \ event.

\begin{verbatim}
function transfer(address _to, uint256 _value) returns (bool success)

[transfer]
callData: #abiCallData("transfer", #address(TO_ID), #uint256(VALUE))
gas: {GASCAP} => _
refund: _ => _
requires:
  andBool 0 <=Int TO_ID    andBool TO_ID  <Int (2 ^Int 160)
  andBool 0 <=Int VALUE    andBool VALUE  <Int (2 ^Int 256)
  andBool 0 <=Int BAL_FROM andBool BAL_FROM  <Int (2 ^Int 256)
  andBool 0 <=Int BAL_TO   andBool BAL_TO   <Int (2 ^Int 256)

[transfer-success]
k: #execute => (RETURN RET_ADDR:Int 32 ~> _.localMem: .Map => ( .Map[ RET_ADDR := #asByteStackInWidth(1, 32) ] :Map )
log: _:List ( .List => ListItem(#abiEventLog(ACCT_ID, "Transfer", #indexed(#address(CALLER_ID

rule
transfer(T, V) => true
caller: F
account:
  id: F
  balance: BF => BF - V
account:
  id: T
  balance: BT => BT + V
log: . => Transfer(F,T,V)
requires
  0 <= V /
  V <= BF /
  BT + V <= MAXVALUE
\end{verbatim}
Notable Contracts We’ve Verified

- ETH2.0 Deposit
- GnosisSafe
- Ethereum Casper FFG
- Uniswap
- DappSys DSToken ERC20
- Bihu KEY token
Designing New (and Better) Blockchain Languages Using K
EVM Not Human Readable
(among other nuisances)

If it must be low-level, then I prefer this:

```assembly
define public @sum(%n) {
    %result = 0
    condition:
        %cond = cmp le %n, 0
        br %cond, after_loop
    %result = add %result, %n
    %n = sub %n, 1
    br condition
after_loop:
    ret %result
}
```
A New Virtual Machine (and Language) for the Blockchain

• Incorporates learnings from defining KEVM and from using it to verify smart contracts
• Register-based machine, like LLVM; unbounded*
• *IELE was designed and implemented using formal methods and semantics from scratch!
• Until IELE, only existing or toy languages have been given formal semantics in K
  – Not as exciting as designing new languages
  – We should use semantics as an intrinsic, active language design principle, not post-mortem

Thanks to IOHK (iohk.io) for funding this project
K Semantics of Other Blockchain Languages

- **WASM** (web assembly) – in progress, in collaboration with the Ethereum Foundation
- **Solidity** – in progress, collaboration between RV and Sun Jun’s group in Singapore
- **Vyper** – in progress, collaboration with the Ethereum Foundation
- **Plutus** (functional) – collaboration with IOHK
- **Flow** (linear types, resources) – in progress, collaboration with DapperLabs (creators of CryptoKitties); plan is have *only* a K “implementation”
Modelling and Verification of Blockchain Protocols

• Matching logic, rewriting and K can also be used to formally specify and verify consensus protocols, random number generators, etc.

• Done or ongoing:
  – Casper FFG (Ethereum Foundation)
  – RANDAO (Ethereum Foundation)
  – Algorand (Algorand)
  – Casper CBC (Coordination Technology)
  – Serenity / ETH 2.0 (Ethereum Foundation)

• Several others planned or in discussions
K Blockchain Products and Tools in the Making. To be SaaS delivered

- **Firefly** – automated smart contract analysis
- **KaaS** – K formal verification as a service
- **Proof objects** – ultimate correctness certificates
Taking K to the Next Level

- Many people use K (40+ repositories and 50,000+ commits)
  + Open source, used also for teaching PL at several universities
  + Most comprehensive and rich in features language framework
    - Hard to use and debug, poor error messages
    - Slow (may take hours to formally verify non-trivial programs)
- Two major underlying engines under development
  1. *Concrete execution* engine (LLVM backend)
     - Many parallel calls in tools like test coverage
  2. *Symbolic execution* engine (Haskell backend)
     - Symbolic paths can be explored in parallel
- Efficient implementations of these two engines will be offered as Software as a Service (SaaS) in the cloud
  + Wait seconds, not minutes or hours!
  + Good error messages, good debugger, good UX
  + Proof objects, too (discussed shortly)
Automated
Lightweight
Powered by

- equivalence checker
- symbolic state explorer / debugger
- test case generation
- property checker
- test case coverage
- test runner

firefly
Firefly = K [ EVM ] + Automation
KaaS

formal verification as a service
Programming Language (EVM, WASM, Solidity, Vyper, ...)

Code to verify (EVM, WASM, Solidity, Vyper, ...)

Hints

Properties to verify

K Tool
(say the program verifier)

To be invoked 100’s or 1000’s of times during a verification effort.
Best Approach Ever! 😊
But still a lot to trust 😞

- Programming Language
  - (EVM, WASM, Solidity, Vyper, ...)

- Code to verify
  - (EVM, WASM, Solidity, Vyper, ...)

- Hints

- Properties to verify

K Tool
(say the program verifier)

100,000+ LOC, several languages. Too much to trust!

- Trusted
- False
- True
Proof Object Generation

Formal Language Definition (Syntax and Semantics)

- Parser
- Interpreter
- Compiler
- Deductive program verifier
- Model checker
- Symbolic execution
- (semantic) Debugger
Proof Object Generation

• Each of the K tools is a best-effort implementation of proof search in Matching μ-Logic:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposition 1</td>
<td>( \varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_1) )</td>
</tr>
<tr>
<td>Proposition 2</td>
<td>((\varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_3)) \rightarrow (\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_1 \rightarrow \varphi_3) )</td>
</tr>
<tr>
<td>Proposition 3</td>
<td>((\neg \varphi_1 \rightarrow \neg \varphi_2) \rightarrow (\varphi_2 \rightarrow \varphi_1) )</td>
</tr>
<tr>
<td>Modus Ponens</td>
<td>( \varphi \rightarrow \varphi_1 \rightarrow \varphi_2 )</td>
</tr>
<tr>
<td>Variable Substitution</td>
<td>( \forall x. \varphi \rightarrow \varphi[y/x] )</td>
</tr>
<tr>
<td>(V)</td>
<td>( \forall x. (\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_1 \rightarrow \forall x. \varphi_2) ) if ( x \notin FV(\varphi_1) )</td>
</tr>
<tr>
<td>Universal Generalization</td>
<td>( \forall x. \varphi \rightarrow \exists x. \varphi )</td>
</tr>
<tr>
<td>Propagation 1</td>
<td>( C_\varphi[\bot] \rightarrow \bot )</td>
</tr>
<tr>
<td>Propagation V</td>
<td>( C_\varphi[\varphi_1 \lor \varphi_2] \rightarrow C_\varphi[\varphi_1] \lor C_\varphi[\varphi_2] )</td>
</tr>
<tr>
<td>Propagation 3</td>
<td>( C_\varphi[\exists x. \varphi] \rightarrow \exists x. C_\varphi[\varphi] ) if ( x \notin FV(C_\varphi[\exists x. \varphi]) )</td>
</tr>
<tr>
<td>Framing</td>
<td>( \varphi \rightarrow \varphi \rightarrow \varphi )</td>
</tr>
<tr>
<td>Existence</td>
<td>( \exists x. \varphi \rightarrow C_\varphi[\varphi_1] \rightarrow C_\varphi[\varphi_2] )</td>
</tr>
<tr>
<td>Singleton Variable</td>
<td>( \neg (C_1[x \land \varphi] \land C_2[x \land \neg \varphi]) ) where ( C_1 ) and ( C_2 ) are nested symbol contexts.</td>
</tr>
<tr>
<td>Set Variable Substitution</td>
<td>( \varphi[\psi/X] \rightarrow \varphi[\psi/X] )</td>
</tr>
<tr>
<td>Pre-Fixpoint</td>
<td>( \varphi[\mu X. \varphi/X] \rightarrow \mu X. \varphi )</td>
</tr>
<tr>
<td>(Knaster-Tarski)</td>
<td>( \mu X. \varphi \rightarrow \psi )</td>
</tr>
</tbody>
</table>

16 proof rules only! Simple proof checker (<200 LOC)!
In contrast, Coq has about 45 proof rules, and its proof checker has 8000+ lines of OCAML

• New Haskell backend of K will explicitly generate proof objects for verification tasks
Proof Objects

K Tool
(say the program verifier)

Properties to verify

Proof Object
(large!)

Proof Checker
(3rd party)
(<200 LOC)

Programming Language
(EVM, WASM, Solidity, Vyper, ...)

Hints

Trusted

Trusted

Trusted

False

True
Assured Trust. Like Never Before!

Programmable Language
(EVM, WASM, Solidity, Vyper, ...

Proof Object (huge!)

Proof Checker (3rd party)
(<200 LOC)

Properties to verify

True

Profitable, too. Proof objects as SaaS. Digital asset rating companies will require them!

Trust in language and properties unavoidable! Proof checker easy to trust: small, public, standardized.
K - Powered
Blockchain
K as a Universal Blockchain Language

• We want to be able to write (provably correct) smart contracts in any programming language.
• All you need is a $K$-powered blockchain!

K language semantics will be stored on blockchain. Fast LLVM backend of K can be used as execution engine / VM.
K as a Smart Contract Language

• Smart contracts implement transactions
  – Often using poorly designed and thus insecure languages, compilers and interpreters / VMs

  K also implements transactions, directly!
  – Indeed, each K rule instance is a transaction

• Each smart contract (Solidity, EVM, ...) requires a formal specification in order to be verified

  K formal specifications are already executable!
  – And indeed, they are validated by heavy testing

Hm, then why not write my smart contracts directly and only as K executable specifications?
Example: ERC20 Token in Solidity

- Snippet -

```solidity
pragma solidity ^0.5.0;

import "./IERC20.sol";
import "../../math/SafeMath.sol";

contract ERC20 is IERC20 {
    using SafeMath for uint256;

    mapping (address => uint256) private _balances;

    function transfer(address to, uint256 value) public returns (bool) {
        _transfer(msg.sender, to, value);
        return true;
    }

    function _transfer(address from, address to, uint256 value) internal {
        require(to != address(0), "ERC20: transfer to the zero address");

        _balances[from] = _balances[from].sub(value);
        _balances[to] = _balances[to].add(value);
        emit Transfer(from, to, value);
    }
}
```
Example: ERC20 Compiled to EVM

- Snippet -

Opcodes:

- Unreadable
- Slow: ~25ms to execute (ganache)
- Untrusted compiler, so it needs to be formally verified to be trusted
  - We formally verify it using KEVM against the following K specification:
K Specification of ERC20
- Snippet, Sugared -

```
rule transfer(To, V) => true
    caller: From
    account: id: From balance: BalanceFrom => BalanceFrom - V
    account: id: To balance: BalanceTo => BalanceTo + V
    log: . => Transfer(From, To, V)
requires 0 <= V <= BalanceFrom \ BalanceTo + V <= MAXVALUE
```

- **Formal**, yet understandable by non-experts
- **Executable**, thus testable (for increased confidence)
- **Fast**: ~2ms to execute with LLVM backend of K
- **No compiler required**, correct-by-construction
- **Use K as programming language for smart contracts**!
  (needed: gas model for K)
Conclusion: It Can Be Done!

Formal Language Definition (Syntax and Semantics)

- Parser
- Interpreter
- Compiler
- Deductive program verifier
- Model checker
- Symbolic execution
- (semantic) Debugger
Extra Slides
Semantics-Based Compilation

- Formal Language Definition (Syntax and Semantics)
  - Compiler
  - Interpreter
  - Parser
  - Deductive program verifier
  - Model checker
  - Symbolic execution
  - (semantic) Debugger
Semantics-Based Compilation (SBC)

Goals

– Execution of \( P \) in \( L \) equivalent to executing \( L_P \) in a start configuration
– \( L_P \) should be “as simple as possible”, only capturing exactly the dynamics of \( L \) necessary to execute program \( P \)
Semantics-Based Compilation (SBC) Experiments with Early Prototype

```c
// start
int b, n, x;
b = 1; n = 1; x = 0;

// outer
while (b <= 27) {
    n = b;

    // inner
    while (2 <= n) {
        if (n <= ((n / 2) * 2)) {
            n = n / 2;
        } else {
            n = (3 * n) + 1;
        }
        x = x + 1;
    }
    b = b + 1;
}
// end
```

<table>
<thead>
<tr>
<th>Program</th>
<th>Original (s)</th>
<th>Compiled (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum.imp</td>
<td>70.6</td>
<td>7.3</td>
<td>9.7</td>
</tr>
<tr>
<td>collatz.imp</td>
<td>34.5</td>
<td>2.7</td>
<td>12.8</td>
</tr>
<tr>
<td>collatz-all.imp</td>
<td>77.4</td>
<td>5.7</td>
<td>13.6</td>
</tr>
<tr>
<td>krazy-loop.imp</td>
<td>67.6</td>
<td>3.3</td>
<td>20.5</td>
</tr>
</tbody>
</table>
K – A Universal Blockchain Language

• *K-powered blockchain* enables (provably correct) smart contracts in *any* programming language!

1. Write contract \( P \) in any language, say \( L \) (unique address)
2. \( \text{SBC}[L] \) your \( P \) into \( L_p \); verify \( P \) (or \( L_p \)) with K prover