Formal Design, Implementation and Verification of Blockchain Languages

Grigore Rosu

University of Illinois at Urbana-Champaign
President & CEO, Runtime Verification Inc.
Think “execute some given, publicly visible code, with shared state”!

Transaction is broadcast, then “validated” by re-executing it on many “nodes”, using agreed upon languages (e.g., virtual machines).

Validated transactions are then deployed by all nodes locally...

Some transactions add new code to the blockchain, called “smart contracts”, which can be executed by other transactions.

In blocks, appending each block, irreversibly, to the public “ledger” or “history” or “blockchain”.

In the end, all code is public, can be invoked by anybody, and can irreversibly change the history (e.g., steal your money).

Hackers have huge incentives to exploit any bugs in smart contracts or underlying infrastructure.
Smart Contract Snippet (ERC20)
(one of the ~40,000 Ethereum ERC20s)

Written in Solidity:

```solidity
function transfer(address _to, uint256 _value) returns (bool success) {
    if (_value == 0) { return false; }

    uint256 fromBalance = balances[msg.sender];
    bool sufficientFunds = fromBalance >= _value;
    bool overflowed = balances[_to] + _value < balances[_to];

    if (sufficientFunds && !overflowed) {
        balances[msg.sender] -= _value;
        balances[_to] += _value;

        Transfer(msg.sender, _to, _value);
        return true;
    } else { return false; }
}
```

ERC20 does not state that...

There should be no overflow when self-transfer...
Attacks Happened. Many.

Parity Multisig Hacked. Again

Yesterday, Parity Multisig Wallet was hacked again:
https://paritytech.io/blog/security-alert.html

“This means that currently no funds can be moved out of the [ANY Parity] multisig wallets”

A lot of people/companies/ICOs are using Parity-generated multisig wallets. About $300M is frozen and (probably) lost forever.

Disclaimer: I lost little money (about $1000) but my friends lost about $300K.

Who hacked it?

Some guy with a nickname @devops199 (not a member of the Parity team) and an “empty” github account. His Ethereum address is 0xae7168Deb525862f4FEe37d987A971b385b96952 and he has successfully verified it.

The exploit occurred in April when the attacker stole all the money in the DAO.
What Can **We** Do About This?

• More specifically, what can we do about the **execution environment**, to increase security?
  – Unacceptable to build this complex and disruptive technology with poorly designed VMs and languages!

• **Ideal scenario feasible, stop compromising!**
  – Everything must be rigorously designed, using formal methods. Implementations must be **provably correct**!
    • Nodes: **provably correct** VMs or interpreters
    • Smart contracts: use **well-designed programming languages**, with **provably correct** compilers or interpreters
    • Verification: Smart contracts **provably correct** wrt their specs
Ideal Language Framework Vision

Formal Language Definition
(Syntax and Semantics)

Parser

Interpreter

Compiler

Deductive program verifier

Model checker

Symbolic execution

(semantic) Debugger
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

...
How It Should Be

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 *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

\[
\text{SYNTAX } \quad \text{Exp} ::= \ldots \quad | \quad \text{Exp} = \text{Exp} \quad [\text{strict(2)}]\n\]
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

\[
\frac{X = V}{V}
\]

rule

\[
<k> \quad X = V \implies V \quad ...<k>
\]

\[
<env>... \quad X \mid \neg\neg (\_ \implies V) \quad ...<env>
\]
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**, **Plutus**, **Vyper**, ...
Ideal Language Framework Vision

- Parser
- Interpreter
- Compiler
- Formal Language Definition (Syntax and Semantics)
- Deductive program verifier
- Model checker
- Symbolic execution
- (semantic) Debugger
1. Translate K lang def to OCAML
2. Compile OCAML code natively

Code (6-int-overflow.c)

```c
int main() {
    short int a = 1;
    int i;
    for (i = 0; i < 15; i++) {
        a *= 2;
    }
    return a;
}
```

RV-Match: Commercial tool
- Instance of K -> OCAML with ISO C11 language
- an automatic debugger for subtle bugs other tools can't find, with no false positives
- seamless integration with unit tests, build infrastructure, and continuous integration
- a platform for analyzing programs, boosting standards compliance and assurance

Conventional compilers do not detect problem

RV-Match’s kcc tool precisely detects and reports error, and points to ISO C11 standard
From RV-Match to Blockchain

• RV-Match currently commercialized within

• The same technology, K, used for defining blockchain languages: EVM, eWASM, IELE, ...
Ideal Language Framework Vision

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

Formal Language Definition (Syntax and Semantics)
State-of-the-Art

• Redefine the language using a different semantic approach (Hoare/separation/dynamic logic)

• Language specific, non-executable, error-prone

\[
\begin{align*}
\mathcal{H} & \vdash \{\psi \land e \neq 0\} \; s \; \{\psi\} \\
\mathcal{H} & \vdash \{\psi\} \text{while}(e) \; s \; \{\psi \land e = 0\} \\
\mathcal{H} & \cup \{\psi\} \text{proc}() \{\psi'\} \vdash \{\psi\} \text{body} \; \{\psi'\}
\end{align*}
\]
What We Want

- Use directly the trusted executable semantics!
- **Language-independent proof system**
  - Takes operational semantics as axioms
  - Derives reachability properties
  - Sound and relatively complete for all languages!
Matching Logic

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

Patterns (of each sort \(s\))

\(\varphi_s \ ::= \)

- \(x \in \text{Var}_s\)
- \(\sigma(\varphi_{s_1}, \ldots, \varphi_{s_n})\)
- \(\neg \varphi_s\)
- \(\varphi_s \land \varphi_s\)
- \(\exists x. \varphi_s\)

Structure

with \(\sigma \in \Sigma_{s_1 \ldots s_n, s}\)

Constraints

with \(x \in \text{Var} \) (of any sort)

Binders
Matching $\mu$-Logic

- Adding support for recursion / induction

$$\varphi_s ::= \ldots \mid X:s \in \text{SVAR}_s$$

$$\mid \mu X:s.\varphi_s \quad \text{if } \varphi_s \text{ is positive in } X:s$$

**Pre-Fixpoint**

$$\varphi[\mu X.\varphi/X] \rightarrow \mu X.\varphi$$

**Knaster-Tarski**

$$\varphi[\psi/X] \rightarrow \psi$$

$$\mu X.\varphi \rightarrow \psi$$
Expressiveness

Important logics for program reasoning can be framed as matching logic theories / notations

- First-order logic
  - Equality, membership, definedness, partial functions
- Lambda / mu calculi (least/largest fixed points)
- Modal logics
- Hoare logics
- Dynamic logics
- LTL, CTL, CTL*
- Separation logic
- Reachability logic
- ...
Reachability Logic (Semantics of K)
[LICS’13, RTA’14, RTA’15, OOPLSA’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
  \[
  \{\text{Pre}\} \text{Code}\{\text{Post}\} \equiv \widehat{\text{Code}} \wedge \widehat{\text{Pre}} \Rightarrow \epsilon \wedge \widehat{\text{Post}}
  \]

• Sound & relative complete proof system
  – Now proved as matching logic theorems
K = (Best Effort) Implementation of RL

- Reachability logic implemented in K, generically

- 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
K for the Blockchain
K Framework Vision

Parser

Deductive program verifier

Interpreter

Model checker

Compiler

Symbolic execution

(symbolic) Debugger

(Blockchain Language Definition

(Syntax and Semantics)
KEVM: Semantics of the Ethereum Virtual Machine (EVM) in K

Complete semantics of EVM in K

- [https://github.com/kframework/evm-semantics](https://github.com/kframework/evm-semantics)
- Passes 60,000+ tests of C++ reference implementation
- 8x (only!) slower than the C++ implementation
- Adoption by the Ethereum Foundation
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!* IOHK has just done that, in collaboration with RV, as a Cardano testnet:
What Can We Do with KEVM?

3) *Formally verify Ethereum smart contracts!* RV is doing that, commercially. RV also won first Ethereum Security grant to verify Casper.
Formalizing ERC20, ERC777, ... in K

• *K is very expressive*: can define not only languages, but also *token specifications and protocols*

• 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

• Others use or integrate our framework and specs:
  – Consensys, DappHub ([KLab](https://kchain.com)), ETHWorks ([Waffle](https://waffle.io)), Gnosis
Transfers \( \_\text{value} \) amount of tokens to address \( \_\text{to} \), and MUST fire the \text{Transfer} event. The function SHOULD throw if the \( \_\text{from} \) account balance does not have enough tokens to spend.

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

```
function transfer(address \_\text{to}, uint256 \_\text{value}) returns (bool success)
```

```
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
```

```
[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
```

```
..........
..........
```
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
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
- **Plutus** (functional) – in progress, by RV following Phil Wadler’s (@IOHK) design of the language
- **Vyper** – in progress, by RV in collaboration with the Ethereum Foundation
- ...
K Modelling and Verification of Blockchain Protocols

• K and rewriting can also be used to formally specify and verify consensus protocols, random number generators, etc.; same tool eco-system

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

• Several others planned or in discussions
Ongoing K Infrastructure Projects
1. Fast LLVM (and IELE) Backend for K

- Deductive program verifier
- Model checker
- Symbolic execution
- (semantic) Debugger
- Compiler
- Interpreter
- Formal Language Definition (Syntax and Semantics)
- Parser
Fast LLVM (and IELE) Backend for K

• Current OCAML backend of K:
  • Fast enough to power RV-Match product and the KEVM and IELE VMs in testnets
  • But still one or two orders of magnitude slower than hand-crafted interpreters

• LLVM backend for K under development:
  • Take advantage of LLVM’s optimizations / pipeline
  • Expected to compete with hand-written interpreters!
  • Will make language designers ask themselves the question
    “Why would I implement an interpreter/VM by hand, when I can generate one automatically, correct-by-construction?”
2. Semantics-Based Compilation

- Formal Language Definition (Syntax and Semantics)
  - Parser
  - Interpreter
  - Compiler
  - 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

// 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>
3. 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 $\mu$-Logic:

- Proposition 1: $\varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_1)$
- Proposition 2: $(\varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_3)) \rightarrow (\varphi_1 \rightarrow \varphi_2) \rightarrow (\varphi_1 \rightarrow \varphi_3)$
- Proposition 3: $(\neg \varphi_1 \rightarrow \neg \varphi_2) \rightarrow (\varphi_2 \rightarrow \varphi_1)$
- Modus Ponens: $\varphi_1 \rightarrow \varphi_2$
- Variable Substitution: $\forall x, \varphi \rightarrow \varphi[y/x]$
- Universal Generalization: $\forall x, \varphi \rightarrow \forall x, \varphi$
- Propagation 1: $C_\varphi[\perp] \rightarrow \perp$
- Propagation 2: $C_\varphi[\varphi_1 \lor \varphi_2] \rightarrow C_\varphi[\varphi_1] \lor C_\varphi[\varphi_2]$
- Propagation 3: $C_\varphi[\exists x, \varphi] \rightarrow \exists x, C_\varphi[\varphi]$ if $x \notin FV(C_\varphi[\exists x, \varphi])$
- Framing: $\varphi_1 \rightarrow \varphi_2$
- Existence: $\exists x, \varphi$
- Singleton Variable: $\neg (C_1[x \land \varphi] \land C_2[x \land \neg \varphi])$
- Set Variable Substitution: $\varphi[\psi/X]$
- Pre-Fixpoint: $\varphi[\mu X, \varphi/X] \rightarrow \mu X, \varphi$
- Knaster-Tarski: $\mu X, \varphi \rightarrow \psi$

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 Object Generation

• No need to trust the (complex) K implementation, nor any company (including Runtime Verification)
  – It is known that program verifiers / tools can have bugs in spite of best efforts, bug finders and company prestige
• Proof objects act as 3rd-party checkable correctness certificates on the blockchain, in a proof-carrying code style (proofs can be stored offchain, or snarked)
• In combination with domain-specific languages for requirements specifications, this will offer the highest level of software assurance known to man
Ultimate Goal

a Universal Blockchain Technology
K – 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 as execution engine / VM.
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. SBC[L] your P into $L_P$; verify P (or $L_P$) with K prover
K-Powered Blockchains

• K may be used one day to generate correct-by-construction (CBC) blockchains; not a dream, no!

Each node is a K VM client (LLVM backend)

Consensus protocol formalized and verified in K, implementation generated from specification, CBC
K – A Universal Blockchain Language

• When all the projected K tools will be completed, K will provide everything we need to
  – Design new smart contract languages, add them in the blockchain and start using them right away, with auto-generated correct(!) implementations and tools
  – Same for virtual machines(!) and consensus protocols(!)
• Everything will be either a trusted formal specification or generated automatically from one
• Proof objects will serve as correctness certificates
• *Perfect. No compromise!*
Moreover...

a Ultimate Smart Contract Language
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, no code to formally verify
- Use K as programming language for smart contracts!