A Language-Independent Approach to Smart Contract Verification

Xiaohong Chen[1], Daejun Park[1,2], and Grigore Roşu[1,2]

[1] University of Illinois at Urbana-Champaign
[2] Runtime Verification Inc.

Abstract. This invited paper reports the current progress on smart contract verification with the $\mathbb{K}$ framework in a language-independent style.

1 Introduction and Motivation

Flaws of blockchain programming languages or virtual machines have led and continue to lead to cryptocurrency software bugs that directly translate into significant money loss [6,4,1,3,14]. Formal analysis and verification of blockchain languages and virtual machines is thus very much in need. Traditionally, this is done by giving a formal model of the program-to-verify, either by a manual construction in theorem provers such as Coq [10] or Isabelle [15], or by a translation to some intermediate verification languages (IVL) such as Boogie [2] or Why [7]. Developing program models in theorem provers can be expensive and is only done to mission critical systems, while a translation to IVL may loose program behavior. In either case, a *trusted formal semantics* of the target language together with a proof of correctness of either the program models built in Coq or Isabelle, or the translation to IVL, are required. Such correctness proofs are often done manually on paper and can be expensive. They are also sensitive to the target languages and programs, so small changes on the verification targets require to redo the proofs. Due to the fact that blockchain programming languages are often moving targets and have a rather rapid development cycle, with new versions being released and deployed in a weekly pace, the traditionally program verification approaches are often too expensive to use in practice.

The $\mathbb{K}$ framework [13] adopts a *language-independent* approach to program verification; it was derived from our firm belief that every programming language must have a formal semantics, and that all formal or informal analysis tools for that language should be automatically generated from that semantics in a correct-by-construction manner. Fig. 1 illustrates the $\mathbb{K}$ approach. In terms of verification, the *language-independent verifier* is parametric on the semantics of the language, and it takes as input a program and a specification of the program and solves the verification problem (see Fig. 2). Extensive experiments and case studies confirm that this language-independent approach to verification is feasible. For example, [5,11] show that when instantiated with formal semantics of real languages such as C, Java, and JavaScript, the generic $\mathbb{K}$ program

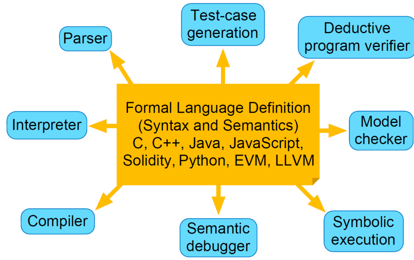

Fig. 1: The $\mathbb{K}$ framework approach to language design and formal verification.

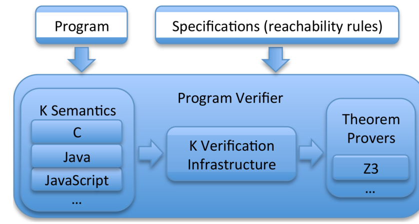

Fig. 2: A language-independent program verifier takes a program and its specification, and verifies it with respect to its formal semantics.

verifier is able to check well-known challenging functional correctness properties of heap manipulation programs with mutable data structures, such as AVL trees, read-black trees, and even the Schorr-Waite graph marking algorithm, all implemented in each of C, Java, and JavaScript. Nothing was needed in the generic verifier specific to any of these languages, except for their formal semantics. When it comes to blockchain languages, the advantage of the $\mathbb{K}$ approach is even more significant, as languages and virtual machines in this field change at an unusually high rate and thus there is no need to redo the correctness proofs for either the high-level program models or the translation to IVL. All verification tools are correct-by-construction, and thus are suitable to the rapid development cycle of blockchain languages.

In the rest of the paper, we briefly introduce the $\mathbb{K}$ framework in Section 2 and summarize the current progress on blockchain languages and smart contracts verification in Section 3. Then we discuss the general workflow of smart contract verification with $\mathbb{K}$ framework in Section 4, and conclude with future work in Section 5.

2 An Overview of the $\mathbb{K}$ Framework

The $\mathbb{K}$ framework is a rewrite-based executable semantics framework for programming language design and development. It can be regarded as a meta-programming language that defines programming languages. As an example, consider the simple imperative language IMP whose syntax is given in Fig. 3. IMP has arithmetic expressions and the usual assignment, sequential, if-, and while-statements. Arithmetic expressions are used as conditions where zero means false and nonzero values mean true. The complete $\mathbb{K}$ definition of IMP is given in Fig. 4. The definition consists of two modules `IMP-SYNTAX` and `IMP`. The module `IMP-SYNTAX` defines the concrete syntax using the conventional BNF grammar where terminals are in quotes. Production rules are separated by the "`|`" and "`>`", where "`|`" means the two productions (before and after "`|`") have the same precedence while "`>`" means the production before has higher precedence (binds tighter) than the ones after. In other words, all the other language

$$
\begin{aligned}
\mathit{Exp} &::= \mathit{Id} \mid \mathit{Int} \mid \mathit{Exp}\ \texttt{+}\ \mathit{Exp} \mid \mathit{Exp}\ \texttt{-}\ \mathit{Exp} \\
\mathit{Stmt} &::= \mathit{Id}\ \texttt{=}\ \mathit{Exp}\texttt{;} \mid \mathit{Stmt}\ \mathit{Stmt} \mid \texttt{\{}\ \mathit{Exp}\ \texttt{\}} \mid \texttt{if(}\mathit{Exp}\texttt{)}\ \mathit{Stmt}\ \mathit{Stmt} \mid \texttt{while(}\mathit{Exp}\texttt{)}\ \mathit{Stmt} \\
\mathit{Ids} &::= \mathit{Id} \mid \mathit{Id}\texttt{,}\mathit{Ids} \\
\mathit{Pgm} &::= \texttt{int}\ \mathit{Ids}\texttt{;}\ \mathit{Stmt}
\end{aligned}
$$

Fig. 3: The syntax of the language IMP.

constructs bind tighter than the sequential operator in IMP. Categories `Int` and `Id` are built-in categories for integers and identifiers (program variables), respectively. `Exp` is the category for expressions, which subsumes `Int` and `Id` and has two productions for plus and minus. `Pgm` is the category for programs, which is a declaration of a list of program variables (the category `Ids`) followed by a statement. `Ids` is defined using 𝕂's built-in list template `List`, whose second argument is the separating character. In other words, `Ids` is the category of comma-separated lists of `Id`'s.

Attributes are wrapped with braces "`[`" and "`]`". Some attributes are only for parsing purpose while others may carry additional semantic meaning and affect how 𝕂 executes programs. The attribute `left` means left-associative. The attribute `strict` defines evaluation contexts, so when 𝕂 sees the expression e_1 `+` e_2 (and similarly e_1 `-` e_2), it first evaluates e_1 to an integer i_1 and e_2 to an integers i_2 in a *fully nondeterministic* way, and then evaluates $i_1 + i_2$. The attribute `strict(1)` means if 𝕂 sees the if-statement `if(`B`)` P Q it should only evaluates the first argument B to a value v while keeping the other arguments P and Q untouched. Therefore, the two branches of if-statement are *frozen* and will not be evaluated if the condition is not a value. The attribute `bracket` tells 𝕂 that certain productions are only used for grouping, and 𝕂 will not generate nodes in its internal abstract syntax trees for those productions. Here, parentheses are used to group arithmetic expressions while curly brackets are used to group program statements. The empty curly bracket "`{}`" represents the empty statement.

The module `IMP` defines the operational semantics of IMP in terms of a set of human readable rewrite rules. The category `KResult` tells 𝕂 which categories contain non-reducible values. It helps 𝕂 perform efficiently with evaluation contexts. The only category of values here is `Int`. Configuration is a core concept in the 𝕂 framework. A *configuration* of a language holds all information that is needed to execute programs, gathered in *cells*. Simple languages such as IMP have only a few cells, while complex real languages such as C usually have more than one hundred. In 𝕂, configurations are defined using using a syntax borrowed from the XML format. The configuration of IMP contains two cells: the `k` cell and the `state` cell. For clarity, we put all cells in configuration in a top cell: the `T` cell, but it is not mandatory. The `k` cell holds the remaining computation (program) that needs to execute and the `state` cell holds a mapping from program variables to their values in the memory. Initially, the `state` cell holds the

```
module IMP-SYNTAX
  imports DOMAINS-SYNTAX
  syntax Exp  ::= Int | Id
                | Exp "+" Exp                                         [left, strict]
                | Exp "-" Exp                                         [left, strict]
                | "(" Exp ")"                                              [bracket]
  syntax Stmt ::= Id "=" Exp ";"                                         [strict(2)]
                | "if" "(" Exp ")" Stmt Stmt                             [strict(1)]
                | "while" "(" Exp ")" Stmt
                | "{" Stmt "}"                                             [bracket]
                | "{" "}"
                > Stmt Stmt                                                   [left]
  syntax Pgm  ::= "int" Ids ";" Stmt
  syntax Ids  ::= List{Id, ","}
endmodule

module IMP
  imports IMP-SYNTAX
  imports DOMAINS
  syntax KResult ::= Int
  configuration <T> <k> $PGM:Pgm </k> <state> .Map </state> </T>
  rule <k> X:Id => I ...</k> <state>... X |-> I ...</state>
  rule I1 + I2 => I1 +Int I2
  rule I1 - I2 => I1 -Int I2
  rule <k> X = I:Int; => . ...</k> <state>... X |-> (_ => I) ...</state>
  rule S1:Stmt S2:Stmt => S1 ~> S2                                    [structural]
  rule if (I) S _ => S requires I =/=Int 0
  rule if (0) _ S => S
  rule while(B) S => if(B) {S while(B) S} {}                          [structural]
  rule {} => .                                                        [structural]
  rule <k> int (X, Xs => Xs); S </k>  <state> ... (. => X |-> 0) </state>  [structural]
  rule int .Ids; S => S                                               [structural]
endmodule
```

Fig. 4: The complete 𝕂 definition of the language IMP, consisting of two modules.

empty map, denoted as `.Map`. In 𝕂, we use dot "`.`" to denote "nothing", and `.Map` means the nothing has type `Map`. The `k` cell initially contains a program `$PGM:Pgm`, where `$PGM` is a special 𝕂 variable name. To execute an IMP program, say `sum.imp`, the name of the source file is passed to 𝕂, and 𝕂 will parse the source file using the concrete syntax and associate the result (of category `Pgm`) to the variable `$PGM:Pgm` in the `k` cell.

𝕂 defines the language semantics in terms of a set of rewrite rules. These rewrite rules specify a transition system on *configurations*. We point out two important characteristics of rewrites rules in 𝕂. The first important characteristic of rewrites rules of 𝕂 is that 𝕂 supports *local rewrites*. In other words, the rewrite symbol "`=>`" does not need to appear in the top level, but can appear locally in which the rewrite happens. Take the lookup rule as an example. Instead of

```
rule <k> X:Id ...</k> <state>... X |-> I ...</state>
  => <k> I ...</k> <state>... X |-> I ...</state>
```

one writes

```
rule <k> X:Id => I ...</k> <state>... X |-> I ...</state>
```

to reduce space and avoid duplicates. The "`...`" in 𝕂 is a shortcut for things that "exist and do not matter and change." The rule says that if the top of the computation in the `k` cell is a program variable `X:Id`, and at the same time `X` binds to the integer `I` somewhere in the `state` cell, then rewrite `X:Id` to `I`.

The second characteristic of rewrite rules in 𝕂 is that 𝕂 also supports *configuration inference* and *configuration completion*. The rewrite rules may not explicitly mention all cells in configuration, but just related ones. 𝕂 will infer the implicit cells and complete the configuration automatically. For example, instead of

```
rule <T> <k> I1 + I2 => I1 +Int I2 ... </k> <state> M </state> </T>
```

one writes

```
rule I1 + I2 => I1 +Int I2
```

which is not only a lot simpler, but also extensible. If we add a new cell to the configuration, we will not need to modify any of the existing rules.

The rest of the semantics is self-explanatory. The rule for assignment statements `X = I:Int;` updates the value bound to `X` in the `state` cell, as specified in the local rewrite `X |-> (_ => I)`. Here the underscore "`_`" is an anonymous 𝕂 variable; it matches whatever integer that is currently bound to `X`. After the update, the assignment statement is removed from the `k` cell, as specified by the local rewrite `X = I:Int; => .`. Recall that the dot "`.`" means nothing, and rewriting something to a dot means removing it. Attribute `structural` means the associated rewrite rule is not counted as an explicit step by 𝕂, but an implicit (quite) one. It should not affect how 𝕂 executes the programs. The empty statement `{}` simply reduces to nothing. The last two rules process the declaration list of program variables and initialize their values to zero.

3 Semantics of Blockchain Virtual Machines in 𝕂

KEVM. The Ethereum Virtual Machine (EVM) [16] is a low-level bytecode language running on a general-purpose "world computer" built by the blockchain cryptocurrency Ethereum. Small programs called smart contracts are allowed to execute on it, often written in high-level languages such as Solidity (`https://github.com/ethereum/solidity`) or Vyper (`https://github.com/ethereum/vyper`) and then compiled to EVM. To verify smart contracts, a formal semantics of the low-level EVM language was developed [9] using the 𝕂 framework, which we refer to as KEVM. As far as we know, KEVM is the first fully executable formal semantics of the EVM language. It is tested against the official 40,683-test stress test suite for EVM implementations that comes with the official C++ implementation of the EVM.

Based on KEVM, the startup Runtime Verification formally verified several smart contracts (`https://runtimeverification.com/smartcontract`), and the result is available for public access in the spirit of open-source (`https://github.com/runtimeverification/verified-smart-contracts`). Since December 2017, a

number of smart contracts have been verified with the 𝕂 framework; the following is a list of them in chronological order (older to newer):

– Vyper ERC20 Token Contact (`https://github.com/ethereum/vyper`);
– HackerGold (HKG) ERC20 Token Contract (`https://github.com/ether-camp/virtual-accelerator`);
– OpenZeppelin's ERC20 Token Contract (`https://github.com/OpenZeppelin/openzeppelin-solidity`);
– Bihu Smart Contract (`https://github.com/runtimeverification/verified-smart-contracts/tree/master/bihu`);
– DSToken ERC20 Token Contract (`https://github.com/dapphub/ds-token`)
– Ethereum Casper Contract (`https://github.com/runtimeverification/verified-smart-contracts/tree/master/casper`)

A surprising and pleasant observation in the process of the development of KEVM and the verification of smart contracts is that the EVM interpreter automatically generated by 𝕂 based on EVM formal semantics is only one order of magnitude slower on average than the official C++ implementation [8]. Since smart contracts are often small programs, the above suggests that KEVM can serve not only as a reference model of the EVM but also as an actual implementation.

IELE. Like EVM, IELE (`https://github.com/runtimeverification/iele-semantics`) is another virtual machines bytecode language. Unlike EVM, IELE was designed in the spirit of *easier formal verification*, and thus it is significantly different from EVM in various aspects. For example, IELE is a register-based machine, and it supports unbounded integers (as unbounded arithmetics is often easier than bounded arithmetics in verification). IELE was designed purely in a semantic-based style using 𝕂, and an automatically-generated virtual machine is derived from its formal semantics, which makes it the first virtual machine whose development and implementation was completely powered by formal methods.

4 Smart Contract Verification

In this section, we briefly discuss the workflow of smart contract verification, taking the open resource of the work of ERC20 verification (`https://github.com/runtimeverification/verified-smart-contracts/tree/master/erc20`) as a case study example. See [12] for more details.

The ERC20 token contract (abbrev. as ERC20 below) is one of the most popular and valuable smart contracts. An informal standard for ERC20 can be found at (`https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`), which we refer to as the ERC20 standard. The ERC20 standard essentially defines an API with an informal specification. Fig. 5 shows an example of a piece of informal specification of the function `transfer` in the ERC20 standard.

Transfers `_value` amount of tokens to address `_to`, and MUST fire the `Transfer` event. The function SHOULD `throw` if the `_from` account balance does not have enough tokens to spend.

Note Transfers of 0 values MUST be treated as normal transfers and fire the `Transfer` event.

```
function transfer(address _to, uint256 _value) returns (bool success)
```

Fig. 5: The informal specification of the function `transfer` in the ERC20 standard.

```
rule
  <k> transfer(To, Value) => true ... </k>
  <caller> From </caller>
  <account> <id> From </id>
    <balance>
      BalanceFrom => BalanceFrom -Int Value
    </balance> </account>
  <account> <id> To </id>
    <balance> BalanceTo => BalanceTo +Int Value
    </balance> </account>
  <log> Log => Log Transfer(From, To, Value)
  </log>
requires To =/=Int From andBool Value >=Int 0
 andBool Value <=Int BalanceFrom
 andBool BalanceTo +Int Value <=Int MAXVALUE
```

```
rule
  <k> transfer(From, Value)
   => throw ...
  </k>
  <caller> From </caller>
  <id> From </id>
  <balance> BalanceFrom </balance>
requires Value <Int 0
  orBool Value >Int BalanceFrom
```

Fig. 6: The formal specification of the function `transfer` in ERC20-K. The rule on the left shows the case when the transfer *succeeds* and the caller is *different* from the receiver. The rule on the right shows the case when the transfer *fails* and the caller is the *same* as the receiver.

The first step of the verification is to take the informal ERC20 standard and *refine* it to a formal specification. The outcome of the refinement, which we refer to as ERC20-K, is a $\mathbb{K}$ definition that captures the complete functionality of the ERC20 API (`https://github.com/runtimeverification/erc20-semantics`). For example, the above informal specification is divided into four cases in ERC20-K, namely all four combinations of whether the transfer succeeds or fails, and whether the caller is the same as or different from the receiver. Fig. 6 gives the formal specification for two of the fours cases. ERC20-K therefore formally specifies the entire ERC20 API and its intended behavior. It is worth mentioning how fast it is to develop such a complete executable formal specification in $\mathbb{K}$: the fully documented ERC20-K took a developer about two weeks to finish, with one week writing the rules and another week revising it, fixing bugs, and writing documentation.

Since smart contracts are compiled to lower-level EVM bytecode, we need to refine the high-level ERC20-K specification further, to an EVM-level formal specification, referred to as ERC20-EVM, which is based on KEVM and takes all EVM-specific details into account. Finally, various smart contracts have been verified with the ERC20-EVM specification and the built-in program verification infrastructure in $\mathbb{K}$. We refer interested readers to [12] as well as our

open source project (`https://github.com/runtimeverification/verified-smart-contracts`) for more experiment details and technical discussion.

5 Conclusion and Future Work

We hope this paper demonstrates that language-independent verification is possible and feasible, and is especially preferable for blockchain languages and smart contracts verification. With only one executable semantics, it suffices to generate all the tools in a correct-by-construction manner, and thus eliminate the need for redundant and error-prone proofs of correctness. In particular, for emerging fields like the blockchain and smart contracts where new languages and programs are released on a weekly or even daily basis, the language-independent approach seems to be the only viable solution. We hope that this wave of blockchain languages and smart contracts verification will raise interest from the community in language-independent semantics frameworks like $\mathbb{K}$ and drives application of the techniques to all languages. As of future work, two sides of research are needed. On the foundation side, a language-independent (program) logic is in need, which allows us to state and reason about any properties of any programs written in any programming languages. On the implementation side, automation of tools is needed.

Acknowledgments: We thank the $\mathbb{K}$ team (`http://www.kframework.org/index.php/People`) for their sustained dedication and help, as well as to numerous other contributors to the $\mathbb{K}$ framework.

References

1. Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts. In *Proceedings of the 6th International Conference on Principles of Security and Trust*, volume 10204, pages 164–186, 2017.
2. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Revised Lectures of the 4th International Symposium on Formal Methods for Components and Objects (FMCO'05)*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.
3. Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. An in-depth look at the parity multisig bug. 2017. `http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/`.
4. Vitalik Buterin. Thinking about smart contract security. 2016. `https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/`.
5. Andrei Ştefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*, pages 74–91. ACM, Nov 2016.
6. Phil Daian. DAO attack, 2016. `http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`.

7. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007.
8. Everett Hildenbrandt, Manasvi Saxena, Xiaoran Zhu, Nishant Rodrigues, Philip Daian, Dwight Guth, Brandon Moore, Yi Zhang, Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KEVM: A complete semantics of the Ethereum virtual machine. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF'18)*. IEEE, 2018. `http://jellopaper.org`.
9. KEVM Team. Kevm: Semantics of EVM in K. `https://github.com/kframework/evm-semantics`, 2017.
10. The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004.
11. Brandon Moore, Lucas Peña, and Grigore Roşu. Program verification by coinduction. In *Proceedings of the 27th European Symposium on Programming (ESOP'28)*, pages 589–618. Springer, 2018.
12. Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. A formal verification tool for Ethereum VM bytecode. In *Proceedings of the 2018 ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'18)*, 2018.
13. Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
14. Jutta Steiner. Security is a process: A postmortem on the parity multi-sig library self-destruct, 2017. `http://goo.gl/LBh1vR`.
15. The Isabelle development team. Isabelle, 2018. `https://isabelle.in.tum.de/`.
16. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014. (Updated for EIP-150 in 2017) `http://yellowpaper.io/`.