Research Proposal
Trustworthy Formal Verification for Ethereum Smart Contracts via Machine-Checkable Proof Certificates

Xiaohong Chen Grigore Roşu

{xc3,grosu}@illinois.edu

Abstract

The $\mathbb{K}$ language framework (https://kframework.org, https://github.com/runtimeverification/k) has been frequently used by companies such as Runtime Verification, DappHub, ConsenSys, and the Ethereum Foundation to formalize and verify the correctness of smart contracts, consensus protocols, and virtual machines. KEVM, which is a complete and executable formal semantics of the Ethereum Virtual Machine (EVM) defined using $\mathbb{K}$, serves a solid foundation for formal analyses for blockchain applications. From KEVM, a program verifier for EVM is automatically generated by $\mathbb{K}$ to formally verify the correctness of Ethereum smart contracts. On the other hand, concerns have risen over the correctness of $\mathbb{K}$ itself. The EVM verifier that is automatically generated from KEVM has not been formally verified. It thus puts the trustworthiness of the verification results obtained using $\mathbb{K}$ for the existing blockchain applications at risk.

In this proposal, we propose to study ***trustworthy formal verification via machine-checkable proof certificates***. The idea is to generate for each verification task in $\mathbb{K}$ a complete, rigorous, and machine-checkable mathematical proof that certifies the verification result. Such a proof certificate consists of the entire formal semantics of the target programming language (say, KEVM) as logical axioms and a logical proof that derives the intended properties of the smart contracts using a sound proof system. Then, the proof certificate is automatically proof-checked by ***a very small proof checker of only 240 lines of code*** (https://github.com/kframework/matching-logic-proof-checker/blob/main/theory/matching-logic-240-loc.mm). This way, the correctness of $\mathbb{K}$'s verification tools is established for each individual verification tasks via machine-checkable proof certificates, checked by a 240-line proof checker, without needing to trust the complex implementation of $\mathbb{K}$ anymore.

The proposed research will improve the trustworthiness of Ethereum smart contracts and blockchain applications by making their existing verification results more transparent and accessible to all stakeholders. The proof certificates will serve as independent correctness certificates of the smart contracts, consensus protocols, and virtual machines that are formally verified using $\mathbb{K}$. The proposed research thus eliminates $\mathbb{K}$ from the trust base, making smart contracts and blockchain applications more trustworthy.

1 Problem Statement

The $\mathbb{K}$ language framework (https://kframework.org, https://github.com/runtimeverification/k) has been frequently used by companies such as Runtime Verification, DappHub, ConsenSys, and the Ethereum Foundation to formalize and verify the correctness of smart contracts, consensus protocols, and virtual machines. As of today, more than 7 blockchain consensus protocols, 31 smart contracts, and 4 blockchain-oriented programming languages have been formalized and verified in $\mathbb{K}$ [29] (https://github.com/runtimeverification/publications).

$\mathbb{K}$ is a language formal semantics framework. The vision of $\mathbb{K}$, as depicted in Figure 1, is that programming languages must have their formal semantics defined in $\mathbb{K}$. From the formal semantics of any programming language, $\mathbb{K}$ automatically generates language tools (such as parsers, interpreters, compilers,

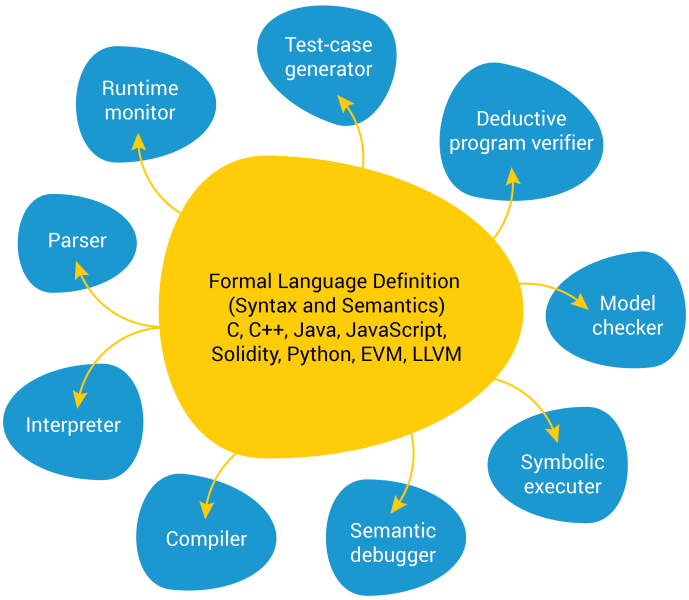

Figure 1: Vision of the $\mathbb{K}$ Framework

virtual machines, deductive verifiers, and others) of that language, at no additional costs. This way, $\mathbb{K}$ serves as a meta-language to define other programming languages.

$\mathbb{K}$ scales. Over the past decade, $\mathbb{K}$ has been used to define the formal semantics of many real-world programming languages such as [15], Java [2], JavaScript [24], Python [16], and x86-64 [12]. $\mathbb{K}$ has also been used to define the formal semantics of the Ethereum virtual machines (EVM) [18], Solidity (`https://github.com/kframework/solidity-semantics`), Vyper (`https://github.com/kframework/vyper-semantics`), and Ewasm (`https://github.com/kframework/ewasm-semantics`). The $\mathbb{K}$ framework itself and all the above formal language semantics defined in $\mathbb{K}$ are open-sourced under the UIUC license, which is permissive and free.

$\mathbb{K}$ ***has been used a lot in blockchain applications and their formal analyses.*** Runtime Verification Inc. uses $\mathbb{K}$ to define the first complete and executable formal semantics of Ethereum Virtual Machine (EVM), called KEVM, which provides an executable formal specification of the EVM's bytecode stack-based language, replacing the informal specification in the Yellow Paper [31] by the community. From KEVM, a deductive verifier for EVM is automatically generated by $\mathbb{K}$ at no additional costs, which has been used to specify and formally verify many Ethereum smart contracts as well as consensus protocols and virtual machines. KEVM has been thoroughly tested against the official Ethereum test suite and powered the commercial toolkit Firefly (`https://fireflyblockchain.com/`) that aims at improving the quality of Ethereum smart contracts, consisting of automatic and interactive tools for unit testing, integration testing, black-box/white-box random testing, runtime monitoring, as well as formal analysis tools such as a assertion violation checker, a bounded model checker, and symbolic testing. KEVM has also been adopted by ConsenSys, DappHub, the Ethereum Foundation, Gnosis, MakerDAO, and Uniswap as the semantic basis for smart contract verification.

However, the correctness of $\mathbb{K}$ itself has not been formally verified, which puts the trustworthiness of all the existing smart contract verification results obtained using $\mathbb{K}$ at risk. Indeed, $\mathbb{K}$ is a complicated software artifact, consisting of more than 500,000 lines of code in Haskell, Java, and C++, with new code committed to the code base on a weekly basis. Thus, the correctness of the current implementation of $\mathbb{K}$ cannot be taken for granted. ***We—both the formal semanticists and the blockchain community—must distinguish what is verified and what is trustworthy.*** Formally verifying a system does not necessarily make it

more trustworthy, but only *transfers* the trust on the system in question to that on the verifiers, which are even more complex than most of the real-world smart contracts.

Our goal is to make blockchain applications and smart contracts more trustworthy, by generating for them machine-checkable proof certificates. Concretely, for each smart contract that is verified using $\mathbb{K}$, we generate a complete, rigorous, and machine-checkable mathematical proof that independently certifies the verification result. The proof certificate consists of the entire formal semantics of the related programming language(s) as logical axioms and a logical proof that derives the intended properties of the smart contract in question using a sound proof system. In our proposed approach, the proof certificates have the following characteristics.

1. The proof certificates will explicit specify the logical axioms that are used, including the entire formal languages semantics and any internal and/or external assumptions made by $\mathbb{K}$ and/or the users.
2. The proof certificates will include all the detailed proof steps, making proof checking very easy.
3. All proof certificates will be checked by the same ***very small and fast 240-line proof checker*** (`https://github.com/kframework/matching-logic-proof-checker/blob/main/theory/matching-logic-240-loc.mm`).

By generating these machine-checkable proof certificates, we establish the correctness of blockchain applications and smart contracts ***without needing to trust*** $\mathbb{K}$. These proof certificates make the existing smart contract verification obtained using $\mathbb{K}$ more transparent and accessible to all the blockchain stakeholders, and not only to expert semanticists and verification engineers, because what is now hidden behind $\mathbb{K}$'s internal verification algorithms, transformations, heuristics, and optimization will be made explicit by concrete and machine-checkable proof steps. Smart contract verification results are therefore reproducible and independently checkable, making the $\mathbb{K}$-verified smart contracts more trustworthy.

Open Source Work. The entire $\mathbb{K}$ framework and all the formal programming languages semantics mentioned above are open-sourced under the UIUC license, which is permissive and free. All our publications are accessible via `https://fsl.cs.illinois.edu/publications/` and `https://runtimeverification.com/publications/`. All the future publications coming from the proposed research will also be made accessible via the two URLs above.

The rest of the proposal is organized as follows.

- Section 2 summarizes the most related prior research.
- Section 3 presents the proposed research and lays down concrete and executable research plans.
- Section 4 presents the timeline.
- Section 5 discusses expected outcomes and the impact of the proposed research.

2 Prior Research

We discuss the prior research on $\mathbb{K}$ (Section 2.1), its underlying logical foundation (Section 2.2), and how proof certificates can be generated for concrete execution of smart contracts or programs in general and allow us to obtain *certifying smart contracts on blockchain* (Section 2.3).

2.1 $\mathbb{K}$ Framework

$\mathbb{K}$ is a language framework that allows language designers to define formal syntax and semantic definition of their language, from which all implementations and tools of that language are automatically generated. The vision of $\mathbb{K}$ is depicted in Figure 1. $\mathbb{K}$ provides a simple and intuitive front-end language for language designers to define the formal syntax and semantics of other programming languages. From such a formal language definition, the framework automatically generates a set of language tools, including a parser, an

```
1  module IMP-SYNTAX
2    imports DOMAINS
3    syntax Exp ::=
4        Int
5      | Id
6      | Exp "+" Exp  [left, strict]
7      | Exp "-" Exp  [left, strict]
8      | "(" Exp ")"       [bracket]
9
10   syntax Stmt ::=
11       Id "=" Exp ";"  [strict(2)]
12     | "if" "(" Exp ")" Stmt Stmt
13                       [strict(1)]
14     | "while" "(" Exp ")" Stmt
15     | "{" Stmt "}"      [bracket]
16     | "{" "}"
17     > Stmt Stmt [left, strict(1)]
18 endmodule
```

```
19 module IMP
20   imports IMP-SYNTAX
21   syntax KResult ::= Int
22   configuration ⟨ $PGM:Stmt, ·Map ⟩
23   // Variable lookup and assignment
24   rule ⟨ C[X], M ⟩ ⇒ ⟨ C[M(X)], M ⟩
25   rule ⟨ C[X = I], M ⟩
26        ⇒ ⟨ C[{}], M[X ↦ I] ⟩
27   // Arithmetic expression
28   rule I1 + I2 ⇒ I1 + I2
29   rule I1 - I2 ⇒ I1 − I2
30   // Control flow
31   rule {} S:Stmt ⇒ S
32   rule if (I) S _ ⇒ S requires I ≠ 0
33   rule if (0) _ S ⇒ S
34   rule while (B) S
35        ⇒ if (B) { S while(B) S } {}
36 endmodule
```

Figure 2: $\mathbb{K}$ Formal Semantics of an Imperative Language IMP

interpreter, a deductive verifier, a program equivalence checker, among many others [10, 28]. $\mathbb{K}$ has obtained much success in practice, and has been used to define the complete executable formal semantics of many real-world languages, such as C [15], Java [2], JavaScript [24], Python [16], Ethereum virtual machines byte code [18], and x86-64 [12], from which their implementations and formal analysis tools are automatically generated. Some commercial products [17, 21] are powered by these auto-generated implementations and/or tools.

As an example, Figure 2, shows the complete formal definition of an imperative language IMP in $\mathbb{K}$. The definition includes both syntax (module `IMP-SYNTAX` in the left column) and formal semantics (module `IMP` in the right column). In $\mathbb{K}$, formal semantics are given as a set of *rewrite rules* of the form $lhs \Rightarrow rhs$. $\mathbb{K}$ carries out program execution by repeatedly *matching* the current configuration with the left-hand side of a rewrite rule, and then *rewriting* it to the right-hand side, until no rewrite rules can be matched further, in which case the execution terminates. The following execution step carried out variable assignment, following the rewrite rule at line 25–26

$$\langle \texttt{x = 0; x = 1;}, \cdot_{Map} \rangle \Rightarrow_{exec} \langle \texttt{\{\} x = 1;}, \texttt{x} \mapsto \texttt{0} \rangle$$

where $\Rightarrow_{exec}$ is called the rewriting relation that represents program execution.

From the $\mathbb{K}$ semantics in Figure 2, we can automatically generate language tools for IMP. Perhaps the most important tool is an interpreter, which allows us to execute concrete programs using the rewrite rules in the $\mathbb{K}$ semantics.

Example 1 (Concrete Execution)**.** Consider the program

$$\texttt{SUM}_{10} \equiv \texttt{n = 10; s = 0; while (n) \{ s = s + n; n = n - 1; \}}$$

which computes $1 + \cdots + 10$. Using the formal semantics of IMP, $\mathbb{K}$ can execute the program and return the final configuration when the execution terminates, as follows

$$\langle \texttt{SUM}_{10}, \cdot_{Map} \rangle \Rightarrow_{exec} \langle \texttt{\{\}}, \{\texttt{s} \mapsto 55, \texttt{n} \mapsto 0\} \rangle$$

Example 2 (Symbolic Execution)**.** Consider the following program with a *symbolic* integer value n

$$\texttt{SUM}(n) \equiv \texttt{n = } n\texttt{; s = 0; while (n) \{ s = s + n; n = n - 1; \}} \tag{1}$$

By (symbolically) matching and applying the rewrite rules, $\mathbb{K}$ carries out *symbolic execution*. Unlike concrete execution, symbolic execution creates *branches*. For example, when $\mathbb{K}$ encounters the `while`-loop, it splits the configuration into two branches, based on whether n is zero:

$$(\langle\ \texttt{.K}, \{\texttt{s} \mapsto 0, \texttt{n} \mapsto 0\}\ \rangle \wedge n = 0) \vee (\langle\ \texttt{UNROLLED}, \{\texttt{s} \mapsto 0, \texttt{n} \mapsto n\}\ \rangle \wedge n \neq 0) \tag{2}$$

where `.K` means the empty computation (i.e., the program has terminated), $n = 0$ and $n \neq 0$ are called *path conditions*, and `UNROLLED` is the unfolded loop:

$$\texttt{UNROLLED} \equiv \texttt{s = s + n; n = n - 1; while (n) \{ s = s + n; n = n - 1; \}}$$

Note that unless we bound the variable n, symbolic execution as above does not terminate. Instead, $\mathbb{K}$ generates a growing disjunction of branches with path conditions $n = 0, n - 1 = 0, \ldots, n - k = 0, n - k \neq 0$, for any $k \in \mathbb{N}$.

Using the same semantics for program execution, $\mathbb{K}$ allows formal deductive verification.

Example 3 (Formal Verification)**.** Consider the program $\texttt{SUM}(n)$ in Example 2. To formally specify the correctness of $\texttt{SUM}(n)$ in $\mathbb{K}$, we write the following *reachability rule*

$$\Phi_{\mathsf{sum}} \equiv \forall n.\, \langle\ \texttt{SUM}(n), \cdot_{Map}\ \rangle \wedge n \geq 0 \Rightarrow_{reach} \langle\ \texttt{.K}, \{\texttt{s} \mapsto n(n+1)/2, \texttt{n} \mapsto 0\}\ \rangle$$

where $\Rightarrow_{reach}$ is the reachability relation. Intuitively, $t_{lhs} \Rightarrow_{reach} t_{rhs}$ states that from the initial configuration t_{lhs}, the target configuration t_{rhs} is reachable in finitely many steps if the execution is terminating. Therefore, reachability relation captures the intended *partial correctness properties* of programs, supported by the classical Hoare's program logics [19]. However, unlike the Hoare's approach where each programming language has its own Hoare logic that powers formal deductive verification for that language, $\mathbb{K}$ is language-agnostic. There is one set of fixed reachability proof rules that support deductive verification for all languages. To verify the above correctness property of $\texttt{SUM}(n)$ in $\mathbb{K}$, we also need to provide an *invariant* property that summarizes the behaviors of the loop. The invariant property is also specified using a reachability rule as follows

$$\Phi_{\mathsf{inv}} \equiv \forall s.\, \forall n.\, \langle\ \texttt{WHILE}, \texttt{s} \mapsto s, \texttt{n} \mapsto n\ \rangle \wedge n \geq 0 \Rightarrow_{reach} \langle\ \texttt{.K}, \{\texttt{s} \mapsto s + n(n+1)/2, \texttt{n} \mapsto 0\}\ \rangle$$

Note that the left-hand side of the above invariant rule uses s to indicate the partial sum that has so far been computed. Then, $\mathbb{K}$'s formal verification tool takes as input both Φ_{sum} and Φ_{inv} and proves them correct automatically.

$\mathbb{K}$ scales. It has been successfully applied to defining the formal semantics of the following real-world languages, from which their execution and formal analysis tools are automatically generated.

- **Ethereum virtual machines (EVM)**. $\mathbb{K}$ has been widely applied to the emerging area of the *blockchain technology* and the formal analysis of *smart contracts*, which are self-executing financial instruments that synchronize their state on a blockchain. Blockchains and smart contracts have gained much development, and at the same time, have been marred by serious bugs and costly exploits [11, 4, 3, 30, 1], which has raised a great demand for formal methods and rigorous program analysis tools in the field.
- **C**. A complete formal semantics of C is defined in $\mathbb{K}$. This semantics powers the commercial RV-Match tool, developed and maintained by Runtime Verification Inc. (RV), founded by the second author, aiming at mathematically rigorous dynamic checking of C programs in compliance with the ISO C11 Standard. An interesting and unique contribution of the C semantics in $\mathbb{K}$ is that it captures all the *undefined behaviors* of C, which are often ignored or not taken seriously by the state of the art.

– **Java**. A complete formal semantics of Java is defined in $\mathbb{K}$, which covers all language features and is extensively tested on a test suite, which is developed in parallel with the semantics in a test-driven manner. The test suite itself is an important outcome because at the time Java appeared to have no publicly conformance test suite.

– **JavaScript**. A complete formal semantics of JavaScript is defined in $\mathbb{K}$, which is thoroughly tested on the ECMAScript 5.1 conformance test suite. The semantics naturally yields a *coverage metric for test suites*, which is the set of $\mathbb{K}$ *semantic rules* that a test suite exercises. As it turned out, the ECMAScript 5.1 test suite was *incomplete* with respect to the semantics-based coverage metric because it did not trigger the application of some semantic rules for JavaScript. New tests were written to cover those semantic rules and found bugs in production JavaScript engines such as Chrome V8, Safari WebKit, and Firefox SpiderMonkey at the time.

– **Python**. A complete formal semantics of Python 3.2 is defined in $\mathbb{K}$, as one of the first efforts in demonstrating the scalability of $\mathbb{K}$. The semantics automatically yields an interpreter for program execution as well as formal analysis tools for state space exploration, static reasoning, and formal program verification. The autogenerated interpreter for Python performs as efficiently as CPython [25], which is the reference implementation of Python.

– **Rust**. As an emerging system programming language, Rust runs blazingly fast, prevents segfaults, and guarantees thread safety [22]. Nevertheless, a formal semantics is still needed to formally verify Rust programs. There are two parallel efforts to define the formal semantics of Rust in $\mathbb{K}$, both have carried out experiments on symbolic execution and formal verification for Rust using $\mathbb{K}$.

– **x86-64**. Not just high-level programming languages, low-level assembly languages can also be given a formal semantics using $\mathbb{K}$. One example is the formal semantics of x86-64 that defines all the non-deprecated, sequential user-level instructions of the x86-64 Haswell instruction set architecture. The semantics is fully executable, and has been tested on thousands of instruction-level test cases as well as the GCC torture test suite, from which bugs have been found in both the x86-64 reference manual and other (non-executable) formal semantics of x86-64.

$\mathbb{K}$ has been used a lot in blockchain applications and their formal analyses. The $\mathbb{K}$ semantics of EVM, called KEVM, is adopted as a replacement for the Yellow Paper [31] by the community. KEVM has powered the commercial toolkit called Firefly that aims at improving the quality of Ethereum smart contracts. Firefly consists of automatic and interactive tools for unit testing, integration testing, black-box/white-box random testing, runtime monitoring, as well as formal analysis tools such as a assertion violation checker, a bounded model checker, and symbolic testing. KEVM has also been adopted by DappHub, the Ethereum Foundation, Gnosis, MakerDAO, and Uniswap as the semantic basis for smart contract verification. $\mathbb{K}$ has automatically generated an EVM interpreter (from the formal semantics KEVM) that *runs faster* than many hand-written reference implementations of EVM, which suggests that it is realistic to generate virtual machines for blockchains from their formal semantics because performance is no longer a main obstacle issue.

2.2 Matching Logic: Logical Foundation of $\mathbb{K}$

Matching logic is a unifying logic for specifying and reasoning about programs in a compact and modular way [26, 7, 8, 6]. Matching logic is the logical foundation of $\mathbb{K}$, in the sense that every formal programming language semantics defined in $\mathbb{K}$ can be translated to a matching logic theory, which consists of a set of axioms that capture the formal semantics. All $\mathbb{K}$ tools can then be specified by matching logic formulas. Specifically,

1. The $\mathbb{K}$ semantics of a programming language L (such as Figure 1) corresponds to a *matching logic theory* Γ^L, which, roughly speaking, consists of a set of logical symbols that represents the formal syntax of L, and a set of logical axioms that specify the formal semantics.

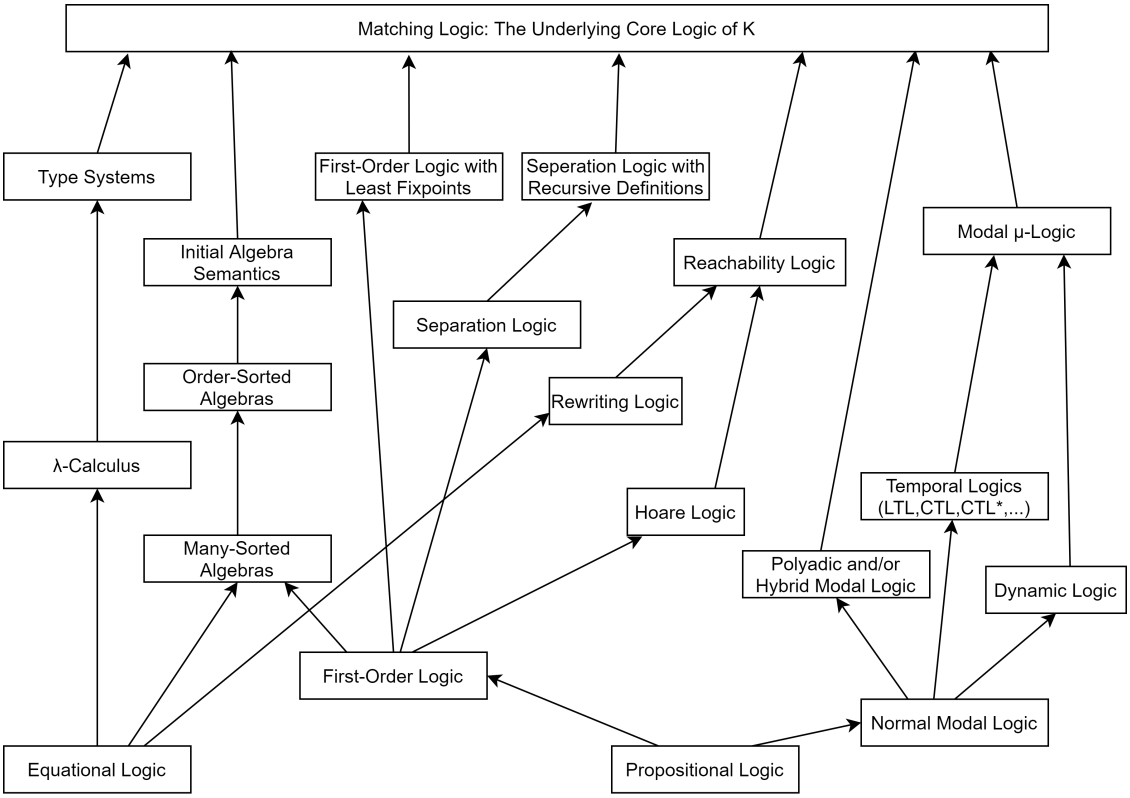

Figure 3: Matching Logic is Expressive

2. All language tools in Figure 1 and all language tasks that $\mathbb{K}$ conducts are formally specified by matching logic formulas. For example, *program execution* is specified by

$$\varphi_{init} \Rightarrow_{exec} \varphi_{final} \tag{3}$$

where φ_{init} is the formula that specifies the initial state of the execution, φ_{final} specifies the final state, and "$\Rightarrow_{exec}$" states the rewriting relation between states.

3. There exists a matching logic *proof system* that defines the provability relation $\vdash$ between theories and formulas. For example, the correctness of the above execution from φ_{init} to φ_{final} is witnessed by the formal proof:

$$\Gamma^L \vdash \varphi_{init} \Rightarrow_{exec} \varphi_{final} \tag{4}$$

Therefore, matching logic is the logical foundation of $\mathbb{K}$. The *correctness* of $\mathbb{K}$ conducting one language task is reduced to the *existence of a formal proof* in matching logic. Such formal proofs are encoded as proof objects, which can be automatically checked by the matching logic proof checker.

Matching logic is an expressive logic. Many important logical systems and/or formalisms can be defined as matching logic theories. Figure 3 summarizes the most common and important logical systems that are definable by matching logic. With such expressive power, matching logic can serve a powerful specification language that allows us to specify both systems and their properties.

The matching logic proof system as shown in Figure 4 allows us to formally derive valid formulas φ from any given theory Γ. The provability relation is denoted $\Gamma \vdash \varphi$, read as "φ is derivable/provable from Γ". Roughly speaking, the 15 proof rules in Figure 4 can be divided into four categories: first-order reasoning, frame reasoning, fixpoint reasoning, and a few miscellaneous proof rules. Fixpoint reasoning is

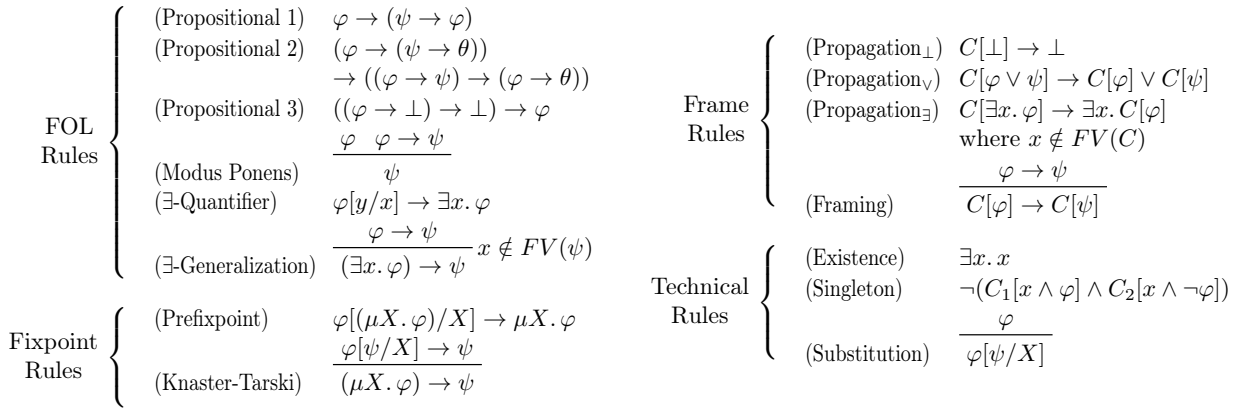

Figure 4: The 15-Rule Matching Logic Proof System, Implementable in 240 Lines of Code

particularly important in the proposed research, because it is the logical foundation of inductive/recursive and co-inductive/co-recursive reasoning. $\mathbb{K}$'s verification tool is based on coinductive reasoning, which can be reduced to the more basic matching logic proof rules for fixpoint reasoning. This way, we reduce the complex and error-prone verification algorithms of $\mathbb{K}$ into simpler, machine-checkable matching logic proofs.

Matching logic and its 15-rule proof system has been completely formalized in Metamath [23]—a tiny language to state abstract mathematics and their proofs in a machine-checkable style. Therefore, matching logic proof objects are encoded in Metamath which can then be checked by the Metamath verifiers.

The formalization of matching logic in Metamath has only 240 lines of code `https://github.com/kframework/matching-logic-proof-checker/blob/main/theory/matching-logic-240-loc.mm`. Figure 5 shows an extract of the 240-line formalization and an example matching logic theorem with its derivation, all encoded in Metamath and checkable by the Metamath verifiers. In line 1, we declare five constants, where `\imp`, `(`, and `)` build the syntax of matching logic, `#Pattern` is the type of matching logic patterns (i.e., formulas), and `|-` is the provability relation. We declare three meta-variables of patterns in lines 3-6, and the syntax of implication $\varphi_1 \to \varphi_2$ as `( \imp ph1 ph2 )` in line 7. Then, we define matching logic proof rules as Metamath axioms. For example, lines 18-22 define the rule (Modus Ponens). In line 23, we show an example (meta-)theorem and its formal proof in Metamath. The theorem states that $\vdash \varphi_1 \to \varphi_1$ holds, and its proof (lines 25-43) is a sequence of labels referring to the previous axiomatic/provable statements.

2.3 Certifying Smart Contracts on Blockchain

[5] presents the prior research on generating proof certificates for concrete program execution that is carried out using $\mathbb{K}$. For each concrete run of a program, a proof certificate is generated, which proves that there indeed exists, according to the formal language semantics, an execution trace with the specified initial and final states. In other words, ***computations are proofs***.

With proof certificates for concrete program executions, we can achieve *certifying smart contracts on blockchain*, where each concrete execution of the smart contracts are certified by proof certificates. Validators do not need to re-execute the code or transactions but only need to check the proof certificates using the 240-line proof checker. In other words, the only program that is running on-chain is the proof checker.

In the following, we highlight the main technical steps of [5] and present experimental results on the performance of the proposed proof generation method.

A proof certificate (i.e., matching logic proof object) for concrete program execution consists of the following:

1. the formalization of matching logic and its provability relation $\vdash$;
2. the formalization of the formal semantics Γ^L as a logical theory, which includes axioms that specify the rewrite/semantic rules $\varphi_{lhs} \Rightarrow_{exec} \varphi_{rhs}$;

```
$c \imp ( ) #Pattern |- $.

$v ph1 ph2 ph3 $.
ph1-is-pattern $f #Pattern ph1 $.
ph2-is-pattern $f #Pattern ph2 $.
ph3-is-pattern $f #Pattern ph3 $.
imp-is-pattern
  $a #Pattern ( \imp ph1 ph2 ) $.

axiom-1
  $a |- ( \imp ph1 ( \imp ph2 ph1 ) ) $.

axiom-2
  $a |- ( \imp ( \imp ph1 ( \imp ph2 ph3 ) )
       ( \imp ( \imp ph1 ph2 )
              ( \imp ph1 ph3 ) ) ) $.

${
  rule-mp.0 $e |- ( \imp ph1 ph2 ) $.
  rule-mp.1 $e |- ph1 $.
  rule-mp   $a |- ph2 $.
$}
```

```
imp-refl $p |- ( \imp ph1 ph1 )
$=
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  imp-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern imp-is-pattern
  imp-is-pattern ph1-is-pattern
  ph1-is-pattern ph1-is-pattern
  imp-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  imp-is-pattern imp-is-pattern
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  ph1-is-pattern axiom-2
  ph1-is-pattern ph1-is-pattern
  ph1-is-pattern imp-is-pattern
  axiom-1 rule-mp ph1-is-pattern
  ph1-is-pattern axiom-1 rule-mp
$.
```

Figure 5: An Extract of the 240-line Formalization of Matching Logic

3. the formal proofs of all one-step executions, i.e., $\Gamma^L \vdash \varphi_i \Rightarrow_{exec} \varphi_{i+1}$ for all i;
4. the formal proof of the final proof goal $\Gamma^L \vdash \varphi_{init} \Rightarrow_{exec} \varphi_{final}$.

Thus, the proof objects have a *linear structure*, which implies a nice separation of concerns. Indeed, Item 1 is only about matching logic and is *not specific* to any programming languages/language tasks, so we only need to develop and proof-check it *once and for all*. Item 2 is specific to the language semantics Γ^L but is independent of the actual program executions, so it can be reused in the proof objects of various language executions for the same programming language L.

The key step in generating proof certificates for concrete program execution is the generation of the proof objects for *one-step executions*, which are then put together to build the proof objects for multi-step executions using the transitivity of the rewriting relation. Suppose the language semantics consist of rewrite rules

$$S = \{t_k \wedge p_k \Rightarrow s_k \mid k = 1, 2, \ldots, K\}$$

where t_k and s_k are the left- and right-hand sides of the rewrite rule, respectively, and p_k is the rewriting condition. Consider the execution trace

$$\varphi_0, \varphi_1, \ldots, \varphi_n \tag{5}$$

where $\varphi_0, \ldots, \varphi_n$ are snapshots. Firstly, we let $\mathbb{K}$ generate the following proof parameter:

$$\Theta \equiv (k_0, \theta_0), \ldots, (k_{n-1}, \theta_{n-1}) \tag{6}$$

where for each $0 \leq i < n$, k_i denotes the rewrite rule that is applied on φ_i ($1 \leq k_i \leq K$) and θ_i denotes the

Table 1: Performance of Proof Generation/Checking for Concrete Program Execution (time in seconds)

programs	proof generation			proof checking			proof size	
	sem	rewrite	total	logic	task	total	kLOC	MB
`10.two-counters`	5.95	12.19	18.13	3.26	0.19	3.44	963.8	77
`20.two-counters`	6.31	24.33	30.65	3.41	0.38	3.79	1036.5	83
`50.two-counters`	6.48	73.09	79.57	3.52	0.98	4.50	1259.2	100
`100.two-counters`	6.75	177.55	184.30	3.50	2.10	5.60	1635.6	130
`add8`	11.59	153.34	164.92	3.40	3.09	6.48	1986.8	159
`factorial`	3.84	34.63	38.46	3.57	0.90	4.47	1217.9	97
`fibonacci`	4.50	12.51	17.01	3.44	0.21	3.65	971.7	77
`benchexpr`	8.41	53.22	61.62	3.61	0.80	4.41	1191.3	95
`benchsym`	8.79	47.71	56.50	3.53	0.72	4.25	1163.4	93
`benchtree`	8.80	26.86	35.66	3.47	0.32	3.80	1021.5	81
`langton`	5.26	23.07	28.33	3.46	0.40	3.86	1048.0	84
`mul8`	14.39	279.97	294.36	3.48	7.18	10.66	3499.2	280
`revelt`	4.98	51.83	56.81	3.35	1.10	4.45	1317.4	105
`revnat`	4.81	123.44	128.25	3.37	5.28	8.65	2691.9	215
`tautologyhard`	5.16	400.89	406.05	3.55	14.50	18.04	6884.7	550

corresponding substitution such that $t_{k_i}\theta_i = \varphi_i$. The following lists all the one-path proof objects.

$$
\begin{array}{ll}
\Gamma^L \vdash \varphi_0 \Rightarrow s_{k_0}\theta_0 & // \text{ by applying } t_{k_0} \wedge p_{k_0} \Rightarrow s_{k_0} \text{ using } \theta_0 \\
\Gamma^L \vdash s_{k_0}\theta_0 = \varphi_1 & // \text{ by simplifying } s_{k_0}\theta_0 \\
\quad \cdots & \\
\Gamma^L \vdash \varphi_{n-1} \Rightarrow s_{k_{n-1}}\theta_{n-1} & // \text{ by applying } t_{k_{n-1}} \wedge p_{k_{n-1}} \Rightarrow s_{k_{n-1}} \text{ using } \theta_{n-1} \\
\Gamma^L \vdash s_{k_{n-1}}\theta_{n-1} = \varphi_n & // \text{ by simplifying } s_{k_{n-1}}\theta_{n-1}
\end{array}
$$

For rewrite-rule proofs $\Gamma^L \vdash \varphi_i \Rightarrow s_{k_i}\theta_i$, the main steps are (1) instantiating $t_{k_i} \wedge p_{k_i} \Rightarrow s_{k_i}$ by

$$\theta_i = [c_1/x_1, \ldots, c_m/x_m]$$

provided by the proof parameter, and (2) proving that the rewriting condition $p_{k_i}\theta_i$ holds. For simplification proofs, suppose s is a term and $p \rightarrow t = t'$ is a (conditional) equation. Then, s can be simplified w.r.t. $p \rightarrow t = t'$, if there is a sub-term s_0 of s (written $s \equiv C[s_0]$ where C is a context) and a substitution θ such that $s_0 = t\theta$ and $p\theta$ holds. The simplification result is denoted $C[t'\theta]$. Therefore, a proof object of the above simplification consists of two proofs: $\Gamma^L \vdash s = C[t'\theta]$ and $\Gamma^L \vdash p\theta$, where the latter can be handled recursively, by simplifying $p\theta$ to $\top$.

The prototype implementation [20] for concrete program execution proof generation is evaluated using the benchmarks from REC [14], which is a popular performance benchmark for rewriting engines. We evaluate both the performance of proof object *generation* and that of proof *checking*. The experimental results are as shown in Table 1. In summary,

1. Proof checking is efficient and takes a few seconds; in particular, the *task-specific* checking time is often less than one second ("task" column in Table 1).

2. Proof object generation is slower and takes several minutes.

3. Proof objects are huge, often of millions line of code.

3 Proposed Research

The proposed research aims to generate matching logic proof objects as proof certificates for $\mathbb{K}$'s formal verification tool. The proposed work can be divided into three parts:

1. Generating proof certificates for symbolic program execution.
2. Generating proof certificates for formal verification.
3. Generating proof certificates for nondeterministic programs (i.e., supporting all-path verification).

In the following, we first give an overview of the $\mathbb{K}$ verification tool. Then, we discuss each of the above research topics and provide concrete and detailed research plans for them.

The $\mathbb{K}$ verification tool was firstly proposed and implemented by Ştefănescu *et al.* as a language-agnostic deductive verifier based on *reachability logic* [10]. Recently, it is further shown in [7] that reachability logic can be embedded into matching logic. Therefore, we can generate matching logic proof objects as proof certificates that certify the correctness of each deductive verification task done by $\mathbb{K}$. Consider the following `SUM` program as in Equation (1)

$$\texttt{SUM}(n) \equiv \texttt{ n = } n\texttt{; s = 0;while (n) \{ s = s + n; n = n - 1; \}}$$

whose correctness can be formalized by the following matching logic proof goal:

$$A \vdash^{reach}_{\emptyset} \langle\, \texttt{SUM}(n), \cdot_{Map} \,\rangle \Rightarrow_{reach} \langle\, \texttt{\{\}}, \{\texttt{s} \mapsto \Sigma_{i=1}^{n} i, \texttt{n} \mapsto 0\} \,\rangle \tag{7}$$

During the proof, we also generalize and prove the following claim using coinduction:

$$A \vdash^{reach}_{\emptyset} \langle\, \texttt{LOOP}, \{\texttt{s} \mapsto s, \texttt{n} \mapsto n\} \,\rangle \Rightarrow_{reach} \langle\, \texttt{\{\}}, \{\texttt{s} \mapsto s + \Sigma_{i=1}^{n} i, \texttt{n} \mapsto 0\} \,\rangle \tag{8}$$

To prove the correctness of `SUM` in $\mathbb{K}$, we write both proof goals in a $\mathbb{K}$ specification and then run $\mathbb{K}$'s verification algorithm to check the correctness of the specification.

The internal verification algorithm in $\mathbb{K}$ is shown in Algorithm 1. $\mathbb{K}$ takes the set of reachability claims R given by the user (containing both the main goal and auxiliary coinduction hypotheses), and for each reachability claim $\varphi_1 \Rightarrow_{reach} \varphi_2 \in R$, $\mathbb{K}$ performs symbolic execution from φ_1 until all branches are subsumed by the right-hand side φ_2 (checked by line 7). However, the difference is that after the first step of symbolic execution (line 3), we can apply rules in R as well (line 8), which is justified by coinduction.

Next, we discuss how these correctness proofs are generated in detail.

3.1 Generating Proof Certificates for Symbolic Execution

Let Γ^L be the matching logic theory that defines the formal definition of a programming language L.

Problem Formulation

Consider the following $\mathbb{K}$ language definition consisting of K (conditional) rewrite rules:

$$\{lhs_k \wedge q_k \Rightarrow^1_{exec} rhs_k \mid k = 1, 2, \ldots, K\} \subseteq \Gamma^L$$

where lhs_k represents the left-hand side of the rewrite rule, rhs_k represents the right-hand side, and q_k denotes the rewriting condition. For unconditional rules, q_k is $\top$. The notation $\Rightarrow^1_{exec}$ stands for one-step execution, defined in Section 2.2.

In symbolic execution, program configurations often appear with their corresponding *path conditions*. We represent them as $t \wedge p$, where t is a configuration and p is a logical constraint/predicate over the free variables of t. We call such patterns *constrained terms*. Constrained terms are matching logic patterns.

Unlike concrete execution, symbolic execution can create *branches*. Therefore, we formulate proof generation for symbolic execution as follows. The *input* is an initial constrained term $t \wedge p$ and a list of final

```
// Checks the validity of claims in R
// using symbolic execution and coinduction.
1  procedure checkReachability(R)
2      for φ ⇒_reach ψ ∈ R do
3          Q ← successors(φ) ;
4          if Q = ∅ and Γ^L ⊬ φ → ψ then fail;
5          while Q ≠ ∅ do
6              Pop any φ′ from Q ;
7              if Γ^L ⊢ φ′ → ψ then continue;
8              Q′ ← successors_R(φ′) ;
9              if Q′ = ∅ then fail;
10             else Q ← Q ∪ Q′ ;
```

Algorithm 1: Verification algorithm in $\mathbb{K}$ [10]. Here, $\texttt{successors}(\varphi)$ is a set of patterns that are the results of symbolically executing φ for one step using the formal semantics (see Section 3.1). $\texttt{successors}_I(\varphi)$ is the result of applying a rule in R if any one is applicable, otherwise we let $\texttt{successors}_R(\varphi) = \texttt{successors}(\varphi)$.

constrained terms $t_1 \wedge p_1, \ldots, t_n \wedge p_n$, which are returned by $\mathbb{K}$ as the result(s) of symbolic executing t under the condition p. Each $t_i \wedge p_i$ represents one possible execution trace. Our *goal* is to generate a proof object for the following proof goal:

$$\Gamma^L \vdash t \wedge p \Rightarrow_{exec} (t_1 \wedge p_1) \vee \cdots \vee (t_n \wedge p_n) \tag{Goal}$$

In other words, using the notations in Algorithm 1, we need to show the correctness of `successors` by proving $\Gamma^L \vdash \varphi \Rightarrow_{exec} \texttt{successors}(\varphi)$, which further implies $\Gamma^L \vdash \varphi \Rightarrow_{reach} \texttt{successors}(\varphi)$.

Proof Hints

To help generate the proof of (Goal), we instrument $\mathbb{K}$ to output *proof hints*, which include rewriting details, such as which semantic rules are applied and what substitutions are used. Formally, the proof hint for the j-th rewrite step consists of:

- a constrained term $t_j^{\text{hint}} \wedge p_j^{\text{hint}}$ before step j;
- l_j constrained terms $t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}}, .., t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}$ after step j, where for each $1 \leq l \leq l_j$, we also annotate the term with the index $1 \leq k_{j,l} \leq K$ of a rewrite rule and a substitution $\theta_{j,l}$;
- an (optional) constrained term $t_j^{\text{rem}} \wedge p_j^{\text{rem}}$, called the *remainder* of step j.

Intuitively, each constrained term $t_{j,l}^{\text{hint}} \wedge p_{j,l}^{\text{hint}}$ represents one execution branch, obtained by applying the $k_{j,l}$-th rewrite rule (i.e., $lhs_{k_{j,l}} \wedge q_{k_{j,l}} \Rightarrow^1_{exec} rhs_{k_{j,l}}$) with substitution $\theta_{j,l}$. The remainder $t_j^{\text{rem}} \wedge p_j^{\text{rem}}$ denotes the branch where no rewrite rules can be applied further and thus the execution gets stuck. Note that t_j^{hint} and t_j^{rem} may look different even if no rewrite step is made. This is because the path condition p_j^{rem} may be stronger than the original condition p_j^{hint}. With this stronger path condidtion, $\mathbb{K}$ may be able to simplify t_j^{hint} to a different term t_j^{rem}.

From the above proof hint, we can generate a proof for the symbolic execution step. For example, the following is the claim for the j-th symbolic execution step:

$$\Gamma^L \vdash \left(t_j^{\text{hint}} \wedge p_j^{\text{hint}}\right) \Rightarrow_{exec} \left(t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}}\right) \vee \ldots \vee \left(t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}\right) \vee \left(t_j^{\text{rem}} \wedge p_j^{\text{rem}}\right) \tag{Step$_j$}$$

Recall that $\Rightarrow_{exec}$ is the *reflexive*-transitive closure of one-step execution, so we can have the remainder configuration at the right-hand side even if no execution is made. To prove (Step$_j$), we need to prove each execution branch: for $1 \leq l \leq l_j$,

$$\Gamma^L \vdash \left(t_j^{\text{hint}} \wedge p_{j,l}^{\text{hint}}\right) \Rightarrow^1_{exec} \left(t_{j,l}^{\text{hint}} \wedge p_{j,l}^{\text{hint}}\right) \tag{Branch$_{j,l}$}$$

For the remainder branch, we need to prove that:

$$\Gamma^L \vdash \left(t_j^{\text{hint}} \wedge p_j^{\text{rem}}\right) \to \left(t_j^{\text{rem}} \wedge p_j^{\text{rem}}\right) \tag{Remainder$_j$}$$

Proof Generation

Given the above proof hints, we prove (Goal) in three phases:

Phase 1. We prove (Branch$_{j,l}$) and (Remainder$_j$) for each step j and branch $1 \leq l \leq l_j$.

Phase 2. We combine (Branch$_{j,l}$) and (Remainder$_j$) to obtain a proof of (Step$_j$).

Phase 3. We combine (Step$_j$) to prove (Goal).

Remark 1 (Lemmas and Their Proofs)**.** We need many lemmas about program execution "$\Rightarrow_{exec}$" when we generate the proof objects for symbolic execution. The most important and relevant lemmas are stated explicitly in this paper. In total, 196 new lemmas are formally encoded, and their proofs have been completely worked out based on the 240-line Metamath database of matching logic [5]. These lemmas can be easily reused for future development.

In the following, we explain each proof generation step.

Phase 1: Proving (Branch$_{j,l}$) **and** (Remainder$_j$)**.** Recall that (Branch$_{j,l}$) is obtained by applying the $k_{j,l}$-th rewrite rule from the language semantics (where $1 \leq k_{j,l} \leq K$):

$$lhs_{k_{j,l}} \wedge q_{k_{j,l}} \Rightarrow^1_{exec} rhs_{k_{j,l}}$$

According to the proof hint, the corresponding substitution is $\theta_{j,l}$. Therefore, by instantiating the rewrite rule with $\theta_{j,l}$, we obtain the following proof:

$$\Gamma^L \vdash lhs_{k_{j,l}}\theta_{j,l} \wedge q_{k_{j,l}}\theta_{j,l} \Rightarrow^1_{exec} rhs_{k_{j,l}}\theta_{j,l} \tag{9}$$

Since the condition $q_{k_{j,l}}\theta_{j,l}$ is a predicate on the free variables of Equation (9) and it holds on the left-hand side, it also holds on the right-hand side. Therefore, we prove that:

$$\Gamma^L \vdash lhs_{k_{j,l}}\theta_{j,l} \wedge q_{k_{j,l}}\theta_{j,l} \Rightarrow^1_{exec} rhs_{k_{j,l}}\theta_{j,l} \wedge q_{k_{j,l}}\theta_{j,l} \tag{10}$$

To proceed with the proof, we need the following lemma:

Lemma 1 ($\Rightarrow^1_{exec}$ Consequence)**.**

$$\frac{\Gamma^L \vdash \varphi \to \varphi' \quad \Gamma^L \vdash \varphi' \Rightarrow^1_{exec} \psi' \quad \Gamma^L \vdash \psi' \to \psi}{\Gamma^L \vdash \varphi \Rightarrow^1_{exec} \psi}$$

Intuitively, Lemma 1 allows us to strengthen the left-hand side and/or weaken the right-hand side of an execution relation. Using Lemma 1, and by comparing our proof goal (Branch$_{j,l}$) with Equation (10), we only need to prove the following two implications, called *subsumptions*:

$$\underbrace{\Gamma^L \vdash \left(t_j^{\text{hint}} \wedge p_{j,l}^{\text{hint}}\right) \to \left(lhs_{k_{j,l}}\theta_{k_{j,l}} \wedge q_{k_{j,l}}\theta_{k_{j,l}}\right)}_{\text{left-hand side strengthening}}$$

$$\underbrace{\Gamma^L \vdash \left(rhs_{k_{j,l}}\theta_{k_{j,l}} \wedge q_{k_{j,l}}\theta_{k_{j,l}}\right) \to \left(t_{j,l}^{\text{hint}} \wedge p_{j,l}^{\text{hint}}\right)}_{\text{right-hand side weakening}}$$

These subsumption proofs are common in our proof generation procedure (e.g. (Remainder$_j$) is also a subsumption), whose detailed proofs will be generated separately.

Phase 2: Proving (Step_j)**.** The proof goal (Step_j) is proved by combining the proofs for each branch and the remainder:

$$\Gamma^L \vdash t_j^{\text{hint}} \wedge p_{j,1}^{\text{hint}} \Rightarrow_{exec}^{1} t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}} \qquad (\text{Branch}_{j,1})$$

$$\vdots$$

$$\Gamma^L \vdash t_j^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}} \Rightarrow_{exec}^{1} t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}} \qquad (\text{Branch}_{j,l_j})$$

$$\Gamma^L \vdash t_j^{\text{hint}} \wedge p_j^{\text{rem}} \rightarrow t_j^{\text{rem}} \wedge p_j^{\text{rem}} \qquad (\text{Remainder}_j)$$

Note that our proof goal (Step_j) uses "$\Rightarrow_{exec}$", while the above use either one-step execution ("$\Rightarrow_{exec}^{1}$") or implication ("$\rightarrow$"). The following lemma allows us to turn one-step execution and implication (i.e. "zero-step execution") into the reflexive-transitive execution relation "$\Rightarrow_{exec}$":

Lemma 2 ($\Rightarrow_{exec}$ Introduction)**.**

$$\frac{\Gamma^L \vdash \varphi \rightarrow \psi}{\Gamma^L \vdash \varphi \Rightarrow_{exec} \psi} \qquad \frac{\Gamma^L \vdash \varphi \Rightarrow_{exec}^{1} \psi}{\Gamma^L \vdash \varphi \Rightarrow_{exec} \psi}$$

Then, we need to verify that the disjunction of all path conditions in the branches (including the remainder) is implied from the initial path condition:

$$\Gamma^L \vdash p_j^{\text{hint}} \rightarrow p_{j,1}^{\text{hint}} \vee \cdots \vee p_{j,l_j}^{\text{hint}} \vee p_j^{\text{rem}} \tag{11}$$

The above implication includes only logical constraints and no configuration terms, and thus involves only *domain reasoning*. Therefore, we simply translate it into an equivalent FOL formula and delegate it to SMT solvers, such as Z3 [13].

From Equation (11), we can prove that the left-hand side of (Step_j), $t_j^{\text{hint}} \wedge p_j^{\text{hint}}$, can be broken down into $l_j + 1$ branches by propositional reasoning:

$$\Gamma^L \vdash \left(t_j^{\text{hint}} \wedge p_j^{\text{hint}}\right) \rightarrow \left(t_j^{\text{hint}} \wedge p_{j,1}^{\text{hint}}\right) \vee \ldots \vee \left(t_j^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}\right) \vee \left(t_j^{\text{hint}} \wedge p_j^{\text{rem}}\right) \tag{12}$$

Note that that right-hand side of Equation (12) is exactly the disjunction of all the left-hand sides of ($\text{Branch}_{j,l}$) and (Remainder_j). Therefore, to prove the proof goal (Step_j), we use the following lemma, which allows us to combine the executions in different branches into one:

Lemma 3 ($\Rightarrow_{exec}$ Merge)**.**

$$\frac{\Gamma^L \vdash \varphi_1 \Rightarrow_{exec} \psi_1 \quad \ldots \quad \Gamma^L \vdash \varphi_n \Rightarrow_{exec} \psi_n}{\Gamma^L \vdash \bigvee_{i=1}^{n} \varphi_i \Rightarrow_{exec} \bigvee_{i=1}^{n} \psi_i}$$

Phase 3: Proving (Goal)**.** We are now ready to prove our final proof goal for symbolic execution. At a high level, the proof simply uses the reflexivity and transitivity of the program execution relation $\Rightarrow_{exec}$. Therefore, our proof generation method is an iterative procedure. We start with the reflexivity of $\Rightarrow_{exec}$, that is:

$$\Gamma^L \vdash (t \wedge p) \Rightarrow_{exec} (t \wedge p) \tag{13}$$

Then, we repeatedly apply the following steps to *symbolically execute* the right-hand side of Equation (13), until it becomes the same as the right-hand side of (Goal):

1. Suppose we have established a proof of

$$\Gamma^L \vdash (t \wedge p) \Rightarrow_{exec} \left(t_1^{\text{im}} \wedge p_1^{\text{im}}\right) \vee \cdots \vee \left(t_m^{\text{im}} \wedge p_m^{\text{im}}\right) \tag{14}$$

where t_1^{im}, p_1^{im}, etc. represent the **i**nter**m**ediate configurations and constraints, respectively.

2. Look for a (Step_j) claim of the form

$$\Gamma^L \vdash \left(t_j^{\text{hint}} \wedge p_j^{\text{hint}}\right) \Rightarrow_{exec} \left(t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}}\right) \vee \ldots \vee \left(t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}\right) \vee \left(t_j^{\text{rem}} \wedge p_j^{\text{rem}}\right) \tag{Step_j}$$

such that $t_j^{\text{hint}} \wedge p_j^{\text{hint}} \equiv t_i^{\text{im}} \wedge p_i^{\text{im}}$, for some intermediate constrained term $t_i^{\text{im}} \wedge p_i^{\text{im}}$. Without loss of generality, let us assume that $i = 1$, i.e., it is the first intermediate constrained term $t_1^{\text{im}} \wedge p_1^{\text{im}}$ that can be rewritten/executed using (Step_j).

3. Execute $t_1^{\text{im}} \wedge p_1^{\text{im}}$ in Equation (14) for one step using (Step_j), and obtain the following proof:

$$\Gamma^L \vdash (t \wedge p) \Rightarrow_{exec} \underbrace{\left(t_{j,1}^{\text{hint}} \wedge p_{j,1}^{\text{hint}}\right) \vee \cdots \vee \left(t_{j,l_j}^{\text{hint}} \wedge p_{j,l_j}^{\text{hint}}\right) \vee \left(t_j^{\text{rem}} \wedge p_j^{\text{rem}}\right)}_{\text{right-hand side of } (\text{Step}_j)}$$
$$\vee \underbrace{\left(t_2^{\text{im}} \wedge p_2^{\text{im}}\right) \vee \ldots \vee \left(t_m^{\text{im}} \wedge p_m^{\text{im}}\right)}_{\text{same as Equation (14)}}$$

Finally, when all symbolic execution steps are applied, we check if the resulting proof goal is the same as (Goal), potentially after permuting the disjuncts on the right-hand side. If yes, then the proof generation method succeeds and we obtain a proof of the goal (Goal). Otherwise, the proof generation method fails, indicating potential mistakes made by $\mathbb{K}$'s symbolic execution engine.

3.2 Generating Proof Certificates for Formal Deductive Verification

Recall that the verification algorithm in Algorithm 1 performs symbolic execution from the left-hand side of each claim until all branches are subsumed by the right-hand side. While the proof generation procedures in previous sections can cover symbolic execution already, the missing part is line 8 in Algorithm 1, where we also apply claims in R itself to perform symbolic execution. This may seem like a circular argument, but the algorithm is in fact performing a coinduction on the rewriting trace. In this section, we describe how we can generate proof objects to justify this coinduction step.

We start with the simplest case when R has only one claim $\varphi \Rightarrow_{reach} \psi$. We assume that we have already rewritten φ to some intermediate configuration φ' in one or more steps:

$$\varphi \Rightarrow^{+}_{reach} \varphi' \tag{15}$$

Then if the proof hints indicate that we need to apply the original claim $\varphi \Rightarrow_{reach} \psi$ itself to φ' as a coinduction hypothesis, we first generate a proof for this single step:

$$\Box(\forall FV(\varphi, \psi).\, \varphi \Rightarrow_{reach} \psi) \rightarrow \varphi' \Rightarrow_{reach} \varphi'' \tag{16}$$

by instantiating the quantifiers $\forall FV(\varphi, \psi)$ using the substitution contained in the proof hints, where φ'' is the result of applying the claim $\varphi \Rightarrow_{reach} \psi$ as a rewrite rule on φ'. Recall that this is the encoding of the reachability judgment $\{\varphi \Rightarrow_{reach} \psi\} \vdash^{reach}_{\emptyset} \varphi' \Rightarrow \varphi''$.

Now by (Transitivity) applied on Equations (15) and (16), we can get a proof of

$$\circ\Box(\forall FV(\varphi, \psi).\, \varphi \Rightarrow_{reach} \psi) \rightarrow \varphi \Rightarrow^{+}_{reach} \varphi''$$

which is the encoding of the reachability judgment $\vdash^{reach}_{\{\varphi \Rightarrow_{reach} \psi\}} \varphi \Rightarrow \varphi''$. Then we can continue the symbolic execution of φ'' using the previous proof generation procedure except with an additional circularity premise $\circ\Box(\forall FV(\varphi, \psi).\, \varphi \Rightarrow_{reach} \psi)$.

Finally, if the entire reachability proof can be eventually done, we would have obtained a proof of

$$\circ\Box(\forall FV(\varphi, \psi).\, \varphi \Rightarrow_{reach} \psi) \rightarrow \varphi \Rightarrow_{reach} \psi$$

which by (Circularity), gives us $\varphi \Rightarrow_{reach} \psi$ as desired.

In general, we could have n claims in $R = \{\varphi_1 \Rightarrow_{reach} \psi_1, \ldots, \varphi_n \Rightarrow_{reach} \psi_n\}$ and their proofs could arbitrarily invoke each other's coinduction hypothesis. This can be justified by a generalization of (Circularity) [27, Lemma 5]:

$$\text{(Set Circularity)} \quad \frac{A \vdash_R^{reach} \varphi \Rightarrow \psi \quad \text{for all } (\varphi \Rightarrow \psi) \in R}{A \vdash_\emptyset^{reach} \varphi \Rightarrow \psi \quad \text{for all } (\varphi \Rightarrow \psi) \in R}$$

That is, we can simultaneously add all claims in R as coinduction hypotheses to do a mutual coinduction.

In our current implementation, however, we do not implement the proof of (Set Circularity) in its full generality, and we make the assumption that the proof of each claim would only invoke its own coinduction hypothesis. This is not a restriction in theory, because Roşu *et al.* [27, Lemma 5] has shown that any proof using (Set Circularity) can be reduced to one using only (Circularity).

3.3 Supporting Nondeterminism and All-Path Verification

In our current work, we consider only *one-path reachability*. In other words, $\varphi \Rightarrow_{reach} \psi$ holds if φ diverges or has *one* finite execution that reaches ψ. While one-path reachability is sufficient for the verification of deterministic programs, it does not support nondeterministic or concurrent programs, where we need to verify that *all* finite traces from φ can reach ψ.

To address the verification of nondeterministic programs, *all-path reachability* was proposed [9]. An all-path reachability $\varphi \Rightarrow^{\forall}_{reach} \psi$ claim holds iff all finite (and maximal) execution traces can reach ψ. Therefore, it supports the verification of nondeterministic programs. On deterministic programs, all-path and one-path reachability coincide.

To support all-path reachability, we need to extend our method with the following (Step) axiom, which introduces all-path claims from (one-path) rewrite rules in the semantics $A = \{lhs_1 \Rightarrow rhs_1, \ldots, lhs_K \Rightarrow rhs_K\}$:

$$\text{(Step)} \qquad A \vdash_\emptyset^{reach} \varphi \Rightarrow^{\forall}_{reach} (\psi_1 \vee \cdots \vee \psi_K)$$

where for $1 \leq k \leq K$, ψ_k is the result of executing φ for one step, using the k-th semantic rule $lhs_k \Rightarrow rhs_k$. Intuitively, the (Step) axiom states that an execution step must be made using one of the semantic rules in A.

4 Timeline

The proposed project is expected to be accomplished in one year with 2 graduate students, 2 undergraduate students, and 1 postdoc or advanced graduate student.

The main deliverable of the project is a proof generation algorithm that can generate proof certificates for $\mathbb{K}$'s formal verification tool. The algorithm should support all the main programming languages whose semantics have been defined in $\mathbb{K}$. We will work closely with Runtime Verification, Inc., which will facilitate and accelerate the publication of the integration proposed in this project.

	2021			2022		Publications
	Q2	Q3	Q4	Q1	Q2	
Implement the proof generation algorithm for $\mathbb{K}$'s symbolic execution tool						#1
Design and implement the proof generation algorithm for $\mathbb{K}$'s program verification tool						
Implement the proof checker on a portable TEE device (such as a USB token)						#2
Case study: the ERC20 standard						#3

5 Expected Outcomes and Impact

The proposed research will make blockchain applications and smart contracts more trustworthy, by generating for them machine-checkable proof certificates. For each smart contract that is verified using $\mathbb{K}$, we generate a complete, rigorous, and machine-checkable mathematical proof that independently certifies the verification result. By generating these machine-checkable proof certificates, we establish the correctness of blockchain applications and smart contracts ***without needing to trust*** $\mathbb{K}$, thus eliminating $\mathbb{K}$ from the trust base.

The proposed research will make the existing smart contract verification obtained using $\mathbb{K}$ more transparent and accessible to all the blockchain stakeholders, and not only to expert semanticists and verification engineers, because what is now hidden behind $\mathbb{K}$'s internal verification algorithms, transformations, heuristics, and optimization will be made explicit by concrete and machine-checkable proof steps. Smart contract verification results are therefore reproducible and independently checkable, making the $\mathbb{K}$-verified smart contracts more trustworthy.

References

[1] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts SoK. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, pages 164–186, Berlin, Heidelberg, 2017. Springer.

[2] Denis Bogdănaş and Grigore Roşu. K-Java: A complete semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456, Mumbai, India, 2015. ACM.

[3] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. An in-depth look at the parity multisig bug. *Hacking Distributed*, 1(1):1–1, 2017. `http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/`.

[4] Vitalik Buterin. Thinking about smart contract security. *Ethereum Blog*, 1(1):1–1, 2016. `https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/`.

[5] Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh, and Grigore Roşu. Towards a trustworthy semantics-based language framework via proof generation. In *Proceedings of the 33rd International Conference on Computer-Aided Verification*, Virtual, July 2021. ACM.

[6] Xiaohong Chen, Dorel Lucanu, and Grigore Roşu. Matching logic explained. *Journal of Logical and Algebraic Methods in Programming*, 120:1–36, 2021.

[7] Xiaohong Chen and Grigore Roşu. Matching μ-logic. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS'19)*, pages 1–13, Vancouver, Canada, 2019. IEEE.

[8] Xiaohong Chen and Grigore Roşu. A general approach to define binders using matching logic. In *Proceedings of the 25th ACM SIGPLAN International Conference on Functional Programming (ICFP'20)*, pages 1–32, New Jersey, USA, 2020. ACM.

[9] Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. All-path reachability logic. In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14)*, volume 8560, pages 425–440, Vienna, Austria, 2014. Springer.

[10] 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, Amsterdam, Netherlands, 2016. ACM.

[11] Phil Daian. DAO attack, 2016. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[12] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*, pages 1133–1148, Phoenix, Arizona, USA, 2019. ACM.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, volume 4963, pages 337–340, Budapest, Hungary, 2008. Springer.

[14] Francisco Durán and Hubert Garavel. The rewrite engines competitions: a RECtrospective. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–100, Cham, 2019. Springer International Publishing.

[15] Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. *ACM SIGPLAN Notices*, 47(1):533–544, 2012.

[16] Dwight Guth. A formal semantics of Python 3.3. Technical report, University of Illinois at Urbana-Champaign, 2013.

[17] Dwight Guth, Chris Hathhorn, Manasvi Saxena, and Grigore Roşu. RV-Match: Practical semantics-based program analysis. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV'16)*, volume 9779, pages 447–453, Toronto, Ontario, Canada, 2016. Springer.

[18] 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 2018 IEEE Computer Security Foundations Symposium (CSF'18)*, pages 204–217, Oxford, UK, 2018. IEEE. http://jellopaper.org.

[19] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[20] K Team. Matching logic proof checker. GitHub page https://github.com/kframework/matching-logic-proof-checker, 2021.

[21] Qingzhou Luo, Yi Zhang, Choonghwan Lee, Dongyun Jin, Patrick O'Neil Meredith, Traian Florin Şerbănuţă, and Grigore Roşu. RV-Monitor: Efficient parametric runtime verification with simultaneous properties. In *Proceedings of the 5th International Conference on Runtime Verification (RV'14)*, pages 285–300, Toronto, Canada, 2014. Springer International Publishing.

[22] Nicholas D. Matsakis and Felix S. Klock. The Rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104, , 2014. ACM.

[23] Norman Megill and David A. Wheeler. Metamath: a computer language for mathematical proofs. Available at http://us.metamath.org, 2019.

[24] Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356, Portland, OR, 2015. ACM.

[25] Python Software Foundation. CPython. Available on GitHub https://github.com/python/cpython, 2020.

[26] Grigore Roşu. Matching logic. *Logical Methods in Computer Science*, 13(4):1–61, 2017.

[27] Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. Reachability logic. Technical Report http://hdl.handle.net/2142/32952, University of Illinois, July 2012.

[28] Grigore Rosu. K—A semantic framework for programming languages and formal analysis tools. In *Dependable Software Systems Engineering*, pages 186–206, Marktoberdorf, Germany, 2017. IOS Press.

[29] Runtime Verification Inc. Publications. Available at `https://github.com/runtimeverification/publications`, 2022.

[30] Jutta Steiner. Security is a process: A postmortem on the parity multi-sig library self-destruct, 2017. `http://goo.gl/LBh1vR`.

[31] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum & Parity*, 1(1):1–39, 2014. (Updated for EIP-150 in 2017) `http://yellowpaper.io/`.