IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received November 28, 2021, accepted January 4, 2022, date of publication January 14, 2022, date of current version January 21, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3143145

Formal Verification of Blockchain Smart Contracts via ATL Model Checking

WONHONG NAM[1] **AND HYUNYOUNG KIL**[2]
[1]Department of Computer Science and Engineering, Konkuk University, Seoul 05029, South Korea
[2]Department of Software, Korea Aerospace University, Goyang 10540, South Korea

Corresponding author: Hyunyoung Kil (hykil@kau.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) Grant through the Korea Government (MSIT) under Grant 2019R1F1A106282813 and Grant 2021R1F1A105038911.

ABSTRACT A blockchain is a list of data blocks as a publicly distributed ledger, which are linked together using cryptography. By allowing Turing-complete programming languages to implement smart contracts, recent blockchains such as Ethereum can reduce needs in trusted intermediators, arbitrations and enforcement costs. However, subtle errors in smart contracts have induced an enormous financial loss—for examples, the DAO attack, Parity multisignature wallet attacks, and integer underflow/overflow attacks. To identify such errors in smart contracts, various researches are performed, which are based on static analysis and theorem proving. However, they only support inspection for pre-defined error patterns, or they cannot explore the whole searching space exhaustively or be fully automatic. Hence, in this paper, we propose a novel formal verification technique to analyze blockchain smart contracts by using ATL model checking. In our methodology, we represent the interaction between users and smart contracts into a two-player game and verify properties we want to check using MCMAS that is an efficient ATL model checker for multi-agent systems. Moreover, we present three case studies to show that our proposal can successfully identify subtle flaws in real world smart contracts.

INDEX TERMS Blockchain, ethereum, smart contracts, formal verification, alternating-time temporal logic (ATL), ATL model checking.

I. INTRODUCTION

In 2008, Nakamoto proposed Bitcoin [1] as the first blockchain system [2], [3], in which users can send a decentralized cryptocurrency to other users on the peer-to-peer network without a central authority. A blockchain is a growing list of cryptographically secured blocks to maintain shared data on decentralized systems, in order to archive transactions between untrusted participants. Each block includes a cryptographic hash of the previous block, a timestamp and transactions. Since every block contains the hash of the previous block, we cannot modify any data in a block without altering all the subsequent blocks in the blockchain. To define transactions between users, Bitcoin supports a Forth-like scripting language [4].

Smart contracts are computer programs that automatically execute legal events according to the terms of contracts. Although the scripting language of Bitcoin can represent a weak version of smart contracts such as an escrowed payment transaction, it is somewhat restricted due to its small instruction set. As a result, Buterin proposed Ethereum [5], [6] in 2013, which supports a Turing-complete language without any expressiveness restriction to build applications (i.e., smart contracts) that users can interact with. Now, we have Turing-complete languages Solidity [7] and Vyper [8] to build smart contracts in the Ethereum blockchain.

As employing blockchain systems to be used in various applications, Ethereum is the most actively used blockchain at this point. Decentralized finance applications written as Ethereum smart contracts can support a number of financial services without the help of brokerages, exchanges, or banks. Also, Ethereum makes it possible to create and exchange Non-fungible tokens, and to operate many other cryptocurrencies as ERC-20 (Ethereum Request for Comment 20) tokens on top of itself. Although the Ethereum blockchain has

The associate editor coordinating the review of this manuscript and approving it for publication was Francisco J. Garcia-Penalvo.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

been successfully applied to a number of interesting applications, there have been several events that subtle flaws in smart contracts induce a huge amount of financial loss; e.g., the DAO attack [9], Parity multisignature wallet attacks [10], and integer underflow/overflow attacks [11]. Especially, in 2016, a hacker exploited an error in a smart contract of the DAO (Digital decentralized Autonomous Organization), and stole $50 million of Ether. We call this event "*the DAO attack*" that caused Ethereum to be split into *Ethereum Classic* and *Ethereum*.

To analyze and verify such defects in smart contracts, there are several research approaches [12] that adopt static program analysis, theorem proving, and model checking. Static program analysis [13] is to analyze computer programs, which is performed without execution of the programs. Theorem proving technique [14] is to prove mathematical theorems of computer programs through axioms and proof rules. Model checking technique [15] is, given a target system and a property, to automatically verify whether the system satisfies the given property. These techniques, however, inherently have some problems. The static analysis techniques should precisely abstract smart contract execution and memory model, and they support only checking for pre-defined error patterns. For theorem proving, it is a significant problem that the technique cannot be performed fully automatically. While model checking techniques can be done automatically, general model checking techniques such as CTL (Computational Tree Logic) model checking [16] and LTL (Linear-time Temporal Logic) model checking [17] cannot exactly capture a game property between users and smart contracts. Accordingly, in this paper, we propose to employ ATL (Alternating-time Temporal Logic) model checking [18] that can model and verify game properties between multi-agent systems. In our method, given target smart contracts written in Solidity, we first translate them into a game structure that is supported by ATL model checking. To do this, we contrive translation rules to completely preserve the semantics of Solidity language. After conversion by our rules, we verify the target smart contracts by using an efficient formal verification tool, MCMAS [19] that is a model checker for multi-agent systems. In this process, we regard the interaction between users and smart contracts as a two-player game, and we distinguish the whole system including users and smart contracts into a *protagonist* and an *antagonist*. In addition, we represent properties between two-players in an ATL formula, and verify whether the whole system satisfies the properties by MCMAS. Finally, we present three case studies to show that our technique can be successfully applied to verify real world smart contracts. In our case studies, we can successfully identify the DAO bug and the `tx.origin` bug, and we also prove that corrected versions satisfy safety properties that there is no such an error.

The rest of this paper is organized as follows. We discuss related work for our research in Section 2. In Section 3, we account for formal verification technique including ATL model checking as background. Then, in Section 4, we propose our novel method to verify blockchain smart contracts written in Solidity. In Section 5, we present three case studies to show that our technique can be used to identify subtle errors in smart contract programs. Finally, we give conclusions in Section 6.

II. RELATED WORK

As blockchain platforms have recently emerged as a new initiative for a secure distributed data storage, many literatures introduce novel research issues on the blockchain system [3]. Since Nakamoto proposed Bitcoin [1] in 2008, a number of blockchain services including Ethereum [5], [6], Facebook's Libra (now, called Diem) [20], and Bitwala [21] have been proposed to be applied to various usages. That is, although the early blockchains were developed as cryptocurrencies, recent blockchain systems have been used in a number of financial services including online payment, digital assets and remittance [22]. Bitwala [21] initiated a blockchain banking solution that allows clients to manage their cryptocurrencies and Euro in bank accounts. Digital assets such as ICO (Initial Coin Offerings) and STOs (Security Token Offerings) are used to tokenize traditional assets such as company, intellectual property, real estate, art, or individual products [23]. Mojaloop was designed to provide a reference model for payment interoperability, which can allow migrants to send remittances [24]. Andoni *et al.* [25] provided a systematic review of challenges and opportunities to employ blockchain technology for the energy sector. In addition, there are several results to apply blockchains to supply chain managements [26], [27]. Moreover, one of the most significant application of blockchain platforms is to expedite smart contracts that are computer programs to automatically be executed without any trusted third party when conditions in the contract hold [28], [29].

Even though smart contracts based on blockchain technology have been used for various applications, subtle defects in smart contracts may incur an enormous financial loss. Recently, various efforts have been made to find or prevent errors in smart contracts. A number of researches [30]–[32] employ static analysis that can analyze computer programs without actually executing programs. In [30], Feist *et al.* propose Slither that is a static analysis framework to analyze rich information about Ethereum smart contracts. Their method translates Solidity smart contracts into an intermediate representation, and then exploits dataflow and taint tracking to automatically detect vulnerabilities of smart contracts. So *et al.* [31] present a static analysis tool VeriSmart to ensure arithmetic safety of Ethereum smart contracts. They can reduce the burden of manually checking for undiscovered or incorrectly-reported issues by using a domain-specific algorithm. In [32], Tsankov *et al.* propose a security analyzer for Ethereum smart contracts, which automatically prove contract behaviors as safe/unsafe with respect to a given property. Their tool symbolically analyzes the contract's dependency graph, and then checks compliance

and violation patterns to capture sufficient conditions for proving whether a property is true. These techniques based on static analysis have an issue that they require precise abstraction of the behavior of smart contracts and memory model.

Theorem proving is a formal verification technique that encodes a target system and its properties into a mathematical logic and derives a formal proof of satisfaction of the properties by axioms and proof rules. The theorem proving technique has been employed to verify the correctness of blockchain smart contracts [33]–[36]. In [34], Amani *et al.* extend an existing EVM (Ethereum Virtual Machine) formalization in Isabelle/HOL by a sound program logic. They encode bytecode sequences into blocks of straight-line code and propose a program logic to analyze them. O'Connor [33] has proposed a new language called Simplicity to maintain and enhance the desirable properties of smart contracts. The language provides formal semantics and facilitates reasoning on smart contracts by using a theorem prover. In [35], Annenkov *et al.* propose a method to embed a functional language into the Coq proof system by using metaprogramming. This allows us to perform reasoning on smart contracts by using the Coq system. However, these techniques employing theorem proving inherently have a problem that they cannot be performed fully automatically and thus need expert's guide.

Another promising approaches [37]–[39] to identify bugs in smart contracts are to employ model checking that is an automatic formal verification technique. Given a target system and a property, it checks whether the system satisfies the given property. If not, model checking tools provide a counter-example that can explain why the system does not satisfy the property. In [37], Nehai *et al.* have modeled into three layers to represent the behavior of Ethereum blockchain, smart contracts and the execution framework. Then, they verify temporal logic properties in CTL formulae of smart contracts by using a model checker NuSMV [40]. On the other hand, Mavridou *et al.* [38] have built the VeriSolid framework for model checking of smart contracts. It allows developers to model and verify smart contracts as well as to generate Solidity codes as an end-to-end design framework. In [39], Osterland and Rose propose a tool chain for model checking of Ethereum smart contracts. First, they translate a Solidity smart contract into a Promela model that is the input of the SPIN model checker [41], and then they employ SPIN to check if the model satisfy given LTL properties. Although these approaches adopt CTL or LTL model checking to analyze smart contracts, these CTL/LTL model checking cannot handle a game property that represents whether a particular player has a winning strategy against the other player. Since many interesting properties about behavior of smart contracts should be modeled into winning strategies of the game between a protagonist and an antagonist [42], it is necessary to verify existence of such strategies. Hence, in this paper, to analyze blockchain smart contracts, we employ ATL model checking that can verify whether a player represented as a protagonist has a winning strategy no matter how other players perform.

III. ATL MODEL CHECKING

In this section, we account for ATL model checking that we employ to analyze properties of smart contracts.

A. ATL MODEL CHECKING PROBLEM

Alternating-time Temporal Logic (ATL) model checking [18] is a formal verification method to analyze if a given system satisfies a given property. In this technique, we represent the system we want to verify with a game structure, in which multiple players are described in separate agents. Besides, we represent the desired properties we want to verify in ATL formulae. Then, an ATL model checker exhaustively searches state space to investigate whether the system satisfies the given ATL properties.

In this section, we present a formal notion of Alternating-time Temporal Logic and ATL model checking. Since the definition of ATL model checking requires the definition of ATL, we first explain ATL and then define ATL model checking.

The formal syntax of ATL is as follows. Π is an underlying set of *atomic propositions* and Σ is a finite set of *players*.

1) Every atomic proposition $p \in \Pi$ is an ATL formula.
2) If φ_1 and φ_2 are ATL formulae, then $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$ and $\varphi_1 \wedge \varphi_2$ are also ATL formulae.
3) If $A \subseteq \Sigma$ is a set of players, and φ_1 and φ_2 are ATL formulae, then $\langle\langle A\rangle\rangle X\varphi_1$, $\langle\langle A\rangle\rangle G\varphi_1$, $\langle\langle A\rangle\rangle F\varphi_1$, and $\langle\langle A\rangle\rangle \varphi_1 U \varphi_2$ are ATL formulae.

The symbols $\neg$, $\vee$ and $\wedge$ are a *negation*, a *disjunction* and a *conjunction*, respectively. $\langle\langle\ \rangle\rangle$ is a *path quantifier*, and X, G, F and U are *temporal operators*. As other temporal logics such as LTL and CTL, X, G, F, and U are the *nexttime* operator, the *global* operator, the *future* operator, and the *until* operator, respectively.

The semantics of ATL formulae is defined with respect to a game structure. Formally, a game structure is a tuple $S = (k, Q, \Pi, \pi, d, \delta)$ in which

- k is the number of *players* in the game. We identify the players with the numbers, $1, \ldots, k$ and write $\Sigma = \{1, \ldots, k\}$ for the set of players.
- Q is a finite set of *states*. Also, we have a set $Q_0 \subseteq Q$ of *initial* states.
- Π is a finite set of *atomic propositions*.
- $\pi : Q \to 2^{\Pi}$ is a *labeling function* which assigns a state $q \in Q$ to a set $\pi(q) \subseteq \Pi$ of atomic propositions true at the state q.
- For each player $a \in \{1, \ldots, k\}$ and each state $q \in Q$, a natural number $d_a(q) \geq 1$ is the number of *moves* available at the state q to the player a. Now, we identify moves of the player a at the state q with the numbers, $1, \ldots, d_a(q)$. For each state $q \in Q$, a *move vector* μ at q is a k-tuple $(m_1, \ldots, m_k)$ such that $1 \leq m_a \leq d_a(q)$

$q \models p$	iff	$p \in \pi(q)$ for propositions $p \in \Pi$
$q \models \neg\varphi$	iff	$q \not\models \varphi$
$q \models \varphi_1 \vee \varphi_2$	iff	$q \models \varphi_1$ or $q \models \varphi_2$
$q \models \varphi_1 \wedge \varphi_2$	iff	$q \models \varphi_1$ and $q \models \varphi_2$
$q \models \langle\langle A\rangle\rangle X\varphi$	iff	there exists a set F_A of strategies, one for each player in A, such that for all computation $\lambda = q_0, q_1, \cdots \in out(q, F_A)$, $q_1 \models \varphi$.
$q \models \langle\langle A\rangle\rangle G\varphi$	iff	there exists a set F_A of strategies, one for each player in A, such that for all computation $\lambda = q_0, q_1, \cdots \in out(q, F_A)$, $q_i \models \varphi$ for all $i \geq 0$.
$q \models \langle\langle A\rangle\rangle F\varphi$	iff	there exists a set F_A of strategies, one for each player in A, such that for all computation $\lambda = q_0, q_1, \cdots \in out(q, F_A)$, there exists some $i \geq 0$ such that $q_i \models \varphi$.
$q \models \langle\langle A\rangle\rangle \varphi_1 U \varphi_2$	iff	there exists a set F_A of strategies, one for each player in A, such that for all computation $\lambda = q_0, q_1, \cdots \in out(q, F_A)$, there exists some $i \geq 0$ such that $q_i \models \varphi_2$ and for all $0 \leq j < i$, we have $q_j \models \varphi_1$.

FIGURE 1. The definition of the satisfaction relation.

for each player a, where k is the number of players. In addition, for each state $q \in Q$, we denote $D(q)$ for the set $\{1, \ldots, d_1(q)\} \times \cdots \times \{1, \ldots, d_k(q)\}$ of all move vectors at the state q. We call the function D a *move function*.

- For each state $q \in Q$ and each move vector $(m_1, \ldots, m_k) \in D(q)$, a state $\delta(q, m_1, \ldots, m_k) \in Q$ is the next state when each player $a \in \{1, \ldots, k\}$ chooses a move m_a at the state q. We call the function δ a *transition function*.

Now, we can define the ATL model checking problem as follows. Given a game structure $S = (k, Q, \Pi, \pi, d, \delta)$, a state $q \in Q$, and an ATL formula φ, we denote that $S, q \models \varphi$ if the state q of S satisfies the ATL property φ. When S is clear, we can omit S as $q \models \varphi$. We define the satisfaction relation $\models$ in Figure 1. Given a game structure $S = (k, Q, \Pi, \pi, d, \delta)$ and an ATL formula φ, we say $S \models \varphi$ if $q \models \varphi$ for all the initial states q of S. Finally, given a game structure $S = (k, Q, \Pi, \pi, d, \delta)$, and an ATL formula φ, the *ATL model checking problem* is to check whether $S \models \varphi$. To understand ATL model checking in more detail, refer to [18].

B. ATL MODEL CHECKING EXAMPLE

Let us consider an example of the ATL model checking problem which includes two processes p_x and p_y. The process p_x assigns a value to the Boolean variable x and the process p_y assigns a value to the Boolean variable y. When $x = false$, the process p_x may leave the value of x unchanged or set the value to *true*. When $x = true$, p_x leaves the value of x as *true*. Likewise, when $y = false$, the process p_y can leave the value of y unchanged or set the value to *true*. When $y = true$, p_y leaves the value of y as *true*. Initially, x and y are both *false*. Now, we can represent the synchronous composition of the two processes with the following game structure $S = (k, Q, \Pi, \pi, d, \delta)$ where

- $k = 2$. Player 1 is the process p_x and player 2 is p_y.
- $Q = \{q_0, q_1, q_2, q_3\}$. The set Q_0 of initial states is $\{q_0\}$.

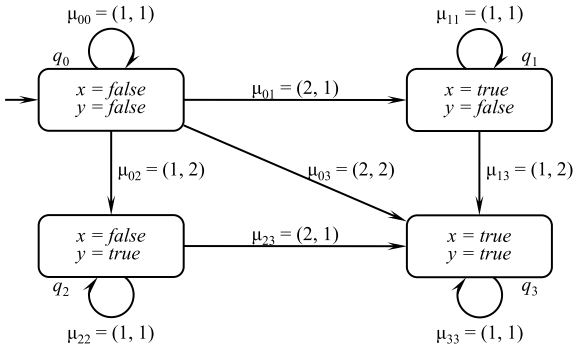


FIGURE 2. The state transition system.

- $\Pi = \{x, y\}$ where both variables are atomic propositions.
- $\pi(q_0) = \emptyset, \pi(q_1) = \{x\}, \pi(q_2) = \{y\}$, and $\pi(q_3) = \{x, y\}$.
- $d_1(q_0) = d_1(q_2) = 2$ and $d_1(q_1) = d_1(q_3) = 1$. That is, in the states q_0 and q_2, the player 1 (process q_x) has two moves by which player 1 can leave the value of x unchanged or set the value to *true*. On the other hand, in q_1 and q_3, the player 1 has only one move that q_x leaves the value of x unchanged as *true*. Likewise, $d_2(q_0) = d_2(q_1) = 2$ and $d_2(q_2) = d_2(q_3) = 1$. In the states q_0 and q_1, the player 2 (process q_y) has two moves by which player 2 can leave the value of y unchanged or set the value to *true*. On the other hand, in q_2 and q_3, the player 2 has only one move that q_y leaves the value of y unchanged as *true*.
- The transition function δ is illustrated in Figure 2.

In this example, we can consider an ATL formula $\langle\langle p_x\rangle\rangle F(x = true)$ which means that p_x can enforce the variable x eventually *true* no matter how the other player chooses its move. This formulae holds at the initial state q_0 since p_x just needs to choose its move as 2; therefore, $S \models \langle\langle p_x\rangle\rangle F(x = true)$. On the other hand, consider an

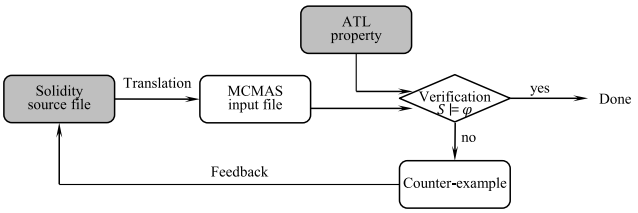


FIGURE 3. ATL model checking procedure for solidity contracts.

ATL formulae $\langle\langle p_y\rangle\rangle X(x = y)$ which means that p_y can enforce that in the next state the variable x has the same value with y no matter how p_x chooses its move. This formulae does not hold at the initial state q_0 since if p_y chooses move 1, then p_x can choose its move as 2, and if p_y chooses move 2, then p_x can choose its move as 1; therefore, $S \not\models \langle\langle p_y\rangle\rangle X(x = y)$.

IV. TRANSLATION FOR ATL MODEL CHECKING

Our verification procedure for smart contracts is illustrated in Figure 3. Since we target Ethereum smart contracts written in Solidity, one of our inputs is a Solidity source file. For formal verification, we employ a state-of-the-art ATL model checker, MCMAS [19]. To apply MCMAS to the given Solidity contract file, we first translate it into a semantically equivalent MCMAS input file. In addition, MCMAS requires another input that describes ATL properties we want to check. Once two inputs are provided, we can perform ATL model checking to check whether a game structure representing the given smart contracts satisfies the ATL properties. If the result is yes, it means that the smart contracts satisfy the properties. Otherwise, MCMAS provides a counter-example that is a witness to explain why the model does not satisfy a property. This counter-example is helpful to revise the original smart contract to fix a bug. In this section, we account for our translation rule in detail.

A. THE SOLIDITY LANGUAGE

Solidity [7] is an object-oriented, high-level language to implement smart contracts in Ethereum. Each smart contract is a program that can govern the behavior of accounts within the Ethereum state. Figure 4 shows a simple example of a smart contract in Solidity. The example is a snippet of the `userWallet` smart contract that contains two variable declarations for `owner` and `myBalance` and three functions (`constructor`, *fallback* function without its name, and `withdrawTotalBalance`). We explain our translation rule by three categories: program structure, control flow, and memory operation.

B. PROGRAM STRUCTURE

A Solidity program includes contracts that contain variable declarations, mapping declarations, event definitions and functions. On the other hand, a game structure in MCMAS is composed of a set of *agents*, each of which represents a player in a game. Accordingly, at the top level, a contract in Solidity is translated into an agent in the MCMAS language.

Then, each part of a Solidity contract (i.e., variable declarations, mapping declarations, event definitions and functions) is translated as follows. The Solidity variable types supported in this paper are *Booleans, integers, addresses, arrays, enums*, and *structures*, which are not all the types of Solidity variables, but enough to analyze real world Solidity contracts. Since verification performance of MCMAS inheritantly depends on state space, integers and addresses are reduced into bounded integers in the MCMAS language after appropriate abstraction. Because arrays and structures are not supported in MCMAS, we flatten them into a set of bounded integers. In Solidity, variables are classified as *state variables, local variables*, and *global variables*. Remark that in a Solidity program, a state variable is a contract-level variable that can be accessed in all functions of the contract, and a global variable is a variable with a reserved name in the global namespace used to get information about the blockchain and the corresponding transaction. Since, in MCMAS, there is no distinction among them, we attach a prefix to a local variable with the function name in order to prevent from overwriting state/local variables with the same name. In addition, we create a set of MCMAS variables corresponding to global variables that appear in a Solidity source file. For instance, `tx.origin` in Figure 4 is a global variable that refers to the address which originally instantiates the transaction. In this case, we create a corresponding MCMAS variable `tx_orgin`.

Like arrays, we unwind a mapping to a set of MCMAS variables. As Solidity events only provide the EVM's logging functionality, we ignore them in this paper.

Another main difference between Solidity and the MCMAS language is the execution mechanism. While a program in Solidity runs sequentially, line by line as ordinary programming languages, the MCMAS language represents an evolution function of a given game structure which describes how local states of agents evolve based on their current local states and on other agents' actions. The evolution function includes a set of lines that consist of a set of assignments of variables and a guided condition. The assignments are enabled in a state if their guided condition is satisfied in that state. To simulate the sequential execution of a Solidity contract with the MCMAS evolution function description, we introduce an auxiliary variable, *program counter* (`pc`) to record which line of a Solidity program is being executed currently. The constructor in Solidity is a special function that is executed automatically when the corresponding contract is instantiated. The assignments in the constructor, therefore, are encoded into a formula in *initial states* section of the MCMAS language. Moreover, since the MCMAS language does not support a function call, we also simulate it with auxiliary variables to represent a *call stack*.

C. CONTROL FLOW

Since the execution of a Solidity program is sequential, we can translate each statement of a Solidity program by increasing the program counter by 1. However, there are

```
 1: contract userWallet {
 2:     address public owner;
 3:     uint myBalance;
 4:
 5:     constructor (address addr) {
 6:        owner = addr;
 7:        myBalance = 100;
 8:     }
 9:
10:     function () public payable {
11:     }
12:
13:     function withdrawTotalBalance(address recipient) public {
14:        if(tx.origin == owner) {
15:           recipient.transfer(myBalance);
16:        }
17:     }
18: }
```

FIGURE 4. Example: A snippet of the userWallet smart contract in solidity.

several exceptional cases of control flow: *function call, branch*, *loop, require* and *assert*. For a function call in a Solidity program, we can translate it by storing the current value of `pc` to call stack variables and assigning the start line number of the callee to `pc`. In addition, we handle parameter passing by assigning the values of actual parameters to formal parameters. To do this, for each formal variable, we introduce a new local variable in the callee function. When a function return, we assign a return value to a variable in the caller and restore `pc` with the return address stored. The *if* statement is reduced into an implication; that is, if the guided condition is evaluated to *true*, we assign the start line number of the *then* block to `pc`. Otherwise, we assign the start line number of the *else* block to `pc`. Loop statements (e.g., *for, while,* and *do..while*) can be translated with the similar way; we add appropriate initialization, termination condition, and increment into the target MCMAS file, and assign the start line number of the loop to `pc` at the end of the loop. In Solidity, when a given condition does not hold, *require* and *assert* terminate the execution. Hence, we translate them by jumping `pc` to the last line number of the function, provided that the condition is false.

D. MEMORY OPERATIONS

Memory operations include assignments and arithmetic operations. Most of them have the same semantics between two languages except some syntactic sugars (e.g., `++`, `--`, `+=`, `-=` and so on) that are not supported in the MCMAS language. In these cases, we first reduce them into equivalent primitive operators, and then translate them into the MCMAS language. For instance, `x += y` is reduced into `x = x + y`, and then it is translated to the MCMAS language.

E. EXAMPLE

Figure 4 and Figure 5 present a translation example for the userWallet smart contract; Figure 4 is the Solidity source file, and Figure 5 is the MCMAS file translated. Remark that a line beginning with – in the MCMAS file is a comment. As our translation method explained above, the contract `userWallet` is translated into an agent with the same name in the MCMAS file. To simulate sequential execution and function calls, we introduce two new variables, `pc` and `returnAddr`. Their maximum bounds are induced from the number of lines of the Solidity source file, and their minimum value (that is 0) represents the initial state which means that the corresponding contract is not activated yet. Then, a state variable, local variables, and global variables appear, which correspond to variables in the Solidity source file. The *Actions* section and the *Protocol* section are to model function calls and returns in the MCMAS language. That is, the *Action* section declares which functions this contract can invoke and which functions of the contract can return. The *Protocol* section specifies at which line the corresponding actions occur.

Each function in Solidity (except the `constructor`) is translated into lines of the *Evolution* section in MCMAS. Note that statements in the constructor are encoded into the *InitStates* section. The lines corresponding to the *fallback* function just increase `pc` because the *fallback* function in this example does not include any statement. The *if* statement in the Solidity file is translated into two lines in the MCMAS file; i.e., when the guided condition (`tx_origin=owner`) is evaluated to true, `pc` is incremented by 1. Otherwise, `pc` is jumped to Line 17. The function call at Line 15 in the Solidity source file is translated into lines where we store the value of `pc` to `returnAddr` and restore `pc` with `returnAddr + 1` once the action representing that the `transfer` function returns occurs. In addition, we suppose that there exists an auxiliary agent `Environment` that manages the turn for agents. Accordingly, each line of Evolution section of `userWallet` may be enable only if `Environment.turn` is `T_userWallet`.

Our proposed approach supports a subset of Solidity syntax. That is, the current version cannot directly deal with various syntactic sugars in Solidity language, function type variables, dynamic creation of contracts, and inheritance

```
Agent userWallet
   Vars:
      pc: 0..18;
      returnAddr: 0..18;
-- state variables
      owner: 0..255;
      myBalance: 0..255;
-- local variables for constructor()
      constructor_addr: 0..255;
-- local variables for withdrawTotalBalance()
      withdrawTotalBalance_recipient: 0..255;
-- global variables
      tx_origin: 0..255;
   end Vars
   Actions = {fallback_return, withdrawTotalBalance_return, call_transfer, none};
   Protocol:
      pc = 11: {fallback_return};
      pc = 15: {call_transfer};
      pc = 17: {withdrawTotalBalance_return};
      Other : {none};
   end Protocol
   Evolution:
-- function () public payable {}
      pc = 10 if (caller.Action=fallback);
      pc = pc + 1 if (pc=10) and (Environment.turn=T_userWallet);
-- function withdrawTotalBalance(address recipient) public {}
      pc = 13 if (caller.Action=call_withdrawTotalBalance);
      pc = pc + 1 if (pc=13) and (Environment.turn=T_userWallet);
      pc = pc + 1 if (pc=14) and (tx_origin=owner) and (Environment.turn=T_userWallet);
      pc = 17 if (pc=14) and !(tx_origin=owner) and (Environment.turn=T_userWallet);
      returnAddr = pc if (pc=15) and (Environment.turn=T_userWallet);
      pc = returnAddr + 1 if (caller.Action=transfer_return);
      pc = pc + 1 if (pc=16) and (Environment.turn=T_userWallet);
   end Evolution
end Agent

InitStates
   (userWallet.pc=0) and (userWallet.returnAddr=0) and
   (userWallet.owner=userWallet.constructor_addr) and (userWallet.myBalance=100)
end InitStates
```

FIGURE 5. Translation example: MCMAS file translated from the userWallet smart contract.

including polymorphism. In the case of syntactic sugars, we can translate them into MCMAS language by reducing them into primitives and then converting the primitives to MCMAS. Because, for remaining issues, the formal verification field is actively challenging them by various approach, we can adopt these state-of-the art techniques in our future work. In spite of the limitation, our current proposal is practical enough to verify real world examples in Section V.

V. CASE STUDY

In this section, we present three case studies to show that our technique can be successfully applied to analyze real world smart contracts.

A. DAO ATTACK

The DAO [9] is a decentralized autonomous organization for investor-directed venture capital fund. It was instantiated on the Ethereum in April 2016, but users exploited a reentrancy vulnerability in the DAO code in June 2016. Finally, they stole 3.6 million Ether valued at around $50M at the time. Figure 6 is a snippet of the DAO attack; Figure 6 (a) is the DAO contract and Figure 6 (b) is the attacker contract. When an attacker calls the `startAttack` function, it deposits 10 wei to its account in the DAO fund and then attempts to withdraw its total balance (Lines 29–30). Once the attacker calls `withdrawTotalBalance`, `DAOfund` executes `msg.sender.call.value(totalBalance)` at Line 12 which refers to the *fallback* function of `DAOattack`. This function call indeed induces to transfer attacker's total balance to its account. However, the attacker maliciously invokes `withdrawTotalBalance` again at Line 35 in the fallback function. This process causes fallback function calls repeatedly, and the attacker can steal all Ether from the DAO fund.

We translate these two contracts into the MCMAS language as our method in Section IV, and add an `Environment` agent which executes the `DAOfund` agent and the `DAOattack` agent according to their turn. To examine the possibility of the above hacking, we check the following ATL formula that means whether the `DAOattack` can eventually withdraw more Ether than it possesses no matter what the other agents (i.e, `DAOfund` and `Environment`) do.

$$\langle\langle \texttt{DAOattack} \rangle\rangle F\ (\texttt{DAOattack.totalAmount} > \texttt{DAOattack.currentBalance})$$

```
 1: contract DAOfund {
 2:     mapping(address=>uint) balanceOf;
 3:
 4:     function depositBalance() external payable {
 5:         balanceOf[msg.sender] += msg.value;
 6:     }
 7:
 8:     function withdrawTotalBalance() external {
 9:         uint totalBalance = balanceOf[msg.sender];
10:
11:         if(totalBalance > 0) {
12:             if(msg.sender.call.value(totalBalance)() == false) {
13:                 revert();
14:             }
15:             balanceOf[msg.sender] = 0;
16:         }
17:     }
18: }
```

(a) The Solidity DAO contract

```
19: contract DAOattack {
20:     address dao;
21:     uint currentBalance;
22:
23:     constructor(address fundAddress) public {
24:         dao = fundAddress;
25:     }
26:
27:     function startAttack () {
28:         currentBalance = 10;
29:         dao.depositBalance(currentBalance);
30:         dao.withdrawTotalBalance();
31:     }
32:
33:     function () public payable {
34:         totalAmount += msg.value;
35:         dao.withdrawTotalBalance();
36:     }
37: }
```

(b) The attacker contract

FIGURE 6. The DAO attack source codes in Solidity.

The verification result is true, and MCMAS produces a witness that is the same with the above scenario. Because the witness is complicate, we illustrate it concisely in Figure 7. The witness presents a sequence of interaction between the DAOfund contract and the DAOattack contract, which begins with `startAttack` and ends in a state where `DAOattack.totalAmount` is greater than `DAOattack.currentBalance`.

This reentrancy bug can be avoided with several ways, and one of them is to reset the balance of the message sender to 0 before calling the fallback function. We switch the assignment (Line 15) with the `if` statement block (Lines 12–14), and then check the same ATL property with MCMAS. Finally, the result is false; that is, our verification result demonstrates that the new version can successfully prevent the above attack.

In the first experiment for the DAO attack, we have adopted a simplified version of attacker's contract that is already known. However, suppose that we are developing a new contract now. At this point, we do not have any concrete attacker contract. Therefore, we cannot proceed verification with the above manner. Therefore, in the second experiment, we introduce *the most permissive model of an attacker* where the attack can invoke any function whenever it has a turn. Moreover, when the DAO fund contract calls the fallback function of the attacker, the attacker can invoke any function of the DAO fund again without returning immediately. A state transition diagram of the most permissive model of an attacker is in Figure 8. At the initial state q_0, the attacker can call `depositBalance` or `withdrawTotalBalance`. At the state q_1, once the function the attacker invokes returns or the DAO fund calls attacker's *fallback* function, it proceeds to q_0 where the attacker can invoke `depositBalance` or `withdrawTotalBalance` again. We encode this model into the MCMAS language. Finally, our verification has proved that this most permissive model of an attacker also can withdraw more amount of Ether than it has no matter what the vulnerable version of the DAO contract does. In addition, our verification has established that the most permissive model of an attacker cannot withdraw more amount than it possesses in

FIGURE 7. A witness scenario of the DAO attack (a simplified version).

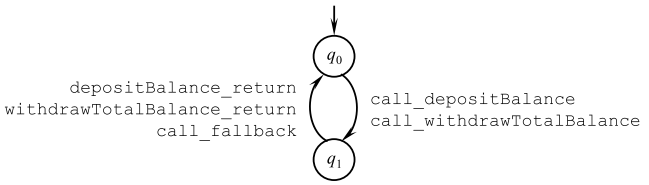


FIGURE 8. The most permissive model of an attacker for the DAO fund.

the bug-fixed version. This experiment proposes that once a programmer has developed a smart contract, she can easily verify it with the most permissive model of an attacker.

In general, a model checking problem is provided as inputs a model as well as properties. In our problem setting, if there is no source code for other necessary players that interact with our target contract, developers have to provide their models, for instance, the most permissive model of an attacker in this example. If the property or the model developers provide is incorrect, model checking results may end in a false positive/negative. In this case, however, we can easily detect and correct it by checking various properties since some of the results for various properties are not coincident with our hypothesis.

B. TX.ORIGIN ATTACK

In Solidity, global variables are variables that always exist in the global namespace and are mainly used to provide information about the blockchain or their corresponding transaction. Since some global variables have very similar meaning, developers should take note of their exact meaning. For instance, if we have a chain of function calls, while `msg.sender` points to the caller of the last function in the call chain, the value of `tx.orgin` represents who originally initiates the full call chain.

The `tx.origin` attack [43] is induced from misuse of the `tx.orgin` variable. Figure 4 in Section IV is a fragment of a contract containing the `tx.orgin` bug. Figure 9 is a snippet of an attacker contract exploiting the bug. The `userWallet` contract transfers the total amount of the balance to the caller if `tx.origin` is `owner` in Lines 13–15 of Figure 4. Now, an attacker deceives a wallet owner into invoking the fallback function of the `attacker` contract in Figure 9, which is enable by some phishing techniques. Once the owner calls the fallback function, attacker's fallback function immediately invokes the wallet's `withdrawTotalBalance` in Line 11. When `userWallet` checks whether `tx.origin` equals to `owner`, `tx.origin` is `owner` since the owner has originally initiated the call chain. Finally, the attacker can swindle out of the total balance. This attack is prevented by replacing `tx.origin` with `msg.sender`. In the above scenario, since `msg.sender` points to the attacker who is not the owner of the wallet, the `userWallet` contract refuses to transfer the fund to the attacker.

For verification of this case study, we translate two contracts into the MCMAS language, and add an environment agent to control their turn. We then check the following ATL formula meaning whether once the owner invokes the fallback function of the attacker, the attacker can withdraw the owner's fund eventually no matter what the other agents perform.

$$AG((\texttt{userWallet.Action} = \texttt{call\_fallback}) \rightarrow \langle\langle \texttt{attacker} \rangle\rangle F\ (\texttt{attacker.balance} > 0))$$

```
 1: contract attacker {
 2:     address phishableContract;
 3:     address attackerAddr;
 4:
 5:     constructor (address phishing, address attacker) {
 6:        phishableContract = phishing;
 7:        attackerAddr = attacker;
 8:     }
 9:
10:     function () payable {
11:        phishableContract.withdrawTotalBalance(attackerAddr);
12:     }
13: }
```

FIGURE 9. The attacker contract simplified.

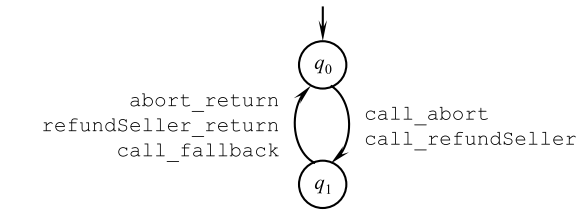
```
 1: contract Purchase {
 2:     uint public value;
 3:     address payable public seller;
 4:     address payable public buyer;
 5:     enum State {Created, Locked, Release, Inactive}
 6:     State public state;
 7:     constructor() payable {
 8:         seller = payable(msg.sender);
 9:         value = msg.value / 2;
10:         require((2 * value) == msg.value, "Value has to be even.");
11:     }
12:
13:     function abort() public {
14:         require(msg.sender == seller, "Only seller can call this.");
15:         require(state == State.Created, "Invalid state.");
16:         emit Aborted();
17:         state = State.Inactive;
18:         seller.transfer(address(this).balance);
19:     }
20:
21:     function confirmPurchase() public payable {
22:         require(state == State.Created, "Invalid state.");
23:         require(msg.value == (2 * value));
24:         emit PurchaseConfirmed();
25:         buyer = payable(msg.sender);
26:         state = State.Locked;
27:     }
28:
29:     function confirmReceived() public {
30:         require(msg.sender == buyer, "Only buyer can call this.");
31:         require(state == State.Locked, "Invalid state.");
32:         emit ItemReceived();
33:         state = State.Release;
34:         buyer.transfer(value);
35:     }
36:
37:     function refundSeller() public {
38:         require(msg.sender == seller, "Only seller can call this.");
39:         require(state == State.Release, "Invalid state.");
40:         emit SellerRefunded();
41:         state = State.Inactive;
42:         seller.transfer(3 * value);
43:     }
44: }
```

FIGURE 10. The purchase contract simplified.

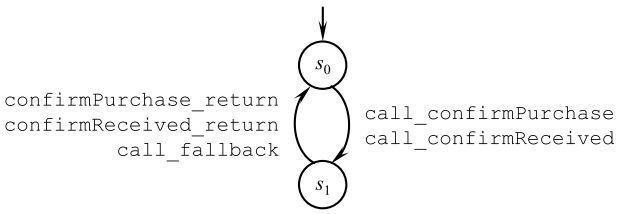
The verification result is true which means that the attacker can steal the owner's fund. On the other hand, in the version where we replace `tx.origin` with `msg.sender`, the attacker cannot dispossess since the `userWallet` contract detects that an unauthorized user demands withdrawal.

C. SAFE REMOTE PURCHASE CONTRACT

Our third case study is the *Safe Remote Purchase* contract from the Ethereum Solidity document [7]. In general, without trust between a seller and a buyer, online trading cannot take place. The `Purchase` contract in Figure 10 resolves this problem. First, a seller and a buyer deposit twice of the value of the item to the contract as escrow, and the money will be locked until the buyer confirms reception of the item. Once the buyer confirms reception, the buyer receives the value back (i.e., the half of the deposit) and the seller is returned three times of the value (i.e., the value of the item plus the deposit). The `Purchase` contract includes its constructor and four functions. When a seller wants to deal an item, the seller instantiates this contract with the deposit. After that, if a buyer wants to purchase the item, the buyer invokes the function `confirmPurchase` with the deposit. When the



(a) The model of a seller



(b) The model of a buyer

FIGURE 11. The most permissive models.

$$AG\,((\texttt{Purchase.state} = \texttt{Created} \wedge \texttt{Env.turn} = \texttt{seller}) \rightarrow \langle\langle \texttt{seller} \rangle\rangle\, F\, \texttt{sellerValueX3}) \quad (1)$$
$$AG\,((\texttt{Purchase.state} = \texttt{Locked}) \rightarrow \langle\langle \texttt{seller} \rangle\rangle\, F\, (\texttt{Purchase.state} = \texttt{Inactive})) \quad (2)$$
$$AG\,((\texttt{Purchase.state} = \texttt{Release}) \rightarrow \langle\langle \texttt{seller} \rangle\rangle\, F\, \texttt{sellerValueX3}) \quad (3)$$
$$AG\,((\texttt{Purchase.state} = \texttt{Created} \wedge \texttt{Env.turn} = \texttt{buyer}) \rightarrow \langle\langle \texttt{buyer} \rangle\rangle\, F\, \texttt{buyerValueX1}) \quad (4)$$
$$AG\,(\texttt{itemLocked} \rightarrow \langle\langle \texttt{buyer} \rangle\rangle\, G\, \neg\texttt{sellerValueX3}) \quad (5)$$

FIGURE 12. ATL properties for the purchase contract.

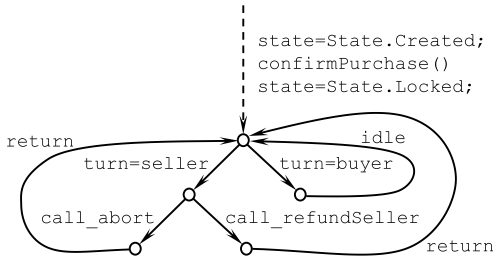


FIGURE 13. A counter-example for property (2) (a simplified version).

buyer receives the item from the seller, the buyer invokes the function `confirmReceived` that returns the half of buyer's deposit to the buyer. Once `confirmReceived` is invoked, the seller can call the function `refundSeller` that returns three times of the value to the seller. In addition, the seller may cancel this sale by calling `abort` at any time prior to buyer's invocation of `confirmPurchase`, which returns the deposit to the seller.

In this case study, since any attack is not reported yet, we do not have a concrete attacker contract. Accordingly, we introduce the most permissive models of a seller and a buyer that are illustrated in Figure 11; Figure 11 (a) is the most permissive model of a seller and Figure 11 (b) is the most permissive model of a buyer. For verification, we translate the `Purchase` contract and the most permissive models of a seller and a buyer into the MCMAS language, and again add an environment agent to manage their turn. We then inspect ATL properties in Figure 12, where Properties (1), (2), and (3) are for the seller, and Properties (4) and (5) are for the buyer. As the results, (1) and (3) are true, but (2) is false, as we have expected. A simplified counter-example for the property (2) is illustrated in Figure 13. The result for Property (1) means that once an item is instantiated and the seller has a turn, the seller can abort the corresponding sale whatever a buyer does. The counter-example for Property (2) in Figure 13 establishes that once a buyer has confirmed purchase, the seller cannot cancel or complete the sale by herself without the buyer's cooperation. Namely, after the buyer has confirmed purchase, although the seller can have a turn and invoke `abort` or `refundSeller`, the request cannot proceed since `state` is `State.Locked`. It is possible to proceed only after the buyer confirms reception. On the other hand, if the buyer has a turn, the buyer can be idle. Note that the buyer's idling does not benefit himself because the buyer cannot get his deposit back until he sends an acknowledgement for reception. The result for (3) proves that once the buyer confirms receiving of the item, the seller can get back the deposit and the item's value. For properties for the buyer, (4) and (5) are both true. The result for (4) means that if an item is instantiated and the buyer has a turn, the buyer can purchase the item and receive the half of the deposit no matter what the seller performs. Finally, the result for (5) demonstrates that once the item has been locked, the buyer can prevent the seller from withdrawing the whole deposit without shipping the item.

VI. CONCLUSION

In this paper, we have proposed a novel formal verification method that can analyze Ethereum smart contracts in the Solidity language. We first translate a given smart contract in the language of MCMAS that is a state-of-the-art ATL model checker for multi-agent systems. If an attacker contact is also given, we also reduce it into the MCMAS language. Otherwise, we introduce the most permissive model of an attacker and then encode it to MCMAS. In addition, we add an environment agent which manages their turn. Finally, we verify this multi-agent system with ATL properties a developer wants to check. If MCMAS returns true, it establishes that the properties are satisfied. Otherwise, MCMAS provides a counter-example which is significantly helpful for a developer to correct an error in the contract. Moreover, we have presented three case studies to show that our verification technique can be successfully applied to real world smart contracts.

There are several issues that are interesting for future study. First, while we have proposed our translation rule in this paper, it is limited to a subset of the Solidity full syntax. Consequently, it is worth extending our translation rule in order to automatically apply our technique to any smart contract written in Solidity. Second, we want to study another way to adopt state-of-the-art verification techniques such as an abstraction-refinement method, SAT-based algorithms for finite-state systems, and SMT-based verification techniques for infinite-state systems. Finally, we plan ample experiment to validate our technique with various Ethereum smart contracts.

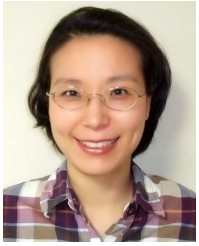
REFERENCES

[1] S. Nakamoto. (2008). *BitCoin: A Peer-to-Peer Electronic Cash System*. [Online]. Available: https://bitcoin.org/bitcoin.pdf
[2] A. M. Antonopoulos, *Mastering Bitcoin: Unlocking Digital Cryptocurrencies*, 1st ed. Newton, MA, USA: O'Reilly Media, 2015.
[3] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain challenges and opportunities: A survey," *Int. J. Grid Services*, vol. 14, no. 4, pp. 352–375, 2018.
[4] M. Swan, *Blockchain: Blueprint for a New Economy*, 1st ed. Newton, MA, USA: O'Reilly Media, 2014.
[5] V. Buterin. (2014). *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. [Online]. Available: https://ethereum.org/en/whitepaper/

[6] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, pp. 1–32, 2014.
[7] (2021). *Solidity V0.8.4 Documentation*. [Online]. Available: https://docs.soliditylang.org/
[8] (2021). *Vyper V0.2.12 Documentation*. [Online]. Available: https://vyper.readthedocs.io/
[9] Q. Dupont, "Experiments in algorithmic governance: A history and ethnography of 'The DAO,' a failed decentralized autonomous organization," in *Bitcoin Beyond: Cryptocurrencies, Blockchains Global Governance*, M. Campbell-Verduyn, Ed. Evanston, IL, USA: Routledge, 2017, ch. 8, pp. 157–177.
[10] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons, "Smart contracts vulnerabilities: A call for blockchain software engineering?" in *Proc. Int. Workshop Blockchain Oriented Softw. Eng. (IWBOSE)*, Mar. 2018, pp. 19–25.
[11] P. Praitheeshan, L. Pan, J. Yu, J. K. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities: A survey," *CoRR*, vol. abs/1908.08605, pp. 1–21, Aug. 2019.
[12] J. Liu and Z. Liu, "A survey on security verification of blockchain smart contracts," *IEEE Access*, vol. 7, pp. 77894–77904, 2019.
[13] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using static analysis to find bugs," *IEEE Softw.*, vol. 25, no. 5, pp. 22–29, Sep. 2008.
[14] D. W. Loveland, *Automated Theorem Proving: A Logical Basis* (Fundamental Studies in Computer Science), vol. 6. North-Holland, Haarlem: Elsevier, 1978.
[15] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 2000.
[16] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, Apr. 1986.
[17] A. Pnueli, "The temporal semantics of concurrent programs," *Theor. Comput. Sci.*, vol. 13, pp. 45–60, 1981.
[18] R. Alur, T. Henzinger, and O. Kupferman, "Alternating-time temporal logic," *J. ACM*, vol. 49, no. 5, pp. 1–42, 2002.
[19] A. Lomuscio, H. Qu, and F. Raimondi, "MCMAS: An open-source model checker for the verification of multi-agent systems," *Int. J. Softw. Tools Technol. Transfer*, vol. 19, no. 1, pp. 9–30, 2017.
[20] (2020). *The Libra Blockchain*. [Online]. Available: http://https://developers.diem.com/main/docs/the-diem-blockchain-paper
[21] *Bitwala White Paper V.1.2.0*, Bitwala GmbH, Berlin, Germany, 2017.
[22] G. W. Peters, E. Panayi, and A. Chapelle, "Trends in crypto-currencies and blockchain technologies: A monetary theory and regulation perspective," 2015, *arXiv:1508.04364*.
[23] L. Collet, P. Laurent, S. Ramos, P. Martino, T. Chollet, and B. Sauvage. (2019). *Are Token Assets the Securities of Tomorrow?*. [Online]. Available: https://www2.deloitte.com/lu/en/pages/technology/articles/are-token-assets-securities-tomorrow.html
[24] M. Flore, *How Blockchain-Based Technology is Disrupting Migrants' Remittances: A Preliminary Assessment*, document EUR 29492 EN, Office Eur. Union, Luxembourg, U.K., 2018, doi: 10.2760/780817.
[25] M. Andoni, V. Robu, D. Flynn, S. Abram, and D. Geach, "Blockchain technology in the energy sector: A systematic review of challenges and opportunities," *Renew. Sustain Energy Rev.*, vol. 100, pp. 143–174, Feb. 2019.
[26] H. Min, "Blockchain technology for enhancing supply chain resilience," *Bus. Horizons*, vol. 62, no. 1, pp. 35–45, 2019.
[27] S. Saberi, M. Kouhizadeh, J. Sarkis, and L. Shen, "Blockchain technology and its relationships to sustainable supply chain management," *Int. J. Prod. Res.*, vol. 57, no. 7, pp. 2117–2135, Apr. 2019.
[28] A. E. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *Proc. IEEE Symp. Secur. Privacy*, Feb. 2016, pp. 839–858.
[29] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F.-Y. Wang, "Blockchain-enabled smart contracts: Architecture, applications, and future trends," *IEEE Trans. Syst., Man, Cybern. Syst.*, vol. 49, no. 11, pp. 2266–2277, Nov. 2019.
[30] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proc. 2nd Int. Workshop Emerg. Trends Softw. Eng. Blockchain*, 2019, pp. 8–15.
[31] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VERISMART: A highly precise safety verifier for Ethereum smart contracts," in *Proc. IEEE Symp. Secur. Privacy*, Feb. 2020, pp. 1678–1694.
[32] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev, "Securify: Practical security analysis of smart contracts," in *Proc. Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 67–82.
[33] R. O'Connor, "Simplicity: A new language for blockchains," in *Proc. Workshop Program. Lang. Anal. Secur.*, Oct. 2017, pp. 107–120.
[34] S. Amani, M. Bégel, M. Bortin, and M. Staples, "Towards verifying ethereum smart contract bytecode in Isabelle/HOL," in *Proc. 7th Int. Conf. Certified Programs Proofs*, 2018, pp. 66–77.
[35] D. Annenkov, J. B. Nielsen, and B. Spitters, "ConCert: A smart contract certification framework in Coq," in *Proc. 9th Int. Conf. Certified Programs Proofs*, 2020, pp. 215–228.
[36] T. Sun and W. Yu, "A formal verification framework for security issues of blockchain smart contracts," *Electronics*, vol. 9, no. 2, p. 255, 2020.
[37] Z. Nehai, P. Piriou, and F. F. Daumas, "Model-checking of smart contracts," in *Proc. Int. Conf. Internet Things*, Oct. 2018, pp. 980–987.
[38] A. Mavridou, A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-design smart contracts for Ethereum," in *Proc. 23rd Int. Conf. Financial Cryptogr. Data Secur.*, 2019, pp. 446–465.
[39] T. Osterland and T. Rose, "Model checking smart contracts for Ethereum," *Pervas. Mobile Comput.*, vol. 63, Oct. 2020, Art. no. 101129.
[40] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "NuSMV version 2: An opensource tool for symbolic model checking," in *Proc. 14th Int. Conf. Comput.-Aided Verification*, 2002, pp. 359–364.
[41] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.
[42] P. Madhusudan, W. Nam, and R. Alur, "Symbolic computational techniques for solving games," *Electron. Notes Theor. Comput. Sci.*, vol. 89, no. 4, pp. 578–592, 2003.
[43] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–12.

WONHONG NAM received the B.S. and M.Sc. degrees from Korea University, Seoul, South Korea, in 1998 and 2001, respectively, and the Ph.D. degree from the University of Pennsylvania, Philadelphia, PA, USA, in 2007.

From 2007 to 2009, he was a Postdoctoral Researcher with the College of Information Sciences and Technology, Pennsylvania State University, University Park, PA, USA. He is currently a Professor with the Department of Computer Science and Engineering, Konkuk University, Seoul. His research interests include formal methods, formal verification, model checking, automated planning, and web services composition.



HYUNYOUNG KIL received the B.S. and M.Sc. degrees from Korea University, Seoul, South Korea, in 1998 and 2001, respectively, the M.S.E. degree from the University of Pennsylvania, Philadelphia, PA, USA, in 2003, and the Ph.D. degree from The Pennsylvania State University, State College, PA, USA, in 2010.

She is currently an Assistant Professor with the Department of Software, Korea Aerospace University, Goyang, South Korea. Her research interests include computational theory, automated planning, web services composition, SOA, and web sciences.