(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
09 January 2025 (09.01.2025)

WIPO | PCT

(10) International Publication Number
WO 2025/010033 A1

(51) International Patent Classification:
H04L 9/32 (2006.01) *G06Q 30/00* (2023.01)

(21) International Application Number:
PCT/SG2024/050440

(22) International Filing Date:
05 July 2024 (05.07.2024)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
10202301940U 06 July 2023 (06.07.2023) SG

(71) Applicant: **NANYANG TECHNOLOGICAL UNIVERSITY** [SG/SG]; 50 Nanyang Avenue, Singapore 639798 (SG).

(72) Inventors: **LI, Yi**; c/o Nanyang Technological University, 50 Nanyang Avenue, Singapore 639798 (SG). **LIU, Ye**; c/o Nanyang Technological University, 50 Nanyang Avenue, Singapore 639798 (SG).

(74) Agent: **DAVIES COLLISON CAVE ASIA PTE. LTD.**; 10 Collyer Quay, #07-01 Ocean Financial Centre, Singapore 049315 (SG).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CV, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IQ, IR, IS, IT, JM, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, MG, MK, MN, MU, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, WS, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, CV, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SC, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, ME, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:
— *of inventorship (Rule 4.17(iv))*

Published:
— *with international search report (Art. 21(3))*

(54) Title: SMART CONTRACT SPECIFICATION MINING

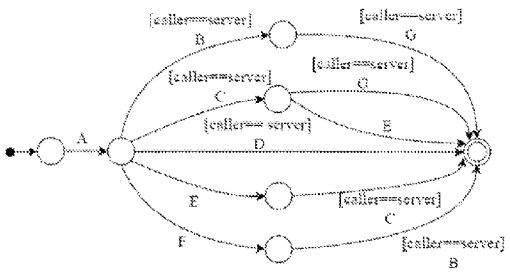


Figure 2 (b) The mined automaton.

(57) Abstract: Systems and methods for smart contract specification mining receive a smart contract transaction history, slice the history to produce invocation traces, and determine predicates from the history. Each predicate belongs to a condition of a function of the smart contract. A finite state machine is then constructed over states in the invocation traces and predicates, and counterexample-guided abstraction refinement is performed to extend the finite state machine satisfying a minimal existential abstraction property.

WO 2025/010033 A1

SMART CONTRACT SPECIFICATION MINING

Technical Field

The present invention relates, in general terms, to mining smart contract specifications. More particularly, the present invention relates to, but is not limited to, inferring smart contract specifications from past transactions histories.

Background

Blockchain technology has been developed rapidly in recent years, since the introduction of Bitcoin. Blockchain itself is a distributed ledger maintained and shared by a peer-to-peer (P2P) network. Blockchain has evolved into a platform supporting the deployment and execution of smart contracts.

Smart contracts are self-executing computer programs used to implement Decentralized Applications (DApps). Users interact with smart contracts by executing transactions on the blockchain, where the transaction data specifies inputs, such as the address of the sender, the smart contract function called, and parameter values.

Despite the high stakes involved in transacting using smart contracts, smart contracts are often developed in an undisciplined way. The existence of bugs and vulnerabilities compromises the reliability and security of smart contracts and endangers the trust of users. For example, many smart contracts restrict user access to functionalities by implementing certain access control policies. Many smart contracts do not comply with standard specifications and contain bugs. As a result, economic behaviour models can be exploited to attack the decentralized finance (DeFi) ecosystem with flash loans.

A major difficulty in validating the conformance of smart contracts, i.e., whether

the contract implementation adheres to the expected behaviours, is the lack of documented formal specifications. Formal specifications capture the expected contract behaviours, in terms of formal languages, based on a formal model with precise semantics. Specifications of a smart contract play a central role in describing, understanding, reasoning about contract behaviours, and detecting, through testing and verification, non-conformance issues such as functional bugs and security vulnerabilities.

Similar to traditional formal specifications, there are two general forms of smart contract specifications: (1) function-level program invariants used in testing, verification and runtime validation of smart contracts; and (2) contract-level behavioural specifications in the form of automata, which can be used to support contract synthesis model-based testing, design verification, and workflow verification.

Mining high-level automata-based specifications automatically for smart contracts, is a difficult problem. However, the way smart contracts behave poses new challenges for mining automata-based behavioural models. As they are usually deployed on public blockchain networks, smart contracts handle multiple user interactions simultaneously. Therefore, the execution traces recorded in contract transaction histories consist of interleaving events triggered by different user interactions and may belong to different sessions. Since there does not exist a standard approach for managing user sessions, the execution traces cannot be easily separated for independent interactions. Moreover, predicate abstraction is crucial in deriving compact but accurate automata. Yet, the choice of predicates remains challenging and is often tightly tied with the specific analysis tasks. The predicate abstraction techniques used in computing state abstractions must be tailored to take into account the specific data structures and runtime environments of smart contracts.

It would be desirable to provide a more reliable mechanism for mining smart contracts to develop specifications, or at least to provide a useful alternative to

pre-existing solutions.

Summary

As described herein, to mine more accurate automata specification efficiently for smart contracts, specification mining algorithms are powered by trace slicing and predicate abstraction. The contract specification mining process is preceded by a slicing of the transaction histories – e.g., via a parametric binding learned from the existing test suites. Relatedly, a slice of history is a sequence of inter-related transactions, e.g., all transactions related to one specific trade session. Smart contract transaction histories, being stored persistently on blockchain, record all past function executions since the contract deployment. To find suitable predicate candidates for state abstraction, a statistical inference technique generates a set of dynamic invariants, based on the transaction histories. A counterexample-guided abstraction refinement approach is used to perform a lazy state abstraction, and introduce minimal existential abstraction to ensure the automata specification is accurate and simple. Finally, the present automata specification subsumes all observed invocation sequences and at the same time preserves its generality.

The present invention derives high-level behavioural automata with program invariants from smart contracts and their transaction histories.

Disclosed herein is a method for smart contract specification mining, comprising:

receiving a transaction history for a smart contract;

slicing the transaction history to produce one or more invocation traces;

determining one or more predicates, each predicate belonging to a condition of a function of the smart contract;

constructing a finite state machine over states in the one or more invocation traces and the one or more predicates; and

performing counterexample-guided abstraction refinement to extend the finite state machine satisfying a minimal existential abstraction property.

Also disclosed is a system for smart contract specification mining, comprising:

memory; and

at least one processor (processor(s)),

the memory storing instructions that, when executed by the processor(s), cause the processor(s) to:

receive a transaction history for a smart contract;

slice, using a history slicing module, the transaction history to produce one or more invocation traces;

determine, using a predicate extractor, one or more predicates, each predicate belonging to a condition of a function of the smart contract;

construct, at a finite state machine module, a finite state machine over states in the one or more invocation traces and the one or more predicates; and

perform, using a refinement module, counterexample-guided abstraction refinement to extend the finite state machine satisfying a minimal existential abstraction property

Brief description of the drawings

Embodiments of the present invention will now be described, by way of non-limiting example, with reference to the drawings in which:

Figure 1 illustrates how interleaving interactions are separated from past transaction histories.

Figure 2, comprising images 2a and 2b, shows a data structure used in a smart contract implementation (image 2a) and a mined automation in accordance with present teachings (image 2b).

Figure 3 shows a two-state automaton containing potential spurious paths.

Figure 4 shows an automaton containing spurious paths.

Figure 5 shows a final automaton containing no spurious paths.

Figure 6 is a schematic of a system for implementing the present methodology.

Detailed description

Described is a specification mining approach to infer contract specifications from past transaction histories. The present approach derives high-level behavioural automata of function invocations, accompanied by program invariants statistically inferred from the transaction histories.

To that end, disclosed is a method for smart contract specification mining. This approach of the method is illustrated, though not limited to, with the GameChannel contract from a DApp called Dicether. Dicether is a decentralized casino application on Ethereum, relying on a smart contract to provide an open, secure, and fair gaming experience. From a user perspective, a smart contract is a set of interface functions. The functions can be invoked to execute contract code. If these interface functions be represented as base events: ξ is the set of interface function names and $e \in \xi$ corresponds to a contract function. The execution of e accepts parameters (denoted as X), including the user-provided function inputs (X1) and the contract state variables (X2) stored on the blockchain.

A new game is created by calling the contract function createGame – i.e., a transaction over the smart contract is initiated. When approaching the end of a game, an admin user may invoke the serverEndGame function to close the game. With reference to Figure 1, the method, embodied by schematic 100, includes receiving a transaction history 102 for a smart contract. The transaction

histories 102 are each a sequence of transactions. Each transaction can be decoded as a contract function invocation. The functions are labelled in Figure 1 as A to G. Each transaction history is then sliced (at Trace slicer 104) to produce one or more invocation traces 106. The slicing function performed by the trace slicer 104 performs is determined by interaction patterns observed in test suites, on the transaction history, to produce a set of independent invocation sequences. For GameChannel, there are six game interaction (invocation) sequences shown in Figure 1, corresponding to six user sessions. For instance, $user_1$: A(gameId:1) indicates the invocation sequence for createGame by $user_1$ for creating a game with index 1. For simplicity, some values and variables are omitted from Figure 1. A function invocation may change the values of state variables, thus updating contract states.

There are two types of trace considered in the present disclosure, namely Non-Parametric and Parametric traces. For Non-Parametric Events and Traces, let ξ be a set of (nonparametric) events, called base events or simply events. An ξ-trace, or simply a (non-parametric) trace is any finite sequence of events in ξ, that is, an element in ξ^*. If event $e \in \xi$ appears in trace $w \in \xi^*$, then $e \in w$. For Parametric Events and Traces, let X be a set of parameters and let V be a set of corresponding parameter values X in parametric traces. Let DX , DX1, and DX2 be the corresponding domains. If ξ is a set of base events as for Non-Parametric traces, then $\xi(X)$ is the set of corresponding parametric events $e(\theta)$, where e is a base event in ξ and θ is a partial function in $[X \rightarrow V]$. A parametric trace is a trace with events in $\xi(X)$, that is, a word in $\xi(X)^*$. Given a smart contract, let $\xi(X)$ be the set of all function executions, and any function invocation sequence can be represented as a parametric trace (word) in $\xi(X)^*$.

After slicing, one or more predicates are determined, wherein each predicate belongs to a condition of a function of the smart contract. In the present example, 11 predicates have been identified, namely:

P_1: status = 0

P_2 : status = 1
P_3 : status = 2
P_4 : status = 3
P_5 : status > 3
P_6 : roundId = 0
P_7 : roundId > 0
P_8 : endInitiatedTime = 0
P_9 : endInitiatedTime > 0
P_{10} : stake = 0
P_{11} : stake > 0

Table 1 shows the pre/post-conditions of GameChannel.

Functions	**Preconditions**	**Postconditions**
createGame	$P_1 \wedge P_6 \wedge P_8 \wedge P_{10}$	$P_2 \wedge P_6 \wedge P_8 \wedge P_{11} \wedge$ (stake = msg.value)
serverEndGameConflict	$(P_2 \vee P_3) \wedge$ (roundId > 0) $\wedge$(caller = server)	$P_1 \vee (P_4 \wedge P_7 \wedge P_9)$
serverCancelActiveGame	$(P_2 \vee(P_3 \wedge P_6))\wedge$(caller = server)	$P_1 \vee (P_4 \wedge P_9)$
serverEndGame	$P_2 \wedge$ (caller = server)	P_1
userCancelActiveGame	$P_2 \vee (P_4 \wedge P_6)$	$P_1 \vee (P_3 \wedge P_9)$
userEndGameConflict	$(P_2 \vee P_4) \wedge$ (roundId > 0)	$P_1 \vee (P_3 \wedge P_7 \wedge P_9)$
serverForceGameEnd	$P_4 \wedge$ (caller = server)	P_1

Table 1: pre/post conditions of GameChannel

Predicates partition the domain of the state. In Figure 1, the first game is created by $user_1$ and after a while ended by $user_2$. Three game states, s0, s1, and s2, are involved. The second game is created by $user_3$, and later cancelled by $user_2$ and $user_3$ via serverCancelActiveGame and userCancelActiveGame,

respectively. From these invocation sequences 106 and predicates, we are able to capture the behaviours of a smart contract using a labelled transition system (LTS) that accepts all its function invocation sequences. A smart contract is a labelled transition system (S, s0, Σ, δ) where S is a set of possibly-infinite states, $S \subseteq DX1$, $s0 \in S$ is an initial state, Σ is a possibly-infinite alphabet, $\Sigma \subseteq \xi(X)^*$, and $\delta \subseteq S \times \Sigma \times S$ is a set of transitions. An LTS can be represented more compactly by abstracting it into an extended finite state machine (EFSM). The finite state machine (or EFSM) is annotated with function pre/post-conditions, as a specification of the observed contract behaviours, and is constructed over states in the one or more invocation traces and the one or more predicates. Specifically, in GameChannel, each function pre/post-condition consists of a set of predicates either relevant to game state variables or function input parameters.

An EFSM in the present disclosure is defined as a tuple – presently, a 6-tuple $(Q, q_0, \Sigma', G, U, T)$ where, Q is a finite set of symbolic states under a predicate abstraction $\alpha : S \rightarrow Q$, $q_0 \in Q$ is the initial symbolic state, Σ' is a finite alphabet defined as $\Sigma' \subseteq \xi^*$, G is a set of guarding functions g_i such that $g_i: D_X \rightarrow \{True, False\}$, – U is a set of update function u_i such that $u_i: D_X \rightarrow D_X$, T is a transition relation, $T : Q \times G \times \Sigma \rightarrow U \times Q$.

Counterexample-guided abstraction refinement can then be performed to extend the finite state machine satisfying a minimal existential abstraction property. Abstraction refinement can be looped to iteratively produce the extended finite state machine. Counterexample-guided extraction involves computing state abstractions for states of the finite state machine. To compute state abstractions, predicate abstraction is typically used. Predicate abstraction is a function to create a partition of the domains of data types. A widely used predicate abstraction for integer domain is {neg, zero, pos} which represent negative, zero and positive numbers respectively. However, there could be many EFSM candidates that an LTS can be abstracted into. For this reason, the

present methodologies employ a minimal existential abstraction to obtain a compact EFSM.

As mentioned above, the finite state machine is defined as a tuple, presently EFSM = (Q, q_0, Σ, G, U, T) is the minimal existential abstraction of LTS = (S, s_0, Σ, δ) with respect to $\alpha : S \rightarrow Q$ *iff*,

$$\exists s_0 \in S \cdot \alpha(s_0) = q \Leftrightarrow q = q_0 \quad (1)$$

$$\exists (s_0, e_0(\theta_0), s1), \ldots, (s_{n-1}, e_{n-1}(\theta_{n-1}), s_n) \in \delta \cdot$$
$$\alpha(s_0) = q_0 \wedge \alpha(s_1) = q_1 \wedge \cdots \wedge \alpha(s_{n-1}) = q_{n-1} \wedge \alpha(s_n) = q_n$$
$$\Leftrightarrow (q_0, g_i, e_0, u_i, q_1), \ldots, (q_{n-1}, g_j, e_{n-1}, u_j, q_n) \in T \quad (2)$$

Under the minimal existential abstraction, (1) the initial concrete state can be mapped to the initial symbolic state in the EFSM, and vice versa, and (2) every concrete path is preserved in the ESFM, and every symbolic path in the ESFM has at least a corresponding concrete path.

Image 2a shows an example data structure – e.g. that used in GameChannel. In this example, server, gameIdCntr, and gameIdGame maintain information about the game manager, the number of created games, and all game state information, respectively. The game state variables include status, roundId, endInitiatedTime, and stake. The variable status being ENDED (0) indicates that a game either has not been created or has already been terminated; roundId refers to the number of current gambling rounds; endInitiatedTime records when a game is required to terminate itself as per users' requests; and stake keeps the funds that a player deposits as the bid when creating a game. All parameter values, including contract state variables and user-provided function inputs, can be decoded from blockchain transactions. Accordingly, dynamic invariants can be inferred as candidates for predicates on function pre/post-conditions. Table 1 shows the predicates identified in this example, those predicates partitioning the domain of each game state variable.

By default, all contract state variables are initialized zero, so s_0 can be represented by $P_1 \wedge P_6 \wedge P_8 \wedge P_{10}$. These predicates also form the pre- and post-conditions in Table 1 where some other parameter predicates in the pre- and post-conditions are over function input parameters, i.e., " roundId", and caller of the function, i.e., "caller". For the createGame function, its precondition is that all the values of variables, namely, status, stake, roundId, and endInitiatedTime, are zero; and its postcondition is that when createGame finishes, the variable status is set to ACTIVE (1) and the deposited stake is greater than zero and equals to the transferred fund, i. e., msg.value.

Image 2b shows the mined automaton of the smart contract underpinning GameChannel. The correctness of the automaton has been verified using the ground truth specification of GameChannel. The mined automaton has seven symbolic states, represented by circles in image 2b. Only createGame can be called in the initial state, hence the single path from the initial symbolic state to the next state under function A. Furthermore, when the caller is server, he/she is allowed to call serverEndGame to terminate the game and move the current state to the final state.

Such an automaton captures the common usages of GameChannel and its permission policies, thus being a likely contract specification. As for automata construction, the tool SmCon uses a CEGAR-like approach, as discussed below. This involves a lazy abstraction – i.e., no refinement of the predicate abstraction unless necessary. To obtain an ESFM, SmCon takes the sliced independent invocation sequences and inferred function pre/post-conditions as input. An initial automaton is constructed containing only two states, which is then revisited to recognize spurious symbolic paths that have no support – i.e., those paths without a corresponding concrete invocation sequence in the past observations (historical transactions).

The automaton is then refined to eliminate the spurious paths. This can be done via splitting larger states into smaller states or sub-states, or by removing

unreachable transitions. This process is repeated until no spurious paths remain in the automaton. Thus constructing the finite state machine involves assuming an automaton comprising a set of states and a set of transitions between states in the set of states, then iteratively refining paths between states to excise spurious paths.

Smart contracts simultaneously accept inputs from multiple users. Contract executions in such a setting result in a linear transaction history consisting of interleaving execution traces triggered through multiple user interactions/sessions. To record data owned by different users, many smart contracts maintain a collection of custom data objects, indexed by user(session)-specific parameters. For example, the GameChannel contract maintains many concurrent game instances as state variables. To interact with a particular game instance, a user needs to specify the value of its gameId, through input parameters of the transaction – refer to "gameId" in Figure 1.

To determine the contract specifications from transaction histories with mixed interactions, the transactions histories need to be sliced into independent traces for each game instance. Functionally, for trace slicing a parametric trace, given a parametric trace $\tau \in \xi(X)^*$ and a parametric binding θ in $[X \rightharpoonup V]$, let the θ-trace slice $\tau \restriction \theta \in \xi^*$ be the nonparametric trace defined as:

$$\epsilon \restriction \theta = \epsilon,$$

where ϵ is the empty trace/word, and:

$$(\tau e(\theta')) \restriction \theta = \begin{cases} (\tau \restriction \theta)e, & \text{if } \theta' \sqsubseteq \theta \\ \tau \restriction \theta, & \text{otherwise} \end{cases}$$

where θ' is assumed to be less informative than θ, written $\theta' \sqsubseteq \theta$ *iff* for any $x \in X$, if $\theta'(x)$ is defined then $\theta(x)$ is also defined and $\theta'(x) = \theta(x)$.

In accordance with present teachings, a transaction history of smart contract is a parametric trace. Trace slicing slices the history into a set or plurality of independent invocation sequences via certain parametric bindings (e.g., θ). Moreover, slicing the transaction history can involve performing a plurality of slices, the method comprising, for each slice, filtering out all events that are irrelevant to a parameter instance of a parameter in the parametric bindings. A trace slice τ /θ first filters out all the parametric events that are irrelevant to the parameter instance θ. An event is irrelevant to a parameter instance if the event does not take the parameter instance as input, manipulate the parameter instance nor output the parameter instance. A trace slice also forgets the parameter bindings of parametric events. Thus, slicing the transaction history comprises performing a plurality of slices, and the method then involves, for each slice, forgetting parameter bindings for parametric events.

As a result, a trace slice is non-parametric and merely a list of base events. To find parametric bindings, the relation between different events should be ascertained, or function invocations in smart contracts. Such parametric bindings can be inferred – e.g., from existing test suites that demonstrate typical usage scenarios and user interaction patterns. In some embodiments, a group of related functions and the parameter values they share can be observed in a unit test. For example, the test suites for GameChannel contain many test cases where game objects are explicitly specified by the "gameId" variable in each contract function. Therefore, the transaction history can be sliced according to the corresponding values of "gameId" to generate a set of independent game invocation sequences.

The choice of predicates is crucial for computing good state abstractions. To this end, statistically likely pre- and post-conditions of contract functions as considered candidate predicates. The method of smart contract specification mining can therefore involve decoding, from the transaction history and before and after each function invocation, one or both of contract state variables values and user-provided inputs (function inputs). Dynamic invariants can then be

statistically inferred for each function, each predicate comprising a dynamic invariant. The inferred dynamic invariants hold for all observed invocations in the past transaction histories. More specifically, a predicate template is defined as "$x \triangleright\triangleleft y$", where $x \in X$ is a parameter, and $y \in X \cup K$ is either a parameter or constant, and $\triangleright\triangleleft$ is an operator from the set {=, !=, >, <, <=, >=}. The template is instantiated on all successful transactions, which are not reverted during executions, and the instances which always hold are kept as predicate candidates for either function pre- or post-conditions. The predicates defined over state variables are used in constructing the symbolic states Q in the EFSM.

Smart contracts employ large integer type systems and some unique data structures when compared with traditional programs, such as "mapping", which defines a key-value collection. Therefore, the existing integer invariant templates must be extended, and new invariant templates must be added. In some smart contracts, a transaction reversion will be raised if contract execution fails to satisfy any execution condition. In other words, the target function's pre- and post-conditions are satisfied in all successful past transactions. But since the transaction history may be limited, the inferred pre- and post-conditions are likely to hold, which is good enough to serve as predicate candidates.

The over-generalization of the inferred function pre/post-conditions is the main difficulty for their direct use in mining high-level automata specifications. To address this problem, a counterexample guided abstraction refinement (CEGAR)-like approach is used to mine automata specifications with predicate abstraction.

To mine a precise specification, a precise state abstraction α is computed. This creates partitions on the contract state variables. The abstraction function α is implicitly computed following the paradigm of counterexample-guided abstraction refinement. In some embodiments, the specification mining algorithm is defined using the following rules:

$$\frac{}{(q_0 \leftarrow \bigwedge_{x \in X_1} x = 0, Q \leftarrow \{q_0, \neg q_0\}, \Sigma, G \leftarrow \{g_m\}_m, U \leftarrow \{u_m\}_m, T \leftarrow \emptyset)} \textsc{Init}$$

$$\frac{(q_0, Q, \Sigma, G, U, T) \qquad \begin{array}{c}\exists\, q_i, q_j \in Q \cdot (q_i \wedge g_m) \wedge (q_j \wedge u_m) \\ \nexists\, t \cdot (q_i, g_m, e_m, u_m, q_j) \in T\end{array}}{(q_0, Q, \Sigma, G, U, T \leftarrow T \cup \{t\})} \textsc{Construct}$$

$$\frac{EFSM : (q_0, Q, \Sigma, G, U, T) \qquad \begin{array}{c}\exists\, \pi_n : q_0 l_1 l_2 \cdots l_n q_n \in EFSM \;\exists\, \text{concretize}(\pi_n) \in LTS_b \\ \exists\, \pi_{n+1} : \pi_n \oplus l_{n+1} q_{n+1} \in EFSM \;\nexists\, \text{concretize}(\pi_{n+1}) \in LTS_b\end{array}}{EFSM \leftarrow \textsc{SplitRemove}(q_n, l_{n+1}, EFSM)} \textsc{RmPath}$$

The algorithm takes as input past observations of concrete invocation sequences and inferred function pre/post-conditions. When the algorithm terminates, it produces an EFSM containing no spurious states and transitions.

The Init rule initializes a preliminary extended finite state machine containing two states: q_0, referring to $\bigwedge_{x \in X1} x = 0$ that all state variables are valued zero, and $\neg q_0$ for the remaining cases. The guard function G and update function U are directly instantiated by the inferred function pre- and postconditions, respectively. The transition relation set *T* is initialized to be empty. The Construct rule is the applied to add theoretically feasible state transitions to the automaton. A state transition is theoretically feasible if and only if it satisfies the logical conjunction of symbolic states and function preconditions or post-conditions.

Algorithm 1 SplitRemove($q_n, t_{n+1}, EFSM$)

Let $(q_0, Q, \Sigma, G, U, T) = EFSM$
Let $t_{n+1} = (q_n, g_m, e_m, a_m, q_{n+1}) \in T$ ▷ a transition from state q_n to q_{n+1} by the invocation to function e_m where g_m, a_m are its precondition and post-condition, respectively.
$\hat{q}_1 = q_n \wedge g_m$
$\hat{q}_2 = q_n \wedge \neg g_m$
if SAT($\hat{q}_1$) $\wedge$ SAT($\hat{q}_2$) **then** ▷ q_n is splittable with g_m.
  $Q \leftarrow Q \setminus q_n \cup \{\hat{q}_1, \hat{q}_2\}$ ▷ replace q_n with two new states.
  T removes transitions starting or ending with q_n
else
  if $\nexists\, \pi'_n, \pi'_n \in EFSM \wedge$concretize$(\pi'_n \oplus t_n) \in LTS_h$ **then** ▷ t_n is not reachable
    $T \leftarrow T \setminus t_{n+1}$ ▷ remove unreachable transition
  else
    Let $S_{q_n|\pi'_n}$ be the set of all the concrete states of q_n in the history LTS_h, which are visited by the observed invocation sequences of π'_n.
    $Q \leftarrow Q \setminus \{q_n\} \cup \{\text{Pred}(S_{q_n|\pi'_n}), q_n \wedge \neg\text{Pred}(S_{q_n|\pi'_n})\}$
    T removes transitions starting or ending with q_n
return $(q_0, Q, \Sigma, G, U, T)$ ▷ return the resulting automaton

The resulting automaton could be over-generalized such that it includes spurious state transition paths. The RmPath rule is thus applied, following Algorithm 1 to eliminate those spurious state transition paths that are not supported in the concrete observations. Algorithm 1 performs one of two actions to exclude spurious paths – state splitting or transition removal. Thus, counterexample-guided abstraction refinement comprises identifying a number of spurious paths resulting from applying the invocation traces to the finite state machine, and extending the finite state machine, removing spurious paths, by either splitting at least one state in the set of states, into two or more states, or by removing at least one transition from the set of transitions, to reduce the number of spurious paths. This abstraction process can be repeatedly performed until the number of spurious paths is zero. These above rules would thus be applied many times according to a fair scheduling. Thus, when the algorithm terminates, it produces an extended finite state machine, containing no spurious states or transitions.

The resulting automaton does not allow loop transitions according to the RmPath rule. However, this kind of automaton may not be able to produce precise and useful contract specifications. Because many smart contracts have behaviour cycles, it is preferred to have loop transitions in the resulting automaton.

To improve the precision and usability of contract specification mined using the present teachings, the range of path selection may be limited when applying RmPath – e.g., a loop transition may only be covered once in any selected path. For example, a state transition path $q - Event_a - q - Event_a - q$ is excluded given the path loops twice. With this minor modification to the RmPath rule, the resulting automaton allows loop transitions so that it may express cycles.

Use GameChannel as an example – initially, rule Init is applied. As shown in Figure 3. The resulting automaton has two states: q_0 and q_1. For the initial state q_0, all state variables are valued zero, namely $P_1 \wedge P_6 \wedge P_8 \wedge P_{10}$, while q_1 covers all the remaining values of the state variables, namely $\neg(P_1 \wedge P_6 \wedge P_8 \wedge P_{10})$. For each function m, its precondition g_m is added as the guard function set G, and the post-condition u_m is added to the update function set U, which we omit from these diagrams for simplicity.

The Construct rule adds all possible transitions to the automaton. It is easy to see in Figure 3 that the resulting automaton contains all the observed concrete function invocation sequences. However, spurious behaviours may have been allowed in the automaton. For example, the symbolic path $q_0 - A - q_1 - D - q_1 - C - q_1$ is spurious, as it has no supported concrete invocation sequence within the observations. Therefore, the RmPath rules is applied to eliminate such spurious paths. In rule RmPath, the concretize function is used on a symbolic path to represent one of its possible concrete invocation sequences. For the spurious path $q_0 - A - q_1 - D - q_1 - C - q_1$, the automaton needs to refine itself by either splitting the state q_1 (q_n in Line 6 of Algorithm 1) or removing the transition $q_1 - C - q_1$ (t_{n+1} in Line 10 of Algorithm 1).

Algorithm 1 shows the present approach to removing a spurious path via state splitting or transition removal. For the state q_1 and the transition $q_1 - C - q_1$ in Figure 3, first an attempt is made to split the state q_1 with the precondition of function C, namely $P_2 \vee (P_3 \wedge P_6)$ of serverCancelActiveGame. The decision procedure SAT (Line 5 of Algorithm 1) decides if a symbolic state is feasible. Due to the fact that the two newly created states (Lines 3–4 of Algorithm 1) for q_1 are feasible, q_1 is replaced with the new states (Line 6 of Algorithm 1) and the outdated transitions starting or ending with q_1 are removed (Line 7 of Algorithm 1). The modified automata structure will be returned (Line 15 of Algorithm 1) and when applying the Construct rule again. A new automaton is then obtained per Figure 4, which now has three states.

In Algorithm 1, it is possible that q_n is not splittable with the function precondition (Lines 8–14 of Algorithm 1). When t_{n+1} has never been visited by replaying all the concrete invocation sequences on the automaton (Line 9 of Algorithm 1), t_{n+1} can be safely removed to eliminate the spurious path. Otherwise, state q_n will be split with an equivalent predicate representing a partial set of the visited concrete states (such as s_0 to s_{17} of Figure 1) along a symbolic path π' (Lines 12–13 of Algorithm 1). Outdated transitions relevant to q_n will also be removed (Line 14 of Algorithm 1).

Figure 4 also contains spurious paths, such as $q_0 - A - q_{11} - E - q_{11} - E - q_{12}$, which have to be eliminated. By applying the aforementioned rules many times, an automaton can be mined as shown in Figure 5, which has the exact automata structure as image 2b, except for the guard condition that has been excluded for simplicity. Apart from the common usage scenarios, root semantics of each state in the automaton can be understood. Each symbolic state is represented by a set of predicates. As described before, q_0 is represented by $P_1 \wedge P_6 \wedge P_8 \wedge P_{10}$. In the automaton, q'_{11} is represented by P_2 that says status is ACTIVE, q'_{12} is represented by $P_4 \wedge P_6$ that says status is SERVER INITIATED END and roundId is zero, q_3 is represented by $P_3 \wedge P_6$ that says status is USER INITIATED END

and roundId is zero, q_4 is represented by $P_4 \wedge P_7$ that says status is SERVER INITIATED END but roundId is larger than zero, q_5 is represented by $P_3 \wedge P_7$ that says status is USER INITIATED END but roundId is larger than zero, and q_6 having no outcoming transitions suggests it the final state whose predicates cover all the remaining cases. Such automata specifications precisely model smart contract semantics, which is helpful for understanding contract behaviours.

To ensure the termination of the present methodology, applications of the rules have to follow a fair scheduling. A fair scheduling is an infinite sequence of actions: $a_1, a_2, \ldots, a_n$, where $a_i \in$ {Construct, RmPath} and the following conditions apply:

(1) Construct is the initial action – this enables the initial construction of the automaton.

(2) if new states are added, Construct keeps applying until it is inapplicable – this ensures the integrity of any subsequent automata construction.

(3) RmPath appears infinitely often – ensures that every spurious path of the automaton will eventually be removed.

Under the above conditions, the assumption is that the algorithm terminates when RmPath and Construct are both no longer applicable.

The present specification mining algorithm terminates after a finite number of actions in any fair scheduling. For LTS_h which is equivalent to the observed concrete invocation sequences, let N_s be the number of concrete states and N_p be the number of all concrete invocation sequences. For each application of rule RmPath, it will take no more than N_{p+1} times to traverse the resulting automaton to find such a spurious path π if it exists. Therefore, RmPath itself is terminated. As discussed in Algorithm 1, a transition may be removed from the resulting automaton. According to the fairness conditions, RmPath will keep applying until it splits a state to generate two new fine-grained states. Suppose the current automaton has n symbolic states and k symbolic state transitions, then $k \leq n \times$

$|M| \times n$, where $|M|$ is the number of contract functions. Therefore, the times for RmPath to find one more reachable states is no more than $n \times |M| \times n$. So it takes no more than $N_s \times max(n) \times |M| \times max(n)$ times to find all reachable states, where $max(n) \leq N_s$.

The number of RmPath actions in any scheduling cannot exceed $(N_{s+1}) \times max(n) \times |M| \times max(n)$. Finding all reachable states does not exceed $N_s \times max(n) \times |M| \times max(n)$. RmPath will then be applied to remove any unreachable transition if applicable. This will apply no more than $max(n) \times |M| \times max(n)$ times. Finally, RmPath will be inapplicable and the algorithm will terminate with no more than $(N_{s+1}) \times max(n) \times |M| \times max(n)$ actions of RmPath.

The automaton resulting from the above process (ignoring loop transitions) is a minimal existential abstraction of LTS_h. To show this is the case, the automaton must satisfy Condition (1) and Condition (2). Condition (1) holds because by default, all uninitialized state variables of smart contracts are valued zero, and s_0 is the initial concrete state where all state variables are valued zero. The automaton takes the zero-valued predicates of all state variables as the initial symbolic state q_0. In this setting, $\{s_0\} = \alpha^{-1}(q_0)$. Thus, Condition (1) is satisfied. Condition (2) can be verified by proving: (i) every observed concrete invocation sequence (i.e., a concrete path in LTS_h) is preserved in the resulting automaton; (ii) every symbolic path of the resulting automaton has a corresponding concrete path in LTS_h, - i.e., no symbolic path is spurious. According to rule Construct, (i) holds in the resulting automaton. Regarding (ii), suppose the resulting automaton contains a spurious path $\pi : q_0 t_1 t_2 \cdots q_n \in$ EFSM where $\nexists$ concretize(π) $\in LTS_h$. The RmPath rule will be applied to remove π either via state splitting or transition removal at finite actions. Thus, there is no such spurious path π, which is in contradiction with the assumption. Therefore, (b) is proved. Hence, Condition (2) is satisfied by the resulting automaton.

The abovementioned processes are executed in a computer system such as system 600 of Figure 6. The system 600 may be any suitable computer, including that set out below in relation to the experiments.

While modules and components of the system 600 are shown, and labelled, separately, it will be appreciated that they can be combined in any suitable combination, and broken down into smaller components as needed, provided the functionality set out herein is preserved. Moreover, each module may be located on a common server, or distributed across multiple servers. In addition, each component may be implemented in software, hardware or a combination of the two.

The system 600 comprises memory 602 that stores computer executable instructions (program code 604) that is executable by processor(s) 606 to implement the above methodology. The system 600 also includes an interface (I/O module 608) for receiving data (e.g., transaction histories) and outputting mined specifications. The system 600 further comprises a history slicing module 610, predicate extractor 612, finite state machine (FSM) module 614 and refinement module 616. The history slicing module 610 slices the transaction histories to produce invocation traces. The predicate extractor determines predicates belonging to a condition of a function of the smart contract received by I/O module 608. The FSM module 614 constructs a FSM over states in the invocation traces and the predicates, and the refinement module 616 performs counterexample-guided abstraction refinement to extend the finite state machine satisfying a minimal existential abstraction property.

Thus, the methodology schematically reflected in Figure 1, and the system 600 that implements that methodology, enable specification mining for smart contracts. The approach uses a CEGAR-like mechanism to mine automata specifications based on past transaction histories. The mined specifications capture not only the allowed function invocation sequences, but also the inferred program invariants describing contract semantics precisely. Such contract

specifications are useful in contract understanding, testing, verification, and validation.

Experiments

Algorithm 1, named SmCon, was written in Python code. Through experiments, SmCon was evaluated to answer the following questions:

1. How accurately and efficiently does SmCon mine smart contract specifications?
2. How useful are the mined specifications for smart contract test case generation?

Table 2: DApp smart contracts benchmark.

Contracts	DApp	Descriptions	LoC	Year
GameChannel	Dicether	gambling game	1,483	2019
CryptoPunksMarket	Crypto Punks	non-fungible token collection	246	2017
SaleClockAuction	Crypto Kitty	digital auction	565	2018
SuperRare	SuperRare	digital art market	688	2018
RpsGame	0xfair[Rock-Paper-Scissors]	guess game	317	2018
MoonCatRescue	MoonCatRescue	crypto-powered game	331	2017

To evaluate SmCon, the selected accuracy metric measures the similarity between the mined automata specification and the ground truth, considering both precision and recall. Precision is defined as the percentage of sequences generated by the mined automata that are accepted by the ground truth, while recall is the percentage of sequences generated by the ground truth that are accepted by the mined automata. The F1-score was used to measure the overall accuracy, which is defined as: $F1 = \frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$. It is impractical to generate all possible sequences for automata with loops. Therefore, the length of the generated sequences was limited for accuracy calculation.

SmCon was evaluated against popular Ethereum DApp smart contracts as shown in Table 2. Ground truth specifications were obtained as well as the transaction data (historical transactions) from Etherscan since contract deployment and raw results are available at: https://sites.google.com/view/smcon/.

Regarding accuracy and efficiency, SmCon was evaluated on eight real-world smart contract instances running on Ethereum, shown in Table 3. SmCon was compared with a baseline approach, RPNI-MDL, implemented based on LearnLib, which is a passive learning algorithm to learn automata from positive trace examples. The recognition of negative examples for smart contracts is difficult since the invocation to a function may fail in some cases but succeed in others. With only positive training data, RPNIMDL was used as the baseline for our automata mining.

Table 3: The comparison result of mined contract specifications. (The trace sequence length limit for automata comparison is 3 or 6.)

Contracts	Ideal		RPNI-MDL						SmCon						
	$\|Q\|$	$\|T\|$	$\|Q\|$	$\|T\|$	precision l=3	precision l=6	recall l=3	recall l=6	F_1-score l=3	F_1-score l=6	$\|Q\|$	$\|T\|$	precision l=3	precision l=6	recall l=3	recall l=6	F_1-score l=3	F_1-score l=6
GameChannel-0x7e			5	8	0.69	0.26	0.53	0.53	0.60	0.35	7	14	1.00	1.00	0.53	0.53	0.69	0.69
GameChannel-0x89	7	17	5	10	0.75	0.13	0.71	0.71	0.73	0.22	9	17	1.00	1.00	0.59	0.59	0.74	0.74
GameChannel-0xac			4	8	0.69	0.26	0.53	0.53	0.60	0.35	8	11	1.00	1.00	0.47	0.47	0.64	0.64
CryptoPunksMarket	5	18	5	11	0.83	0.19	0.43	0.64	0.57	0.07	6	11	1.00	1.00	0.41	0.03	0.58	0.06
SaleClockAuction	2	5	4	7	0.50	0.35	0.25	0.06	0.33	0.10	2	3	1.00	1.00	0.25	0.06	0.40	0.12
SuperRare	5	20	8	14	0.50	0.47	0.36	0.01	0.45	0.02	9	16	1.00	1.00	0.46	0.04	0.63	0.07
RpsGame	4	5	5	4	0.71	0.31	1.00	1.00	0.83	0.48	5	7	1.00	1.00	1.00	1.00	1.00	1.00
MoonCatRescue	10	58	99	134	0.65	0.72	0.29	0.00	0.40	0.00	9	24	0.99	0.95	0.37	0.04	0.52	0.07

Table 3 shows the comparison result. The first column shows the names of the subject contract instances. The next two columns show the number of states and the number of transitions of the ideal automata, namely the corresponding ground truth. The five columns under RPNI-MDL list its results, including the number of states and transitions, precision, recall and F1-score of the mined automata. Similarly, the next five columns show the results of SmCon. Table 3 shows that SmCon outperforms RPNI-MDL for all cases in precision when the trace length limit for automata comparison is set to be 3 or 6. The precision of the mined automata by SmCon is 100 % except for MoonCatRescue since spurious loop transitions were allowed in the present automata construction process – note: the method may include identifying a smart contract type and selectively permitting (or disabling) loop transitions. The precision of RPNI-MDL on the subject contracts is mostly lower than 85% when the trace length limit

is 3 and lower than 50% when the trace length limit is 6. The recall of SmCon is comparable with RPNI-MDL.

When the trace length limit is set to 6, the recall on some contracts is low for both techniques. The main reason is that, for some contracts, many designed behaviours have never been exercised, thus could not be mined from the transaction histories. For example, the contracts CryptoPunksMarket, SaleClockAuction, SupeRare, and MoonCatRescue, have the lowest recall when the trace length limit is set to 6. Specifically, in SaleClockAuction, we did not observe any invocation to its function createAuction taking four parameters in the transaction histories.

The F1-score indicates that SmCon outperforms RPNI-MDL for nearly all cases. Notice that GameChannel has three deployed instances 0x7e, 0x99, and 0xae, which share the same ground truth specification, but have different transaction histories. The precision, recall, and F1-score for their mined automata by SmCon are reasonably similar to each other. This shows that variations in transaction histories do not have a significant impact on SmCon's performance. In terms of computation time, SmCon on average took about 3 minutes while RPNI-MDL took around one second per contract. In summary, SmCon achieves relatively high precision, comparable recall, and good F1-score in the automata mining from past transaction histories of smart contracts.

Regarding how useful the methodology is, SmCon was used to help generate test cases to complement the existing test suites. Various test sequences were derived from the mined extended finite state machine by SmCon. Each test sequence walks through a symbolic path of the automaton.

Table 4: The comparison result of state and transition coverage.

Contracts	Ideal		Test suites		SmCon		Test suites + SmCon	
	$\|Q\|$	$\|T\|$	$\|Q_{ts}\|$	$\|T_{ts}\|$	$\|Q_{sm}\|$	$\|T_{sm}\|$	$\|Q_{ts+sc}\|$	$\|T_{sm}\|$
GameChannel	7	17	7	17	7	12	7	17
CryptoPunksMarket	5	18	5	16	4	10	5	17
SaleClockAuction	2	5	2	4	2	3	2	4
Supeflaze	5	20	5	10	5	13	5	16
RpsGame	4	5	NA	NA	4	5	4	5
MoonCatRescue	10	58	6	13	9	34	9	35

Table 4 shows the comparison results of the state and transition coverage on the ground truth by the existing test suites and the test sequences generated from the mined automata. The columns "Test suites", "SmCon", and "Test suites + SmCon", show the number of covered states and transitions by the existing test suites, SmCon, and their combination, respectively. The existing test suites have better coverage than SmCon in some cases, but SmCon is better for others.

Table 4 shows that the test suite of MoonCatRescue has insufficient test cases covering only 6 out of 10 states and 13 out of 58 transitions. Interestingly, SmCon covered 9 states and 34 transitions. Meanwhile, when test suites are missing, SmCon can be used to generate new test cases for RpsGame from the scratch. Moreover, the combination of SmCon and the existing test suites contributes to higher state and transition coverage. These results indicate SmCon can generate test cases as good as those written by developers, achieving higher test coverage.

It will be appreciated that many further modifications and permutations of various aspects of the described embodiments are possible. Accordingly, the described aspects are intended to embrace all such alterations, modifications, and variations that fall within the spirit and scope of the appended claims.

Throughout this specification and the claims which follow, unless the context requires otherwise, the word "comprise", and variations such as "comprises" and "comprising", will be understood to imply the inclusion of a stated integer or step or group of integers or steps but not the exclusion of any other integer or step or group of integers or steps.

The reference in this specification to any prior publication (or information derived from it), or to any matter which is known, is not, and should not be taken as an acknowledgment or admission or any form of suggestion that that prior publication (or information derived from it) or known matter forms part of the common general knowledge in the field of endeavour to which this specification relates.

Claims

1. A method for smart contract specification mining, comprising:
receiving a transaction history for a smart contract;
slicing the transaction history to produce one or more invocation traces;
determining one or more predicates, each predicate belonging to a condition of a function of the smart contract;
constructing a finite state machine over states in the one or more invocation traces and the one or more predicates; and
performing counterexample-guided abstraction refinement to extend the finite state machine satisfying a minimal existential abstraction property.

2. The method of claim 1, wherein constructing the finite state machine comprises assuming an automaton comprising a set of states and a set of transitions between states in the set of states.

3. The method of claim 2, wherein performing counterexample-guided abstraction refinement comprises:
identifying a number of spurious paths resulting from applying the one or more invocation traces to the finite state machine; and
extending the finite state machine by splitting at least one state in the set of states, into two or more states, to reduce the number of spurious paths.

4. The method of claim 2 or 3, wherein performing counterexample-guided abstraction refinement comprises:
identifying a number of spurious paths resulting from applying the one or more invocation traces to the finite state machine; and
extending the finite state machine by removing at least one transition from the set of transitions, to reduce the number of spurious paths.

5. The method of claim 3 or 4, comprising repeatedly performing counterexample-guided abstraction refinement until the number of spurious paths is zero.

6. The method of any one of claims 1 to 5, wherein slicing the transaction history into one or more invocation traces comprises slicing the transaction history into a plurality of independent invocation sequences via parametric bindings.

7. The method of claim 6, wherein slicing the transaction history comprises performing a plurality of slices, the method comprising, for each slice, filtering out all events that are irrelevant to a parameter instance of a parameter in the parametric bindings.

8. The method of claim 7, wherein an event is irrelevant to a parameter instance if the event does not take the parameter instance as input, manipulate the parameter instance nor output the parameter instance.

9. The method of any one of claims 6 to 8, wherein slicing the transaction history comprises performing a plurality of slices, the method comprising, for each slice, forgetting parameter bindings for parametric events.

10. The method of any one of claims 1 to 9, wherein determining one or more predicates comprises:
decoding, from the transaction history and before and after each function invocation, one or both of contract state variables values and user-provided inputs; and
statistically inferring dynamic invariants for each function, each predicate comprising a said dynamic invariant.

11. A system for smart contract specification mining, comprising:
memory; and

at least one processor (processor(s)),
the memory storing instructions that, when executed by the processor(s), cause the processor(s) to:
receive a transaction history for a smart contract;
slice, using a history slicing module, the transaction history to produce one or more invocation traces;
determine, using a predicate extractor, one or more predicates, each predicate belonging to a condition of a function of the smart contract;
construct, at a finite state machine module, a finite state machine over states in the one or more invocation traces and the one or more predicates; and
perform, using a refinement module, counterexample-guided abstraction refinement to extend the finite state machine satisfying a minimal existential abstraction property.

12. The system of claim 11, wherein the finite state machine module constructs the finite state machine by assuming an automaton comprising a set of states and a set of transitions between states in the set of states.

13. The system of claim 12, wherein the refinement module performs counterexample-guided abstraction refinement by:
identifying a number of spurious paths resulting from applying the one or more invocation traces to the finite state machine; and
extending the finite state machine by splitting at least one state in the set of states, into two or more states, to reduce the number of spurious paths.

14. The system of claim 12 or 13, wherein the refinement module performs counterexample-guided abstraction refinement by:
identifying a number of spurious paths resulting from applying the one or more invocation traces to the finite state machine; and

extending the finite state machine by removing at least one transition from the set of transitions, to reduce the number of spurious paths.

15. The system of claim 13 or 14, wherein the refinement module is configured to repeatedly perform counterexample-guided abstraction refinement until the number of spurious paths is zero.

16. The system of any one of claims 11 to 15, wherein the history slicing module slices the transaction history into one or more invocation traces via parametric bindings.

17. The system of claim 16, wherein the history slicing module slices the transaction history by performing a plurality of slices and, for each slice, filters out all events that are irrelevant to a parameter instance of a parameter in the parametric bindings.

18. The system of claim 17, wherein an event is irrelevant to a parameter instance if the event does not take the parameter instance as input, manipulate the parameter instance nor output the parameter instance.

19. The system of any one of claims 16 to 18, wherein the history slicing module slices the transaction history by performing a plurality of slices and, for each slice, forgets parameter bindings for parametric events.

20. The system of any one of claims 11 to 19, wherein the predicate extractor determines one or more predicates by:
decoding, from the transaction history and before and after each function invocation, one or both of contract state variables values and user-provided inputs; and
statistically inferring dynamic invariants for each function, each predicate comprising a said dynamic invariant.

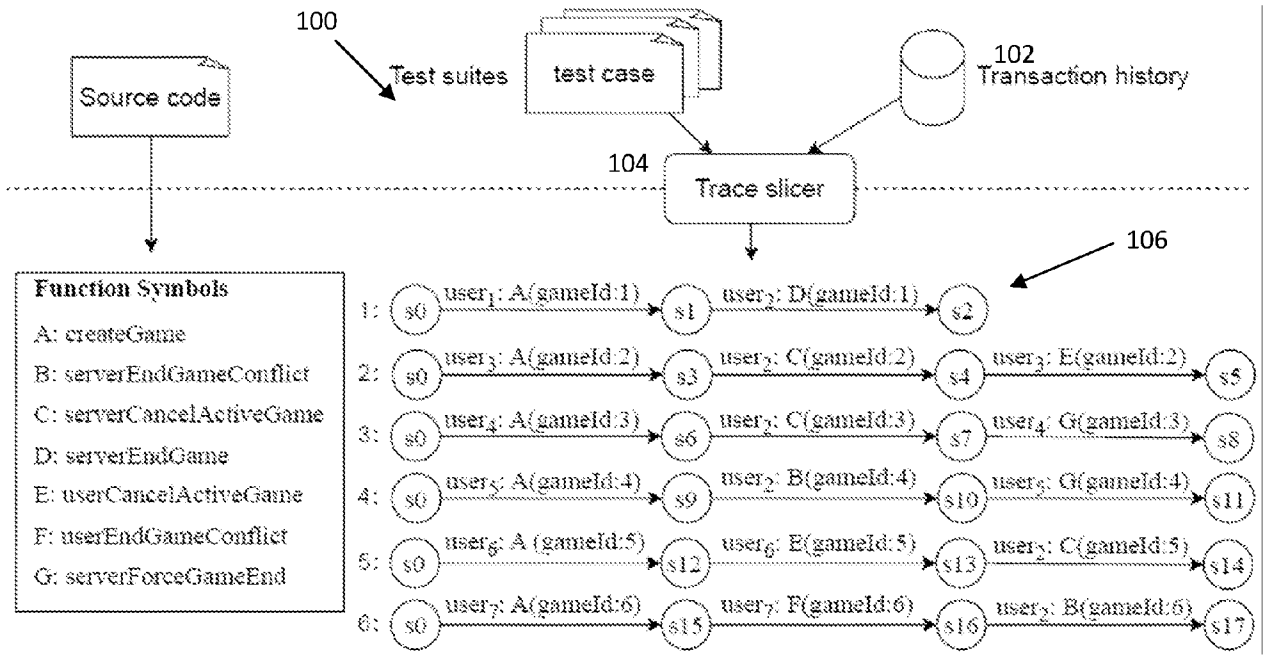
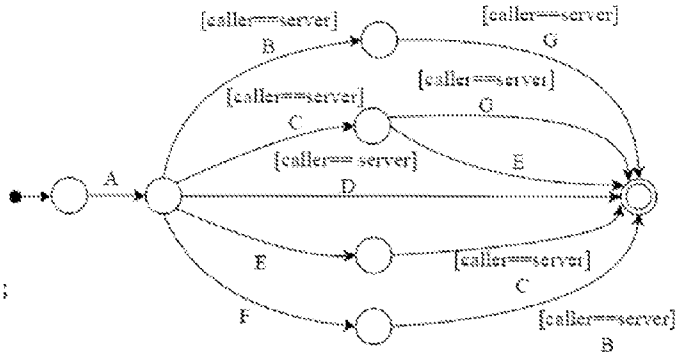


Figure 1

```
1  contract GameChannel{
2    enum GameStatus {
3      ENDED, ACTIVE, USER_INITIATED_END,
       ↪ SERVER_INITIATED_END
4    }
5    struct Game {
6      GameStatus status;
7      uint128 stake;
8      uint32 roundId;
9      uint endInitiatedTime;
10   }
11   // @dev Game id counter.
12   uint public gameIdCntr = 0;
13   address public server;
14   mapping (uint => Game) public gameIdGame;
15   // functions ...
16 }
```

(a) State variables of GameChannel.



(b) The mined automaton.

Figure 2

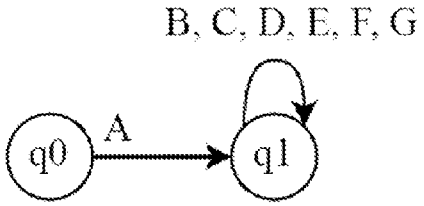


Figure 3

Figure 4

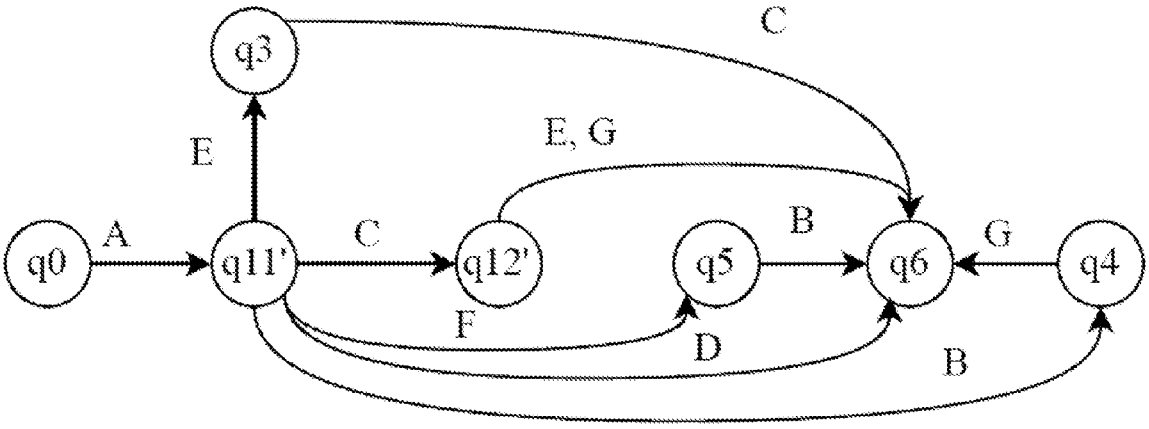


Figure 5

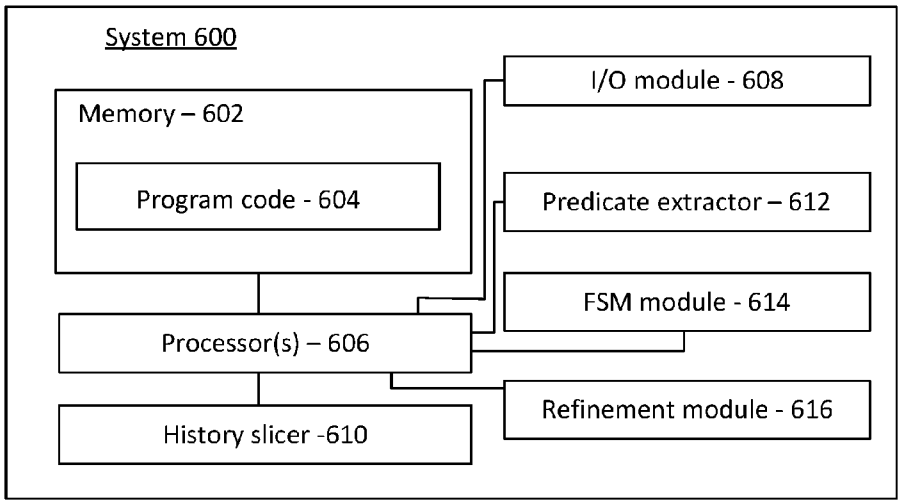

System 600
Memory – 602
Program code - 604
I/O module - 608
Predicate extractor – 612
FSM module - 614
Processor(s) – 606
Refinement module - 616
History slicer -610

Figure 6

INTERNATIONAL SEARCH REPORT

International application No.
PCT/SG2024/050440

A. CLASSIFICATION OF SUBJECT MATTER

H04L 9/32 (2006.01) *G06Q 30/00 (2023.01)*

According to International Patent Classification (IPC)

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

H04L, G06Q

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

Database: FAMPAT, Google Scholar

Keywords: smart contract, specification mining, transaction history, traces, predicates, condition, finite state machine, counterexample, minimal existential abstraction property, 智能合约, 规格, 推断, 交易, 历史, 跟踪, 谓词, 有条件, 有限状态机, 反例, 最小存在抽象 and other related terms

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	LIU Y., A Unified Specification Mining Framework for Smart Contracts. *ASE '22: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 5 January 2023, article 190, pages 1-3 [Retrieved on 2024-09-12] <DOI: 10.1145/3551349.3559512> sections 1, 2.1-2.3	1, 2, 6-12, 16-20
Y	LORENZOLI D. ET AL., Automatic generation of software behavioral models. *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, 10 May 2008, pages 501-510 [Retrieved on 2024-09-12] <DOI: 10.1145/1368088.1368157> section 3, page 4	1, 2, 6-12, 16-20

☒ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

*Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"D" document cited by the applicant in the international application

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
12/09/2024 *(day/month/year)*	13/09/2024 *(day/month/year)*

Name and mailing address of the ISA/SG	Authorized officer
Intellectual Property Office of Singapore 1 Paya Lebar Link, #11-03 PLQ 1, Paya Lebar Quarter Singapore 408533 Email: pct@ipos.gov.sg	Danny Yap Ming Ann (Mr) IPOS Customer Service Tel. No.: (+65) 6339 8616

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	AARTS F. ET AL., Automata Learning Through Counterexample Guided Abstraction Refinement. *18th International Symposium on Formal Methods, Paris, France, August 27-31, 2012. Proceedings*, 31 August 2012, pages 10-27 [Retrieved on 2024-09-12] <DOI: 10.1007/978-3-642-32759-9_4> the whole document especially page 3	
A	LIU Y. ET AL., InvCon: A Dynamic Invariant Detector for Ethereum Smart Contracts. *ASE '22: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 5 January 2023, article 160, pages 1-4 [Retrieved on 2024-09-12] <DOI: 10.1145/3551349.3559539> the whole document especially the abstract	
A	GUTH F. ET AL., Specification Mining for Smart Contracts with Automatic Abstraction Tuning, arXiv:1807.07822v1 [cs.SE]. *Computer Science Preprint Archives*, 20 July 2018, pages 1-12 [Retrieved on 2024-09-12] <DOI: 10.48550/ARXIV.1807.07822> the whole document	